計算機システムの安定的な稼働に関する
ソフトウェアの研究

提出先　　大阪大学大学院 情報科学研究科
提出年月　2016年1月

植田　良一

業績リスト

関連発表論文

論文

[1-1] 植田良一, 角井健太郎, 爲岡啓, 松下誠, 井上克郎, “Webサービスシステムの応答性能劣化診断のための学習データ自動選定方法”, 電子情報通信学会和文論文誌 D(採録決定), 2015.

[1-2] 植田良一, 練林, 井上克郎, 鳥居宏次, “再帰を含むプログラムのスライス計算法”, 電子情報通信学会論文誌 D-I, Vol.J78-D-I, No.1, pp.11-22, 1995.

国際会議

[1-3] Ryoichi Ueda, M. Hiltunen, R. Schlichting, “Applying Grid Technology to Web Application Systems”, Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 1, pp.550-557, Cardiff, UK, 2005.

研究会発表等

[1-4] 爲岡啓, 植田良一, 松下誠, 井上克郎, “ベイジアンネットワークとクラスタリング手法を用いたシステム障害検知システムの有効性検証”, 情報処理学会研究報告, Vol.2015-SE-188, No.4, pp.1-8, 2015.

[1-5] 佐藤慎一, 植田良一, 井上克郎, “再帰やポインタを含むプログラムの効率的な依存関係解析法の提案”, 電子情報通信学会ソフトウェアサイエンス研究会報告 SS95-37 p.9-16, 1996.

[1-6] 植田良一, 練林, 井上克郎, 鳥居宏次, “再帰を含むプログラムの依存関係解析とそれに基づくプログラムスライシング”, 電子通信学会ソフトウェアサイエンス研究会報告 SS94-5 p.33-40, 1993.

その他の論文

論文

[2-1] 柳井孝介, 植田良一, 佐川暢俊, “大規模分析のための木構造データ処理プラットフォーム”, 人工知能学会論文誌, Vol.26, No.5, pp.594-606, 2011.

[2-2] 田中哲雄, 植田良一, 相薗敏子, 牛嶋一智, 内藤一郎, 薦田憲久, “情報のメタデータに着目した情報ライフサイクル管理向けポリシー記述方式”, 電気学会誌C部門論文誌, Vol.126, No.4, pp.498-505, 2006.

研究会発表等

[2-3] 植田良一, 佐藤嘉則, 森正勝, 中村浩三, 佐川暢俊, “社会インフラの革新に貢献する知識化サービス基盤 KaaS”, 日立評論, Vol.92(5), pp.36-39, 2010.

内容梗概

スマートフォン等の普及を背景に，日常生活の多くの場面でオンラインサービスを利用するようになった現在，そのサービスを支える計算機システムの安定稼働の重要性が増している．

本論文では，オンラインサービスシステムの安定的な稼働に資する以下の3つの手法を提案する．

(1) Webサービスシステムの応答性能劣化診断のための学習データ自動選定方法[1-1]

計算機システムのCPU／メモリ等のメトリクス値を元に，システムの状態，特に応答性能異常を検知するために，機械学習を応用した手法が提案されているが，学習に使うデータの選定を人手で注意深く行う必要があるという問題がある．

そこで本研究では，選別労力をかけずに選んだ初期学習データから開始し，観測データの中から自動的に学習データに追加すべきデータを選別する方法を提案する．本方法を採用した場合と，選別なしに全データを学習した場合とで同等の診断結果が得られる事を確認した．

(2) 計算機システムの処理性能を拡大するGrid技術の，オンラインWebサービスシステムへの適用方法[1-3]

本研究では，Webサービスシステムに Gridコンピューティング技術を適用する際の2つの課題解決に取り組んだ．ひとつはWebアプリケーションのGrid基盤上への移行方法，もうひとつはGrid基盤への移行がもたらす効果の判定である．これらの検証のため，Grid技術の動的リソース割当機能を活用するスケーラブルアーキテクチャを設計，J2EEインターネットバンキングアプリケーションをGrid化し，元のバージョンとの性能比較を行った．

その結果，Grid技術が，システム全体の管理性を向上するためのひとつの方法として有効であることが分かった．

(3) プログラムのバグを発見するために役立つ，プログラムスライスの計算方法[1-2]

本研究では,再帰を含むプログラムの依存関係の静的解析と,その結果に基づいてスライスを求めるためのアルゴリズムを提案する. 手続きの境界を越えて解析を行うために，独自の改良を施したプログラム依存グラフを定義し，計算結果が収束するまで解析を繰り返す方法を採用した．

本アルゴリズムに従ってスライスを計算／表示する試作システムを作成し,本アルゴリズムが正しく動作する事を確認した.

オンラインサービスシステムの安定稼働を達成するために，研究成果(1)は，システムレベルのマクロな視点から，システムの応答性能劣化の原因がリクエスト過多であるか，それとも，システムの障害であるかを判定する際に役立つ．

応答性能劣化の原因がリクエスト過多にある場合，研究成果(2)のGrid化技術を適用して，オンラインサービスシステムの処理能力を拡大することでこれに対処する事ができる．

また，応答性能劣化の原因がシステムの障害であった場合，今度はミクロの視点で研究成果(3)のスライス計算方法を活用して，システム障害原因のひとつとなるプログラム内のバグ発見を支援する事ができると考える．

謝辞

本研究の推進にあたり，常日頃より適切なご指導を賜わりました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上克郎教授に，心から深く感謝申し上げます.

本研究に関して，適切なご助言を頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻萩原兼一教授，楠本真二教授に深く感謝申し上げます.

本研究の推進にあたり，直接具体的なご助言とご指導を頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻松下誠准教授に心より御礼申し上げます.

本研究を進めるに当たり，ご協力いただいた，大阪大学大学院情報科学研究科爲岡啓氏,（株）日立製作所研究開発グループ角井健太郎氏に感謝申し上げます.

目次

図目次

表目次

第1章　はじめに

1.1　システムの安定的な稼働の重要性

スマートフォンをはじめとする高機能携帯端末の爆発的な普及を背景に，SNSやゲーム等の情報交換，娯楽関連のサービスだけでなく，商品の売買，銀行取引等生活の多くの活動を，いつでも，どこにいても行う事ができるようになった [1]. これら社会インフラの一部となったオンラインサービスを支える計算機システムは，大規模かつ複雑化する一方で，銀行のATMが数日にわたって利用できなくなる等 [2], サービス停止時の影響は年々大きくなり，その長期的な安定稼働の重要性が増している．

オンラインサービスシステムの安定稼働を阻害する要因は，(1) 処理性能を越えるリクエスト，(2) プログラムのバグ／設定ミス，(3) バックアップ等の運用による負荷，(4) 攻撃等が考えられる [3].

本論文では，これらオンラインサービスシステムの安定稼働の阻害要因のうち，(1) システムの処理性能に関するマクロな視点と，(2) プログラムのバグに関するミクロな視点の両面からアプローチする事を検討した．

1.2　システムの処理性能問題へのアプローチ

オンラインサービスシステムの処理性能問題（性能劣化）への対策には，発生時のシステムの状態を知る必要がある．すなわち，(a) 単にリクエスト量が多すぎて処理に時間がかかっているのか，それとも，(b) システムのどこかに発生した障害により，本来利用できるはずのリソースの一部が利用できなくなったため処理に時間がかかっているのか，を見極める必要がある．

前者の場合，システムの処理性能を動的に拡大して対処可能であるが，後者の場合，障害の箇所／原因を突き止めて対処する必要がある．この見極めを行うのがシステムの処理性能劣化の検知／原因診断技術，また，性能劣化の原因がリクエスト過多の場合の対処方法が，システムの処理性能の動的拡大／縮小技術である．

1.2.1　システムの処理性能劣化の検知／原因診断技術

大規模かつ複雑化したシステムでは，設定ミスや部分的な障害等による処理性能の劣化を早期に検知し，システム全体が停止に至る前に，その原因を究明し対処する必

要がある.

従来, システム障害の検知にはCPU利用率, ネットワーク通信量, Disk IO量等の個々のメトリクスの閾値越えを監視する方法がとられてきたが, システムの複雑化に伴って, 個々のメトリクスの監視だけでは不十分となってきた[4]. また, 熟練者が実際の運用を通じて獲得した知識を, 障害判断条件とその対応方法の組で表現したif-thenルールによる監視手法は, 障害の検知を行うためにある程度の効果はあるものの, ルール記述が困難, 人間が認識できない隠れたルールは記述できない, 等の課題がある[5]. そこで, 近年, 収集した大量のデータを機械学習等の統計処理を施すことで, 複雑に絡み合ったメトリクス間の関連をモデル化し, システムの正常／異常の判定, 異常の根本原因究明に活用する手法が提案されている[4, 5, 6, 7, 8, 9, 10]. 本論文では, 2章で, 本課題に対する解決策を提案／評価した結果を示す.

1.2.2 システムの処理性能の動的拡大／縮小技術

処理の開始前にあらかじめ処理量の正確な見積もりを行う事ができるバッチ処理システムとは異なり, オンラインサービスシステムでは単位時間当たりのリクエスト量を正確に見積もることは非常に困難である[11]. これに対処するために, 過去の最大リクエスト量を処理するのに十分なリソースを保持するのは, ほとんど利用されない余剰リソースを持つことになり, 資産効率の面で問題となる.

そこで, 変化するリクエスト量に応じてシステムの処理性能を拡大／縮小する方法が提案されている. 例えば, Web3階層システムでは, ユーザからのリクエスト量が増加し, リクエストに対する平均応答時間が長くなると, リクエストを受け付けるWebサーバの台数を増加させて並列処理することで, 応答時間の改善を図る手法（スケールアウト）が知られている[12]. 本論文では, 3章で本課題に対する解決策を提案／評価した結果を示す.

1.3 プログラムのバグ問題へのアプローチ

大規模かつ複雑化したシステム内で稼働するプログラムはサイズが大きく, プログラム内に潜むバグを発見／除去する事は困難である[13]. この困難性の削減に効果的なのが, 巨大なプログラムから, 注目する箇所や変数に影響を与える部分だけを抽出するプログラムスライシング技術[14]である. 抽出されたプログラムの断片をスライスと呼ぶ.

プログラムPのスライスとは, 直感的にはP中のある地点nおよびある変数vに対して, nにおけるvの値に影響を与えるP中の各文や式の集合を言う. すべての可能な入力データに対して解析したものを静的スライス, 特定の入力データに対して解析したもの

を動的スライスという[15](本論文では静的スライスのみを扱うので,以下単にスライスと言えばそれは,静的スライスを意味するものとする).

スライスの技法はMark Weiser[14]によって提案され,当初はプログラムのデバッグを支援するために使われていたが,現在では,デバッグだけでなくテストや保守,プログラム合成などにも利用されている[16, 17]. 本論文では，デバッグ時の利用を想定して，対話的にスライス計算結果をユーザに提示する手法の提案を目指す.

スライスの計算には,プログラム内の文の間の依存関係の正確な解析が必要であるが,再帰が存在するプログラムを正確に解析をするのは容易ではない[18].

Weiserは,スライスを計算するために,データフロー方程式(詳しくは4.3節参照)を使ったが,正確な計算ができるのはひとつの手続き内だけだった. Ottenstein[19]が,このスライスの計算をグラフ上の到達可能性問題に置き換える手法を考案した.この手法を使って,Horwitz[20, 21]が,手続きの境界を越えてスライスが計算できるアルゴリズムを紹介したが,このアルゴリズムでは各手続き内のデータフロー解析とスライス計算のためのグラフ作成等をいくつかのフェーズに分けて行なうため,実際のシステム作成には応用しにくい.また,Hwang[18]は,再帰を含むプログラムに対して最小不動点を求める手法を用いて,スライスを直接計算する方法を提案したが,この方法ではプログラム中の地点n,変数vごとに再帰方程式を解く必要が生じ，時間がかかるため，対話的にスライス情報を提供するシステムには組み込みにくい.

本論文では，4章にて，手続きの境界を越えてデータフローを解析できるように拡張したプログラム依存グラフを用い，さらに，再帰を含むプログラムにも適用可能となるように解析方法を工夫することで，本課題を解決する方法およびその評価結果を示す.

1.4 本論文の概要

本論文では，前節であげた課題を解決するために，以下の3つの手法を提案し，提案手法を実装／評価した結果を述べる.

(1) マクロ視点 システムレベルの視点で，システム処理性能の劣化を検知する方法，および，システム処理性能をリクエスト量に応じて拡大／縮小する方法

(a) Webサービスシステムの応答性能劣化診断のための学習データ自動選定方法 [1-1]

Webサービスシステムの CPU／メモリ等を対象としたメトリクス値を元に，システムの状態，特に応答性能異常の検知に寄与する技術を提案する. 計算機システムの異常検知の方法は，熟練者が見出した相関に基づくルールベース方式から，過去の運用データから機械学習等の統計処理にて生成したモデルに基づいて異常の診断を行う統計処理方式へと変化しつつある. 機械学習を

応用する方法では，診断結果の精度を上げるために学習データの量を十分に多くする必要があるが，計算量が多くなるため，やみくもに過去の全データを学習に使うことは現実的ではない．

そこで本論文では，選別労力をかけずに選んだ初期学習データから開始し，観測データの中から自動的に学習データに追加すべきデータを選別し，これが出現するたびに学習データへ自動的に組み入れ，モデルを逐次更新する方法を提案する．選別を行わずに全データを追加して学習させた場合と，本論文で考案する方法で選定した部分データのみ追加して学習させた場合とで，診断結果を比較する実験を行った．その結果，本論文の方法を採用した場合，10種中7種の実験で，全データで学習した場合と同等の診断結果が得られる事を確認した．

(b) 計算機システムの処理性能を拡大するGridコンピューティング技術の，オンラインWebサービスシステムへの適用方法[1-3]

Globus ToolkitのようなGridミドルウェアや，Open Grid Services Architecture (OGSA)のようなGrid標準は，分散企業アプリケーションを構築するのに十分な能力を備えている事を期待されているが，このような用途での活用はまだ十分になされていない．

企業アプリケーションの多くがWebアプリケーションである現状を踏まえ，本論文では，これにGrid技術を適用することに関連する2つの課題解決に取り組んだ．ひとつめは，既存のミドルウェアがWebアプリケーションをGrid基盤上に移動させるのに有効かどうか，もし有効ならその方法の発見である．ふたつめは，このようなGrid化は性能拡張や管理容易性の観点で，優位性をもたらすのかどうか，である．これらの検証のため，J2EEインターネットバンキングアプリケーションをGrid化し，元のバージョンとの性能比較を行った．さらに，変動するリクエスト量に対応するためにGrid技術の動的リソース割当機能を活用するスケーラブルアーキテクチャを設計した．基本的な性能計測の結果，本アーキテクチャはリクエスト量の変化に応じてスケールすること，すなわち，Grid技術が，システム全体の管理性を向上するためのひとつの方法として有効であることが分かった．

(2) ミクロ視点 システム内で稼働するプログラムの視点で，システムの安定稼働を阻害するバグの発見に寄与するプログラムスライスの計算方法

(c) プログラムのバグを発見するために役立つ，プログラムスライスの計算方法[1-2]

本研究では,再帰を含むプログラムの依存関係の静的解析と,その結果に基づ

いてスライスを求めるためのアルゴリズムを提案する.

本アルゴリズムでは,再帰的な手続き定義に対応するために,計算に必要な情報をあらかじめ仮定し,その情報をより正しいものへと変化させ,それが収束するまで解析を繰り返すという方法をとった.さらに,手続きの境界を越えてスライスを計算できるように,独自の改良を施したプログラム依存グラフを定義した.

このアルゴリズムは,従来のものに比べ,再帰を含むプログラムを効率的に解析する事ができ,対話的にスライス情報を提供するシステムに組み込む事ができる.

本アルゴリズムに従ってスライスを計算し,表示する試作システムを作成し,本アルゴリズムが正しく動作する事を確認した.このシステムはSUN SPARCstation ELC上で動作し,13個の手続き,7個の大域変数を含む再帰呼び出しのない249行のプログラムを0.80秒で,また,8個の手続き,9個の大域変数を含み,すべての手続きがお互いに呼びあう可能性のある275行のプログラムを1.82秒で,それぞれ解析する事ができる.

1.5 各章の構成

2章では，Webサービスシステムの応答性能劣化診断のための学習データ自動選定方法について述べる．3章では，Grid技術のWebサービスシステムへの適用について述べる．4章では，再帰を含むプログラムのスライス計算方法について述べる．最後に5章で，研究成果のまとめと今後の研究について述べる．

第2章　Webサービスシステムの応答性能劣化診断のための学習データ自動選定方法

2.1　まえがき

本章では，Webサービスシステムの応答性能劣化診断のために，機械学習を適用する際の，学習データの自動選定方法を提案する．

これまでに提案された，機械学習によるシステムの異常診断方法の代表的な2つを以下で紹介する．

(a) ベイジアンネットワークによる異常診断 [4, 5]

ベイジアンネットワーク（以下BN）は，個々の事象の因果関係を条件付き確率で表すモデルで，観測対象の過去の状態を学習し，観測対象がある状態Sにある時の注目事象E（例：応答時間が3秒以上となる）の発生確率P_Eを算出する事ができる．ただし，学習データに含まれる過去の状態と同じとみなされる既知の状態での注目事象の発生確率は計算できるが，学習データに含まれない未知の状態下での注目事象の発生確率は正しく計算できない．注目事象の発生確率を正しく計算するためには学習データの量を増やすことが効果的である．しかし，データ量が増えると学習処理にかかる時間が増大し，実用的な時間で完了しなくなるという課題がある．学習データに含まれる観測項目数を一定にした場合，計測回数に比例して学習時間が長くなる [22].

(b) クラスタリングによる異常診断 [6]

クラスタリング（以下CL）は，観測対象の過去の状態をグルーピングし，いくつかのクラスタに分割する方法のひとつである．教師データが存在しない場合，個々のクラスタに対する意味付け（例：正常時のクラスタ等）は当該クラスタに含まれる観測値から人間が判断する．入力（学習）データとして正常時のデータのみ与え，最近傍クラスタからの距離が閾値を越える観測値を異常と判定する診断方法が知られているが [5]，正常範囲の一部しか学習データに含まれない場合，正常であるにもかかわらず異常と判定してしまう．このため，異常データを含まず，かつ，正常範囲内でなるべく広い範囲を含む入力データが必要となる．この要件が十分に

満たされないと，検知漏れや誤報を引き起こす可能性が高くなる [6].

2.1.1 既知の方法の課題

機械学習を活用する方法の多くは，期待する効果を得るために適切に学習データを選定しなければならない，という共通の課題がある．現在は，学習パラメータの調整と合わせて，学習データの選定を人間が試行錯誤で行う事で，この課題に対処している．

2.1.2 提案手法の発想

これまでの経験から，Web サービスシステムの運用中のほとんどの時間は安定した正常状態の範囲内にとどまり，ごく稀にこの正常状態からの逸脱が発生すると考えている．このような状況では，システムの状態に大きな変化がない期間の膨大な観測データの全てを学習データに利用することは効率的ではない．そこで，本論文では学習データの選定を自動化する方法を模索した．BN の学習データに，同じ状態が繰り返し出現する回数を減らしつつ，異なる状態を多数含めることができれば，より少ないデータ量で，全データで学習した場合と同等の学習効果を得る事ができると考えた．刻々と収集される観測データの異常診断を行うと同時に，現在の状態が学習データに含まれているかどうかの判定を行い，現在の学習データに含まれていないと判定した場合に，その期間のデータを学習データに追加して再学習する，という一連の動作を繰り返せば，より少量でありながら多種の状態を含む学習データを作り上げることができるとの仮説を立案した．現在の学習データに含まれていない状態を検知するには，BN と同じ学習データを CL に学習させ，現在の計測値と最近傍クラスタとの距離を計算することで実現できる．よって，BN と CL を上記のように組み合わせることで，学習データを自動選定する異常検知システムが実現できると考えた．

2.2 学習データ自動選定方法

本論文で提案する，学習データの自動選定手法の手順を図 2.1 に示す．本手法では一定の時間分の連続した観測データの集合をデータ区画と呼び，学習データとしての追加要否を判定する最小単位としている．はじめに，(1) あらかじめ手動で選定した初期学習データを CL で学習，モデルを生成し，診断を開始する．次に (2) 監視対象システムの現在の状態を収集し，CL にて最近傍クラスタからの距離を計算し，記録する．次に (3) 当該データ区画の最後まで到達したら，(4) 直近のデータ区画の CL での診断結果の平均が閾値を越えていないか検査する．閾値を越える事は，監視対象システムが，当該データ区画の間，現在の学習データに存在しない未知の状態にあった事を意味する．ゆえに，閾値を越えていた場合，(5) 当該データ区画を学習データに追加し，新たな学習

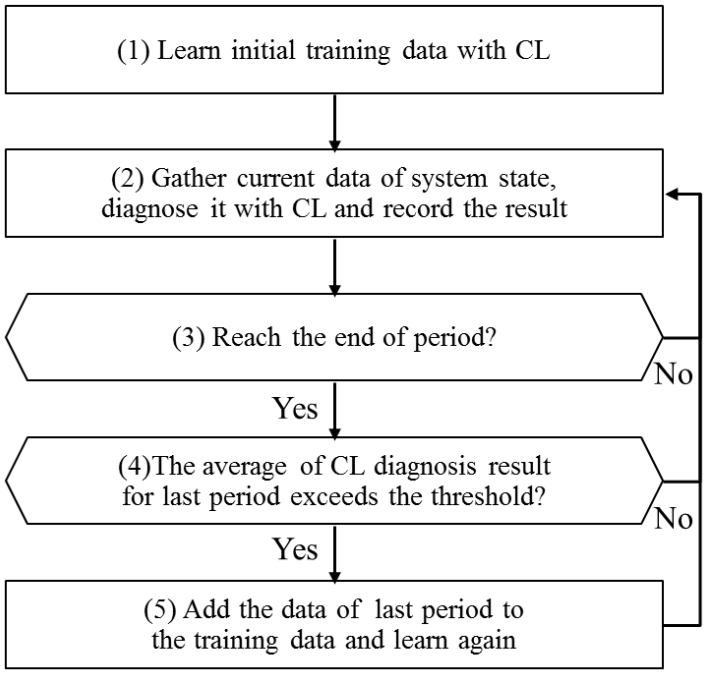

図 2.1: 学習データ自動選定手順

データで再度学習を行う. この再学習プロセスにより, 新たな状態を含んだ学習データによるモデルが生成される. 上記 (2) から (5) を繰り返すことで, 徐々に, CL が未知と判定する頻度が減少し, 学習データの集合が収束に向かい, 全データと比較して少量の学習データの集合を作り上げる事ができる.

2.3 提案手法の有効性検証

本研究の目的は, 全データを学習した場合と同等の診断結果が得られるような, より少ないデータの集合を自動的に選定する事にある. 提案手法の有効性を検証するために, 全データを学習データに追加した場合と, 提案手法によって自動選定した学習データのみ追加した場合の, それぞれの BN による診断結果 (平均応答時間が 1 秒を越える確率の時系列データ) のユークリッド距離と相関係数を比較することとした. 全データを学習した場合の診断結果の移動平均を $W = (w1, w2, w3, ...)$, 提案手法を適用した場合の診断結果の移動平均を $P = (p1, p2, p3, ...)$ とし, ユークリッド距離 E, 相関係数 C をそれぞれ以下の計算式で算出した. $\overline{w}$ は $w_1, w_2, ..., w_n$ の平均値を表す. $\overline{p}$ も同様.

$$E = \sqrt{\sum_{i=1}^{n}(w_i - p_i)^2} \tag{2.1}$$

$$C = \frac{\sum_{i=1}^{n}(w_i - \overline{w})(p_i - \overline{p})}{\sqrt{\sum_{i=1}^{n}(w_i - \overline{w})^2}\sqrt{\sum_{i=1}^{n}(p_i - \overline{p})^2}} \tag{2.2}$$

これにより，学習データ自動選定結果の善し悪しを判断する．

2.3.1 評価方法

JPetStore[23]（オンラインペット販売サイトアプリケーション）に，実アクセスパターンに基づくバックグラウンドトラフィックを与えながら，同時に，システムの異常状態を模擬する負荷を，システムを構成するサーバの一部に与える．監視対象システムを構成する全てのサーバで監視エージェントが稼働しており，一定間隔で観測データを監視サーバに集約される．実際の負荷パターンとして，WorldCup98[24]で提供されるWebサーバへのアクセスデータの一部を活用した．システム全体の処理能力の差を考慮して秒間リクエスト数を500分の1程度に縮約し，リクエストの増減パターンのみ維持した．また，システムの異常状態を模擬するためにstressコマンドにて負荷を与えた．はじめに，あらかじめ収集されたデータの中から任意の初期学習データを抽出し，CLにて学習を行い，モデルを生成する．次に，現在のシステムの状態をCLにて診断し，記録しておく．観測時刻がデータ区画の末尾に到達した際に，直近のデータ区画の，CLでの診断結果の平均が閾値を越える場合，未学習であると判断し，当該データ区間のデータを学習データに追加し，再学習を行う．以降の診断は再学習で生成したモデルにて行う．この処理を一定期間繰り返し実行後，生成された学習データが，提案手法にて自動選定された学習データである．最後に，自動選定された学習データの善し悪しを判定するために，全データを学習した場合と，提案手法にて自動選定した学習データのみ学習した場合の，それぞれのBNによる診断結果のユークリッド距離と相関係数を比較する．

2.3.2 実験システム構成

実験環境（図2.2）は，(1)監視対象システム，(2)トラフィック発生部(A)，(3)システム異常検知部(E)，からなる．監視対象システムはロードバランサ(B)，Web-APサーバ(C1)(C2)，DBサーバ(D)からなり，JPetStoreアプリケーションが稼働している．各サーバには，CPU/ Memory/ Network/ Diskの利用状況データを10秒間隔で収集するための監視エージェント(collectd)が組み込まれており，システム異常検知部(E)に観測データを送付する．トラフィック発生部(A)は，あらかじめ与えたシナリオ（例：商品検索→1商品をカートに入れる→チェックアウト）と負荷パターンに応じて，ユーザの挙動を模擬するリクエストを発生させる．

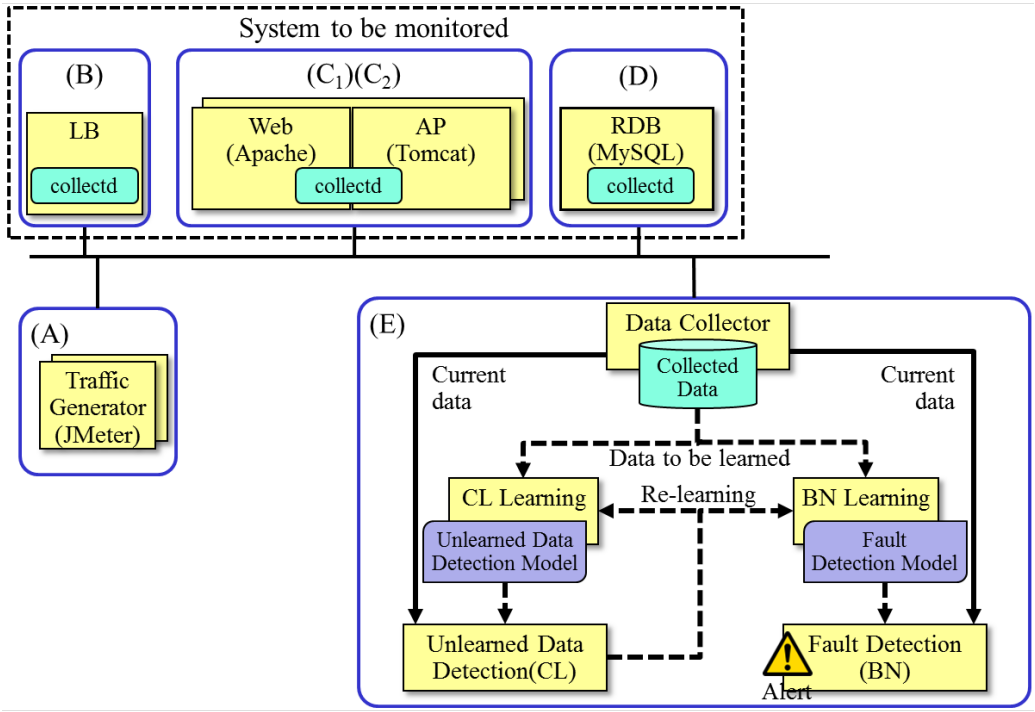

図 2.2: 実験システムの構成

収集データの一覧を表2.1に示す．今回の実験では，LBで#1から#3と#5までの7項目，2台のWebサーバそれぞれで#1から#4までの合計8項目，DBサーバで#1（2コア分）と，#2, #3, #4, #6の合計7項目，計22項目のシステム監視で一般的なメトリクスを収集した．監視対象システムには，指定時刻になると，当該システムの性能を劣化させるstressコマンドが実行されるように設定した．システム異常検知部は，各サーバ内の監視エージェントが収集する観測データを蓄積し，常時異常検知を行いつつ，必要に応じて再学習を行う．今回の実験では仮想サーバ環境上に，CPU:1core, Memory:1GB, HDD:20GB, Network:1Gbpsのリソースを持つ2つのWeb/APサーバ，DBサーバ，LBの合計4サーバを使った．DBサーバだけは2コアを利用した．各仮想サーバ同士がリソースを取りあう事がないように異なる物理サーバ上に仮想サーバを作成した．OSは全サーバでLinux(CentOS 6.5)を採用した．また，BNの実装として，統計解析環境R[25]用ライブラリbnlearn[22]を，CLの実装としてR標準ライブラリのk-meansを活用した．CLのクラスタ数は10を指定した．システムの状態数に相当するクラスタ数は過去の運用の経験，および，事前のデータ分析により10から20程度が妥当であると判断したため，モデルが最も単純となる10を採用した．

表 2.1: 監視対象メトリクス

#	Resource	Monitoring Items
1	CPU(DB has 2 cores, others have 1 core)	User usage for each core(%)
2	Memory	Used (bytes)
3	Network(LB has 2)	Used (bps, sent+received)
4	Disk(except LB)	IOPS (ops/sec)
5	Web Access(only LB)	(a)Request count, (b)average and (c)max response time(sec)
6	DB Access(only DB)	(a)Processed data (bytes), (b)Written data (bytes)

2.3.3 実験方法

以下の (1)から (4)に示す一連の実験プロセスを 1セットとして，合計 10セットの実験を行った．10セット中 5セットは，初期学習期間を変化させ，診断期間を共通とした．残り 5セットは逆に初期学習期間を共通とし，診断期間を変化させた．

(1) 初期学習データの生成

WorldCup98のアクセスパターンの縮約比率を変化させて，(a)監視対象システムが平均応答時間 1秒以下を維持できる程度のリクエストを処理している状態，(b)リクエストがほとんどない状態，(c)平均応答時間が 1秒を越える状態，の 3つの状態が繰り返し発生する 5時間分のデータを用意した．その中から図 2.3に示すように，(a)(b)(c)の 3状態が現れる 50分間のデータを異なる時間帯から選択し，初期学習データとして活用した．この 5時間の間にはシステム障害を模擬する負荷は与えなかった．

(2) サーバ異常の模擬生成

WorldCup98のアクセスパターン（図 2.4）を再現したトラフィックをバックグラウンドトラフィックとして与えている状態で，検知すべき障害を模擬する異常を各サーバ内に作り出した．今回の実験では，CPU, Memory, HDDの各リソースに対して stressコマンドにてユーザリクエスト処理を阻害する負荷をかけた．

(3) 提案手法による追加学習データの自動選定

上記 (2)を実行中に 10秒毎に収集される，40分間の観測データに対し，BNにて平均応答時間が 1秒を越える確率を逐次診断する．この診断と同時に CLでの最近傍クラスタからの距離計算も行い，5分毎に直近 5分間の距離の平均が閾値 6を越えるかどうかを判定，越えた場合は学習データに当該 5分間分のデータを追加し，再度学習を行う．本実験では，各クラスタの重心ベクトルと 1回分の観測データのベクトルを用いて，正規化後ベクトル間のユークリッド距離の最小値を，当該観測の

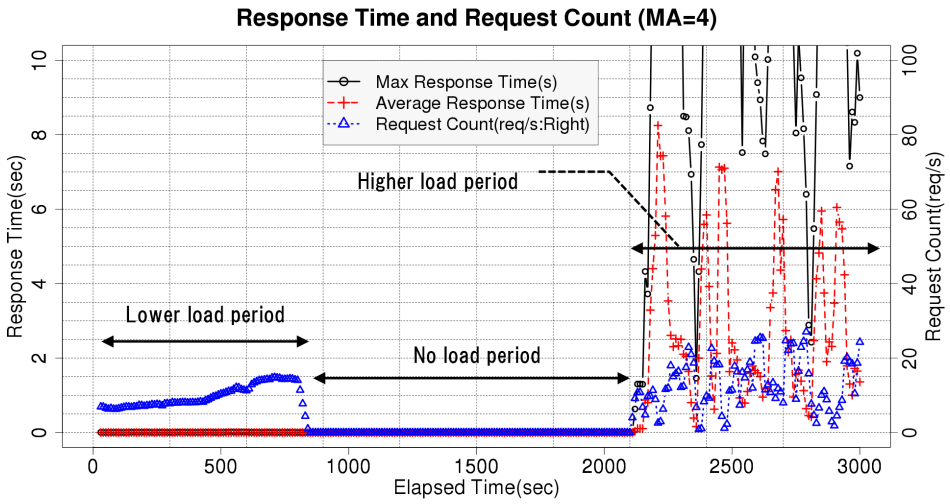

図 2.3: 初期学習データのリクエスト量

最近傍クラスタからの距離として利用した．以降の観測データに対しては，再学習で生成されたモデルで診断を行う．50 分間の初期学習データに加え，最大で 8 区画 40 分間のデータが学習データに追加される．今回の実験では，後半 8 区画 40 分中の，連続しない 3 区画でシステム異常を模擬する 5 分間の負荷を，2 台の Web サーバと DB サーバのうちいずれか 1 台，または 2 台，または 3 台全てに与えた．診断期間を共通とした 5 セットの実験では，第 2 区画で Web サーバ Web_A に，第 4 区画ではもう一方の Web サーバ Web_B に，第 6 区画では 2 つの Web サーバに加えて，DB サーバにも負荷を与えた（図 2.5）．診断期間を変化させた 5 セットの実験では，連続しない 3 区画で上記負荷を順不同で与えた．

(4) システム異常検知結果の比較

CL にて 40 分間の診断を終えた後，生成された学習データの善し悪しを判定するために，(a) 全データを BN にて学習し，後半 40 分を診断した結果（以下 Whole）と，(b) 上記 (3) にて自動選定した学習データを BN にて学習し，(a) と同一データを診断した結果の類似性を判定した．類似性判定には，ユークリッド距離と相関係数を採用した．また，学習データの追加の効果を見るために，学習データへの追加を行うたびに，新たな学習データによる診断結果と Whole との類似性を評価した．異常検知結果の例を図 2.5 に示す．上段グラフの丸点が BN による平均応答時間が 1 秒を超える確率 P（左軸），三角点が CL による最近傍クラスタからの距離 D（右軸），横軸は経過時間（秒）を表す．下段グラフの〇点が 10 秒間のリクエストの最大応答時間（左軸：秒），三角点が同 10 秒間の平均応答時間（左軸：秒），＋点が同 10 秒間の

図 2.4: バックグラウンドトラフィックパターンの例

処理リクエスト数（右軸：リクエスト／秒），横軸は経過時間（秒）を表す．各グラフタイトルの MA=4 はグラフ上の点が隣接 4 データの移動平均値である事を表す．

2.3.4 実験結果

表 2.2 に 1 セット分の実験結果を示す．1 セットの実験は初期学習データのみから生成し

表 2.2: 実験結果: 1 回分

Heat ID	Training Data	Avg Dist.	Euclid.	Correlation
Base	Base	–	5.36	-0.16
L1	Base+0-5	20.49	3.21	0.34
L2	L1+5-10	21.97	3.03	0.38
L3	L2+15-20	6.02	0.91	0.66
L4	L3+25-30	8.79	0.16	0.98

たモデルで診断した結果 (Base) で始まり，1 データ区画を学習データに追加して全データを診断した結果 (L1)，さらに 1 データ区画を追加して全データを診断した結果 (L2)，と学習データを追加するたびに番号を増やして区別できるようにした．表中の各行を Heat と呼ぶ．表中 Heat ID は当該実験の Heat 名を，Training Data は当該 Heat の学習データを，Avg Dist. は当該 Heat で初期データに組み入れたデータ区画の最近傍クラスタからの距離の平均を，Euclid. と Correlation は全データで学習したモデルでの BN の診断結果と，当該 Heat のモデルでの BN の診断結果間のユークリッド距離と相関係数を，それぞれ表す．Euclid. が 0 に近付き，Correlation が 1 に近付くほど，Whole との類似性が高い．例えば表 2 では，L1, L2, ... と学習データを増やすごとにユークリッド距離が 0 に近付き，かつ，相関係数が 1 に近付いている．これは，学習データの追加により，全データを学習した

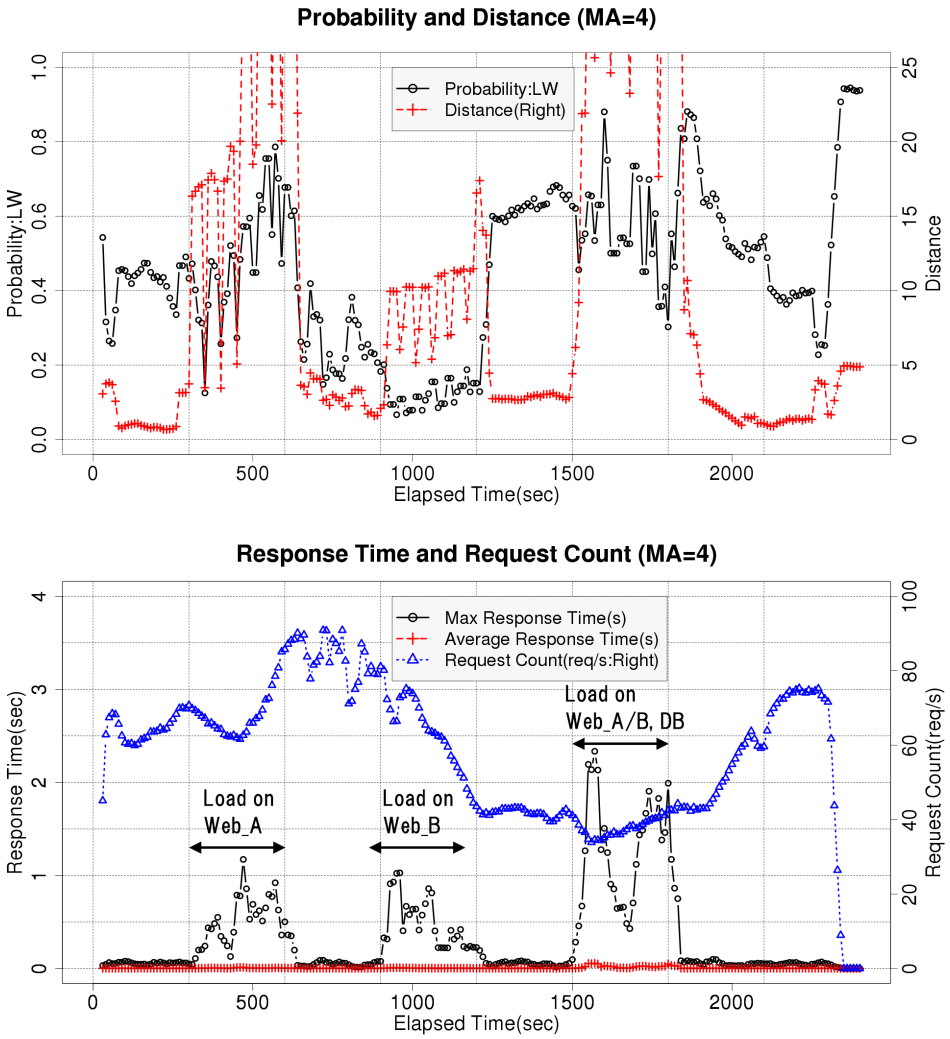

図 2.5: BN および CL での診断結果の例

場合と同等の診断結果を出せる学習データの集合に徐々に近づいている事を意味する．

Base の診断結果を可視化したグラフを図 2.6 に示す．第 1 区画（冒頭の 5 分）で CL の計算結果（三角点，右軸）の平均が，あらかじめ設定した閾値 6 を越えている（平均 20.49）ため，当該区画を学習データに組み入れる．組み入れた後の Heat である L1 の診断結果を図 2.7 に示す．L1 では第 2 区画（5 から 10 分）で，CL の計算結果が 6 を越えている（平均 21.97）ため，当該区画を学習データに組み入れる．同様に L2, L3 と処理を進めた結果である L4 の診断結果（閾値越えなし）を図 2.8 に示す．

この実験では第 1,2,4,6 の 4 区画を初期学習データに追加した L4 が最終形で，Heat が L1 から L4 まで進む間に，ユークリッド距離および相関係数が共に改善された．最終的に L4 で，Whole とのユークリッド距離が 0.16，相関係数が 0.98 となり，全 8 区画中 4 区画の観

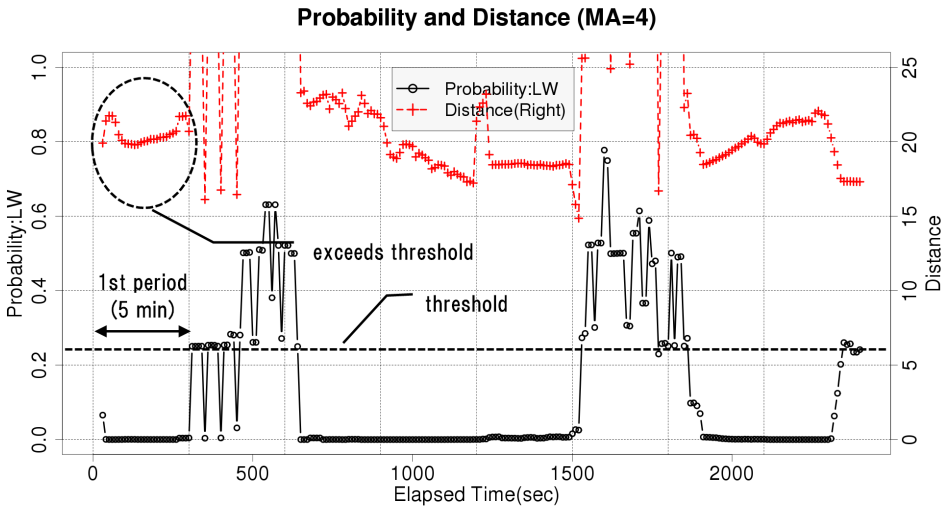

図 2.6: Base の診断結果

測データで，全区画を学習した場合（図 2.9）と同等の診断結果が得られた．この実験では，さらに，提案手法による学習データ選定の妥当性を検証するために，8 区画中，L4 で選択しなかった 4 区画を学習した場合 (R4) や，提案手法による選定とは無関係に，L4 と同じ 4 区画分を，単純に 8 区画中の前半 (F4)，中盤 (C4)，後半 (S4) の 4 区画，および，先頭と末尾の 2 区画ずつを選択した場合 (P4) の診断結果との比較を行った（表 2.3）．

表 2.3: 他の学習結果との比較

Heat ID	Training Data	Avg Dist.	Euclid.	Correlation
L4	L3+25-30	8.79	**0.16**	**0.98**
R4	Reverse of L4	–	0.38	0.96
F4	Base+0-20	–	0.42	0.78
S4	Base+20-40	–	1.21	0.85
C4	Base+10-30	–	1.50	0.60
P4	Base+0-10,30-40	–	1.49	0.43

その結果，いずれも L4 の結果には及ばず，L4 が Whole に最も近い診断結果を得られる事が分かった．同様の実験（L1 から最大 L4 まで）を 10 回行った結果を表 2.4 と表 2.5 に示す．Exp. ID は実験 ID，Heat ID は当該実験で提案手法を適用した場合の最終 Heat ID または比較対象の Heat ID，Euclid. は当該 Heat と全データを学習させた結果とのユークリッド距離，Correlation はその相関係数を示す．各実験の 1 行目は最終 Heat の結果を，2 行目に，前項で説明したような他の学習データを選定した場合で最良の結果を出した Heat の結果を比較対象として記載した．Heat ID が R で始まる Heat は，提案手法で選択しなかった

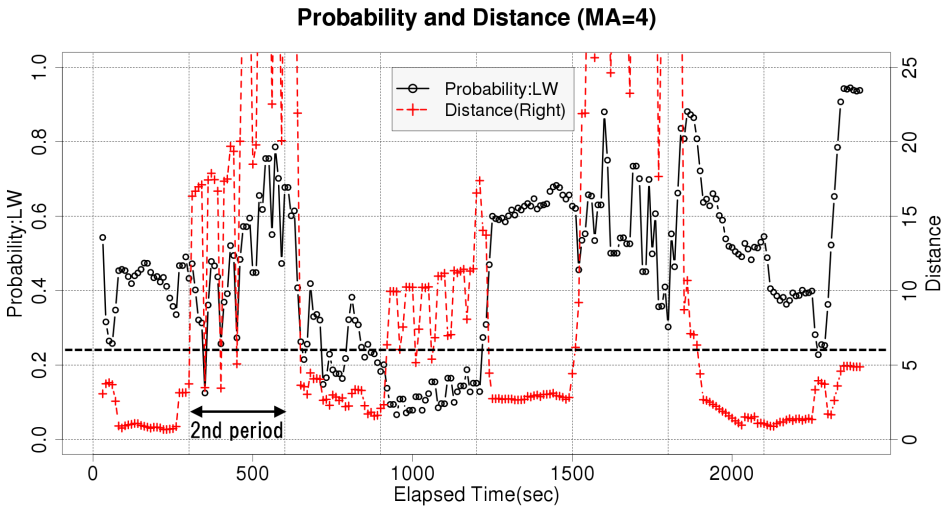

図 2.7: L1 の診断結果: 第 2 区画で CL(△点)がしきい値を超える

区画の中から，R に続く数字分の区画をランダムに選択して学習データに追加した場合の結果を意味する．また，Eval. には提案手法が期待通りの選定結果を出せた場合に O，そうでない場合に X を記載した．ただし，ユークリッド距離および相関係数の一方でのみ優位性が現れた，実験 Exp_Dd に関しては，相関係数のわずかな差よりもユークリッド距離の顕著な差を重要視して O とした．実験の結果，提案手法は 10 種中 7 種の実験で期待する結果を出す事ができた．また，初期学習データを変更した場合と，診断データを変更した場合の実験結果に顕著な違いは見られなかった．以上から，提案手法は大量の観測データの中から学習データとして利用すべき部分データを自動的に選択可能である事，また，BN の学習時間は学習データ量に比例するため [22]，ひいては，異常診断を行う BN の学習時間の短縮に貢献できる事が分かった．

2.3.5 結果の考察

今回の実験では再学習の発生頻度が上がりすぎないようにするために 5 分の固定長で各区画を区切り，この区切りに合わせて監視対象システムへ負荷をかけている．しかし，結果を詳細に分析したところ，観測データが表す負荷はこの 5 分区画の境界を越えて次の区画に影響を及ぼしている事が分かった．これにより，提案手法で選定した区画の次の区画を学習すると選定区画の一部を学習したのと同じ効果が得られる可能性があると考えられる．この状況を勘案すると，負荷の影響の始まりと終わりを正しく認識して学習データに組み入れ，かつ，再学習の発生頻度を一定以下に制御することが，今後の改良点としてあげられる．

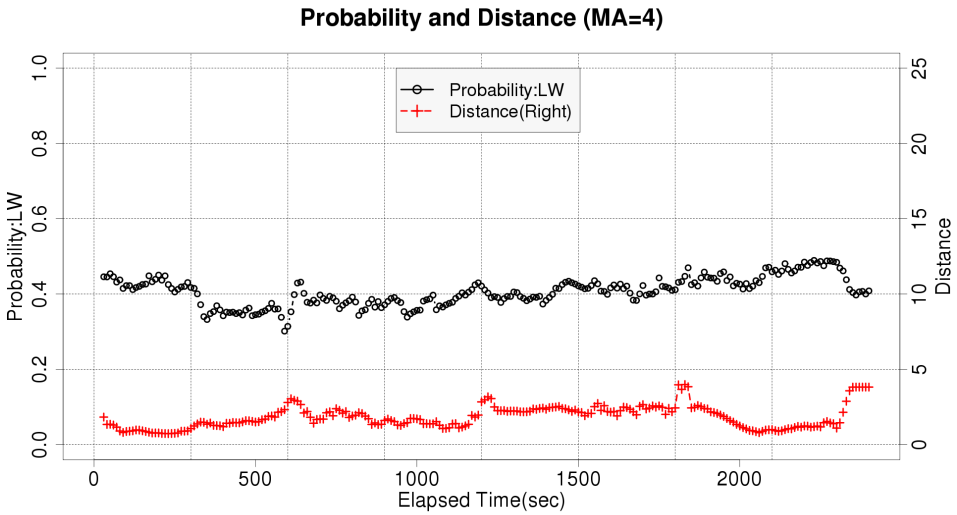

図 2.8: L4 の診断結果: CL のしきい値超えなし

2.4 関連研究

Ira Cohen らは [5] にて，本論文での提案手法と同類の TAN(Tree-Augmented Bayesian Network) を活用して，様々なメトリクス観測結果とシステムの状態を関連付ける技術を発表した．Steve Zhang らは [4] にて，本手法を拡張して，直近の区画の観測データから生成した新モデルが，当該区画の診断において，現行モデル集合内のどのモデルよりも良い診断結果を出した場合，この新モデルをモデル集合に追加する診断方法を提案した．この手法では複数のモデルを結合するのに比べ，本論文の手法では学習データを結合する点が異なる．学習データの結合処理がデータファイルの単純な結合で実現できるのに比べ，モデルの結合は時間のかかる複雑な処理となる．Satoshi Iwata らは [9] にて，性能劣化の根本原因を推定する方法を発表した．システムを構成する各サーバでの処理時間の平均／中央／最大／最小をメトリクスとしてクラスタリングを行い，原因の判明している既知のインシデントと同じクラスタに属する新規インシデントは同じ原因で発生していると推定する．事前に人手による分析を終えた学習データ（教師データ）を必要とする事が本論文の手法と異なる点である．Thanh H. D. Nguyen らは [10] にて，システムの性能劣化原因の自動特定技術を発表した．この技術はまず，事前の性能テストで性能計測データと性能劣化原因のペアを複数収集し，これを教師データとする．次に，発生中の事象が教師データのどれと類似しているかを機械学習アルゴリズムにて同定することで事象の原因を推定する．学習データ量削減のために Control Charts を利用する．17 種の機械学習アルゴリズムにて，それぞれの 6 種の性能劣化原因の究明率を比較した結果，1 原因あたり 4 つの学習データを準備することで，74-80%の精度で原因究明で

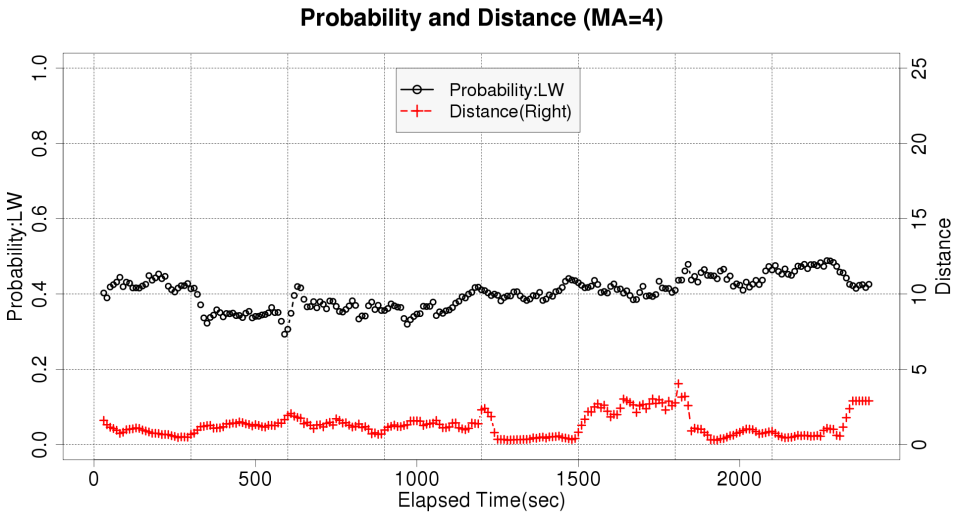

図 2.9: Whole の診断結果

きる事を確認した．少量ではあるが，事前に人手による分析を終えた学習データ（教師データ）を必要とする事が本論文の手法と異なる点である．

2.5 むすび

Web サービスシステムの CPU／メモリ／ネットワーク等の利用状況の観測値を元に，機械学習を活用してシステムの状態，特に性能異常の検知に寄与する，学習データの自動選定手法を考案した．オンライン販売システムを使い，あらかじめ用意した初期学習データに加えて，提案手法で自動選定した学習データを追加した場合の診断結果と，全データによる診断結果の類似性を判定する評価実験を行った結果，10 種中 7 種で，より少量のデータで全データ学習と同等の学習効果が得られることを確認した．今後の課題として，(1)CL での既学習／未学習を判定する際の閾値の決定方法，および，(2)未学習データを学習データに組み入れる際のデータ区画長の決定方法があると考える．本実験では，CL の判定閾値として，既学習データに対しては十分に大きいと人間が判定した 6 を，また，データ区画の分割方法として固定長 5 分を，それぞれ採用したが，後者に関してはその問題点を確認した．将来は，前者に関しては自動化を，後者に関しては CL の出力結果が急激に大きくなるエッジを捉えてから，急激に小さくなるまでの，可変長のデータ区画を採用し，かつ，再学習の頻度が上がりすぎないように制御するのが適切と考える．

表 2.4: 初期学習データを変更した場合の実験結果

Eval.	Exp. ID	Heat ID	Euclid.	Correlation
X	Exp_Sa	L4	0.66	0.36
		R4	0.60	0.88
O	Exp_Sb	L2	2.57	0.95
		R2	3.19	0.95
O	Exp_Sc	L4	0.16	0.98
		R4	0.38	0.96
X	Exp_Sd	L4	1.27	-0.07
		R4	0.73	0.28
O	Exp_Se	L4	1.26	0.37
		R4	1.31	0.30

表 2.5: 診断データを変更した場合の実験結果

Eval.	Exp. ID	Heat ID	Euclid.	Correlation
O	Exp_Da	L2	1.27	0.90
		R2	1.45	0.90
X	Exp_Db	L2	1.34	0.90
		R2	0.97	0.95
O	Exp_Dc	L3	0.39	0.97
		R3	3.50	0.88
O	Exp_Dd	L3	0.57	0.80
		R3	2.42	0.85
O	Exp_De	L3	0.97	0.95
		R3	3.37	0.72

第3章　Grid技術のWebサービスシステムへの適用

3.1　導入

Globus toolkit [26, 27] のようなGridコンピューティングミドルウェアと，Open Grid Services Architecture (OGSA) [28] のようなGrid標準は，大規模分散システムの実装の容易化を主目的とし，サービス指向アーキテクチャ(Service-Oriented Architecture:SOA)，アプリケーションを構成するコンポーネントのライフサイクル管理，リソースの利用効率やアプリケーションのスケーラビリティの改善に役立つリソース管理，といった機能を提供する．これまで，Grid技術を効果的に活用できたアプリケーションのほとんどは大規模な科学計算であったが，組織横断の分散企業アプリケーションにも適用可能であると考える．これが，Global Grid Forum (GGF) とW3CやOASIS等の標準化団体によるGridサービスとWebサービスの標準を有機的に融合することの動機となった．

本章では，Grid技術を企業アプリケーションに適用するという課題を扱う．特に，以下の2つの疑問への回答を試みる．

- 既存のGridミドルウェアGlobus toolkitがインターネットバンキングやeコマース等の企業アプリケーションに適用可能かどうか．これをアプリケーションの**Grid化**と呼ぶ．
- 機能性，性能，管理容易性等の観点で，Grid技術がどんな付加価値をもたらすか．

本章ではこれらの課題に取り組むためにケーススタディを実施した．バックエンドのデータベースサーバと，それにアクセスするアプリケーションサーバ，フロントエンドWebサーバからなる，典型的な3層Web構成のeコマースアプリケーション [29] を活用した．

もちろん，これは企業アプリケーションの1形態でしかないが，旧来の科学計算アプリケーションとは異なり，1リクエストあたりの計算量は小さいが，同時に大量のリクエストを許容し，かつ，応答時間を短く保つ必要があるという特性を持つアプリケーションの代表としてはちょうどよい．

Grid技術が成熟するにつれ，このようなアプリケーションは，設計者がGridミドルウェアを使って実装する典型的なアプリケーションとなると考えられる．

既発表の技術とは異なり，本章では，既存の3層J2EE Webアプリケーションをいかに

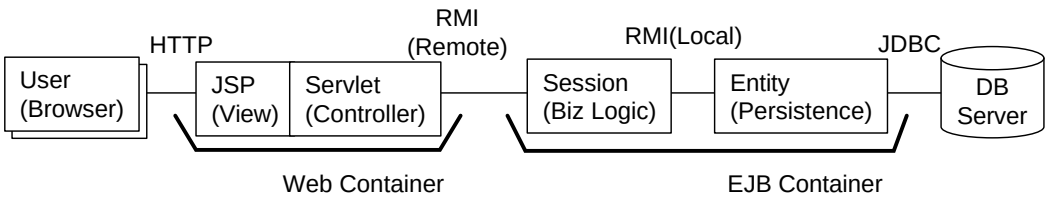

図 3.1: 典型的な Web3 階層アプリケーション

Grid 化するかを詳細に述べ，その変換の実行可能性と，Grid 技術の適合性を評価する性能指標を提示する．

本章は，3.2 節で Web アプリケーションを Grid 化する方法について述べる．Web アプリケーションの構造，Grid 化の 2 つの方法:*delegation* と *porting*，簡単なオンラインバンキングアプリケーションを使って，アプリケーションを変換する方法，オリジナル版と Grid 化版との性能比較を含む．3.3 節では，変動するリクエストに対応するために，Grid 技術の動的リソース割り当て機能を活用するスケーラブルなアーキテクチャと，性能評価結果を示す．最後に，3.4 節で結論と今後の研究について述べる．

3.2 アプリケーションの Grid 化

3.2.1 概要

ここでは Web ブラウザから HTTP プロトコルでアクセスする Web アプリケーションを想定する．このようなアプリケーションを開発する方法はいくつかあるが，ここでは，図 3.1 に示すような，J2EE アプリケーションモデルと Enterprise Java Beans (EJB)[29] を活用した 3 層 Web アプリケーションに注目する．ここでいう 3 層とは，

- ***Web* サーバ**. HTTP でアクセスされるフロントエンドで，JavaServer Pages (JSP) と Servlet から構成される．コードは Web コンテナ上で実行される．
- **アプリケーションサーバ**. 業務ロジック記述とアプリケーションの状態を保持するために Session Beans と，Entity Beans を活用する．Session Beans はクライアントとアプリケーションサーバ間のやり取りを表現し，クライアントのためのタスクを実行する．Entity Beans は永続化オブジェクトを表現する．コードは EJB コンテナ上で実行される．
- **データベースサーバ**. アプリケーションサーバのバックエンドで Entity Beans の永続化に利用され，JDBC API でアクセスされる．

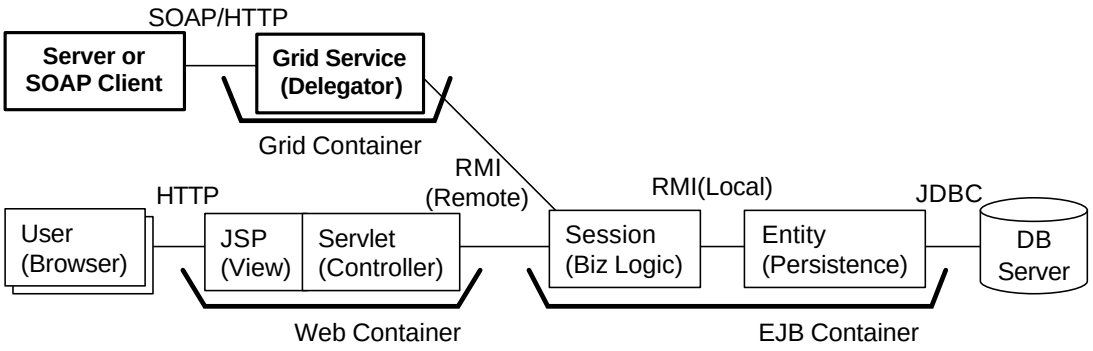

図 3.2: Delegation モデル

本実験では，全データサイズを小さくし，データベースへのアクセス帯域が不足していない状況を維持し，システム全体の計算能力に注目できるようにした．

このようなアプリケーションがある場合の最初の課題は，Grid 化の意味であろう．例えば，Grid 機能を構成するどのコンポーネントをどこにどのようにして適用するのか．

本研究では 2 種類の解と，それぞれのアーキテクチャモデルを考案し，それぞれの実装を行った．一方は，Web アプリケーションそのものを OGSA 準拠の Grid サービスインターフェイスを持つ Grid サービスに変換する方法である．他方は，Web アプリケーションはそのままで，その実行基盤に Grid 技術を活用することである．本論文では，前者を Porting モデル，後者を Delegation モデルと呼ぶ．

3.2.2 Delegation 対 Porting

Delegation モデルでは，Grid サービス SOAP リクエストを Java RMI リクエストに変換する Delegator が必要となる（図 3.2 参照）．Delegation モデルでは，既存の EJB アプリケーションのコードを変更することなく，Grid サービスインターフェイスを提供することができる．しかし，業務ロジックは EJB コンテナ上で，実行／管理されるため，最終システムは EJB コンテナに依存する．結果として，アプリケーションの一部は EJB アプリケーションのままで，ここに Grid プラットフォームの利点を生かすことはできない．この方法は，フロントエンドのアクセスプロトコルを HTTP から SOAP にするため，Web アプリケーションを SOAP ベースの Web サービスに変換するのにしばしば利用される．

一方，Porting モデルでは，アプリケーションは Web アプリケーションのまま，その実装に Globus Toolkit を利用する．すなわち，Porting モデルでは，EJB コンテナを ***Grid* コンテナ** に置き換えることで，スケーラビリティ／管理容易性／動的リソース割当等 [28]Toolkit が提供する全機能を活用可能となる．しかし，この恩恵を受けるには，Java Beans で記述された Web アプリケーションのコードを書き換える必要が生じる．

本論文では，より多くのGrid技術の恩恵を受けることができ，かつ，既存のWebアプリケーションインターフェイスを維持できる，Portingモデルに注力する．最初に，Webアプリケーションを Gloubs 3.2 上に Porting する手順を示し，次に，移植前の EJB 版と，移植後の Grid 化版，および，その中間版の性能面への影響を評価する．

3.2.3 Porting 手順

実験のために，ログイン／ログアウト，残高照会，振込，振込履歴照会の機能を持つインターネットバンキングアプリケーションを開発した．これらの5つの機能はそれぞれ異なる特性を持つ．

- ログイン. 1つの単純なデータベース内テーブル（以下単にテーブル）の参照を発生させる少量のデータ転送を発生させる
- 残高照会. 2つの単純なテーブルの参照と，リクエストよりもわずかに大きい応答メッセージを発生させる
- 振込履歴照会. 複雑なデータベース問合せとリクエストよりもかなり大きな応答メッセージを発生させる
- 振込. テーブルの Lock を必要とする複数の SQL トランザクションと，複数のテーブルの更新を発生させる
- ログアウト. Web コンテナ内で処理され，EJB コンテナには到達しない

この Web アプリケーションは 1 つの EJB Session BeanTeller と，4 つの EJB Entity BeanAccount, Customer, Transaction, and AcctCustRel を使って実装した．

EJB で実装された Web アプリケーションは以下の手順で Porting した．はじめに，Entity Beans を標準 Java クラスに変換し，次に，Session Beans から OGSA 準拠の Grid サービスへの変換と Java RMI から SOAP への変換を行った．2つの変換それぞれで，実行可能なアプリケーションができるため，元のアプリケーションと合わせて，3つのバージョンが出来上がった．元のバージョン（以下 ***EJB* 版**）は Session Beans と Entity Beans からなる EJB 技術のみ活用する．中間的なバージョン（以下 ***Hibernate* 版**）は，Session Beans を活用するものの，Entity Beans は標準 Java クラスで置換し，Object-Relation Mapping Framework[30] の Hibernate を使って永続化を実現する．Porting が完了したバージョン（以下 ***Grid* 版**）は，Grid サービスと Hibernate を活用する．図 3.3 に，最後の 2 つのバージョンを示す.

Hibernate 版では，Entity Beans を標準 Java クラスで置換し，永続化のために Hibernate を活用する．Hibernate は Java Object と関係データモデル間を SQL ベースのスキーマでマッピングするためのフレームワークである．Hibernate は Java クラスとテーブル間の相互変換を行う. コードの変更が必要になったのは，EJB の create メソッドを通常のコンストラクタに，

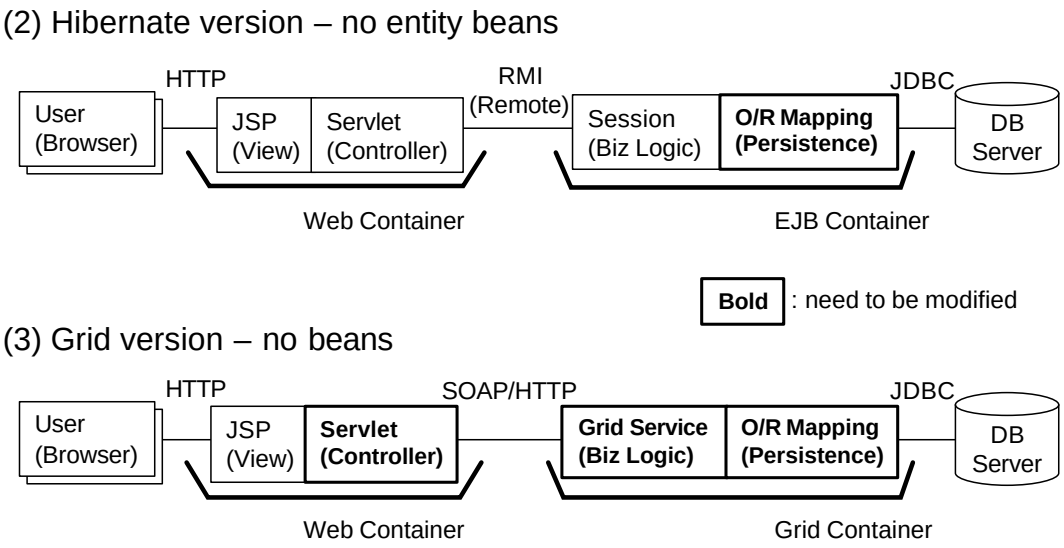

図 3.3: Hibernate 版と Grid 版

EJB の finder メソッドを通常メソッドに変換する事だけだった．業務メソッドと setter/getter メソッドはそのまま使うことができた．その他の変更は，EJB 版のデプロイメントディスクリプタを生成するための XDoclet 向けの指示文に関連する．最後の 2 つのバージョンでは，これを Hibernate 環境に合わせるために修正を行った．

Grid 版では，Web コンテナと EJB コンテナ間の Java RMI ベースの通信を Web コンテナと Grid コンテナ間の SOAP/HTTP ベースの通信で置換した．これには，Entity Beans に相当する Java クラスとその XML 表現間のマッピングが必要で，SOAP 通信コンポーネント Apache Axis 内の静的 Stub コード生成ツール `wsdl2java` を活用した．

Session Beans Teller を Grid サービス TellerService に変換するのは少し複雑であった．最初に，GWSDL (Grid Web Services Definition Language) によるインターフェイス定義が必要で，Session Beans のメソッドの集合を GWSDL の XML ベースの表現形式で記述した．この作業は `java2wsdl` のような WSDL 生成ツールに Java クラスを入力として与えた結果をベースとして変更することで単純化できる．最後に，Globus Toolkit 内の `gwsdl2java` ツールを使って，GWSDL からシリアライゼーション/逆シリアライゼーションメソッドを生成した．[1]

3.2.4 性能比較

本アプローチは，Web アプリケーションは Globus による Grid 基盤上に Porting 可能であることを示しているが，Grid 化 Web アプリケーションの性能は課題となりうる．一般的

[1] 本生成ツールには複雑なデータ型を扱う際に問題が生じたため，Stub コードを手で修正する必要があった．

に，SOAPのようなXMLベースのプロトコルは，Java RMIのようなバイナリベースのプロトコルに比べ，性能面で劣る．これはXMLベースのメッセージサイズが同じ情報量のバイナリベースメッセージよりもかなり大きくなり，その処理オーバーヘッドが顕在化するためである．このオーバーヘッドが重大な差となったり，他の性能問題が持ち上がる場合，Grid基盤上へのWebアプリケーションのPortingはあまり採用されなくなるだろう．

この問題を調査するために，本論文ではサンプルアプリケーションの3つのバージョンを評価した．このアプリケーションを実行するために，EJBコンテナとしてJBoss3.2.3[31]を，Webコンテナとして JBossに組み込まれたTomcat4.1を，データベースマネージャとしてPostgreSQL7.3.4を，また，Gridコンテナとして Globus Toolkit3.2に組み込まれたApache Axisを活用した．

実験では，Webコンテナ, Gridコンテナ, データベースマネージャはそれぞれ個別のサーバに，1サーバあたり1コンテナをデプロイした．全てのサーバが1.8GHz Pentium4プロセッサ1個と2GBのメモリを搭載し，RedHat Linux 9上にJava実行環境J2SE1.4.2が稼働するように設定した．

複数ユーザの並列動作を模擬するアプリケーションへのリクエストを生成するために，JMeter2.0トラフィックジェネレータを活用した．

トラフィックジェネレータを2.4GHz Xeonプロセッサ2個と1GBメモリを搭載し，RedHat Linux 9が稼働するサーバ2台にデプロイした．個々のトラフィックジェネレータは，6個のスレッドでリクエストを生成する．個々のスレッドは，ログイン／口座残高照会／振込／口座履歴照会／ログアウトや，より単純に，ログイン／残高照会／ログアウト，といった典型的な個人の銀行取引の振る舞いを模擬する一連の動作を行う．各動作の間に待ち時間を挿入しなかったので，レスポンスを受信後すぐに次のリクエストを送る．また，不正なユーザIDとパスワードの組のような，不正なリクエストは挿入しなかった．

実験では，トラフィックジェネレータは25秒間リクエストを送ってから停止する．この25秒間に最大1500リクエストが送信／処理された．各版のアプリケーションに対して5回の計測を行い，リクエスト種別ごとに平均応答時間を計算した．応答時間は送信開始当初が常に短くなる傾向にあったため，最初の3秒分を捨てて，その後の20秒分のみ使った．

図3.4に，3つの版の，リクエスト種別ごとの平均応答時間を示す．この性能比較により，事前の予想に反して，Grid版の性能はEJB版と大差なく，振込リクエストに対しては，EJB版よりもHibernate版，Grid版の方が良い結果が出た．

さらに詳細に見るために，Webコンテナ内のServlet，EJBコンテナ内のSession Beans，Gridコンテナ内のGridサービスでの応答時間を計測した．以下では，次の表記を使う．

- T_t. トラフィックジェネレータで計測したWebアプリケーション全体の応答時間

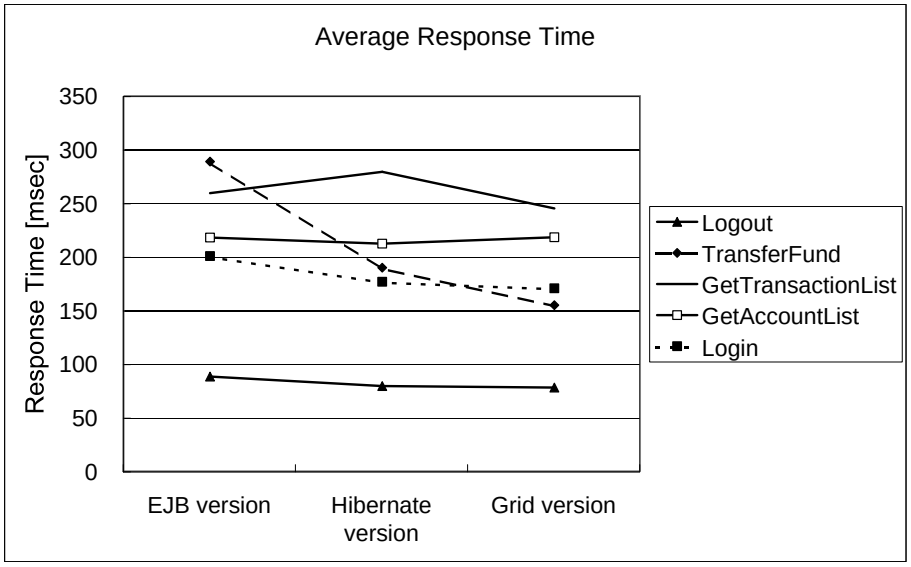

図 3.4: 3 つの版のリクエスト種別の平均応答時間

表 3.1: RMI に対する SOAP のオーバーヘッド

	T_m in EJB (ms)	T_m in Grid (ms)	Overhead (ms)	Ratio(%)
Login	101.7	109.0	7.3	3.7
GetAccountList	112.6	160.9	48.3	22.2
GetTransactionList	96.7	158.7	62.0	23.9

- T_w. Web コンテナ内の Servlet で計測した応答時間
- T_a. EJB コンテナ内の Session Beans または Grid コンテナ内の Grid サービスで計測した応答時間

T_a は Session Beans と Entity Beans または Grid サービスの実行時間および，データベースへのアクセス時間を含む．T_w は T_a 全体および，Web コンテナでの Servlet の実行時間，Web コンテナと EJB/Grid コンテナ間の通信時間を含む．この通信時間は，引数のバイト列化および，HTTP や SOAP 等のシステム層のメッセージ処理時間を含む．

上で示したように，おそらく最も関連性の高い計測値は，Java RMI と SOAP 間の差だと考えられる．指標 T_m を $T_w - T_a$ と定義すると，T_m は，Servlet 内での経過時間および，Servlet と Session Beans または Grid サービス間の通信時間を表すので，T_m は 2 つのプロトコル間の性能差見積りに使うことができる．

表 3.1 に，3 つのリクエストに対するこの指標を示す．計測の結果，SOAP プロトコルの RMI に対するオーバーヘッドは 20%に達し，転送データサイズに依存する事が分かっ

た．いずれも他の報告に準ずる結果であった [32].

以上に述べたSOAPプロトコルのオーバーヘッドがあるにも関わらず，本実験で開発したGrid版はEJBと近いか，場合によってはそれ以上の処理性能を示した．これはHibernateによるJavaオブジェクトの効率的な永続化実装によるものだと推測する．EJBコンテナはEntity Beansの永続化を行うためにContainer Managed Persistence (CMP)と呼ぶ，冗長性と実行時制限を必要とするモジュールを使う．Hibernateを活用すると，データレコードをテーブルに書き込むタイミングと，テーブルのロックが実際に必要となるタイミングをHibernateに指定するコードを追加することで，これらを排除することができる．これにより，テーブルロックの時間を短くすることができる．

本実験のアプリケーションでは，追加コードは単純なものであったが，実際の大規模アプリケーションでは，より複雑化する可能性がある．これは，開発容易性と，追加コードによって得られる性能／管理容易性間のトレードオフである．

Hibernate版とGrid版で，データベースロックを管理する方法をそれぞれJTA (Java Transaction API)からJDBC Transactionに変更してみたところ，Grid版で若干の改善が見られた．

3.3 スケーラブルアーキテクチャ

Grid技術を適用するメリットは，その動的リソース割当機能を使って，処理性能をシームレスに変更できる**スケーラブルアーキテクチャ**を採用できる点にある．特に，Webアプリケーションでは，スケーラブルであることはリクエスト量の変化に対応するために重要となる．

負荷の変動に対処するために，Grid基盤の動的リソース割当機能を活用するように設計した3層Webアプリケーションのアーキテクチャを示す．追加リソースはWeb層，アプリケーション層，データベース層のいずれにも有効であるが，既存のデータベース技術がデータベース層に対して十分な処理性能を供給できるという想定で，Webコンテナと Gridコンテナへの適用に注力した．

本章で提案するスケーラブルアーキテクチャを図3.5に示す．本アーキテクチャでは，全てのWebサーバとGridサーバ上で，それぞれひとつのWebコンテナとひとつのGridコンテナを稼働させ，Gridコンテナ上で，アプリケーションロジックを実装した，ひとつ以上のGridサービスインスタンス（以下単にGridインスタンス）を稼働させる．

*Grid Manager*はシステムの負荷と性能情報を収集し，動的にGridインスタンスを生成／破棄し，アクティブサーバリストへのGridサーバの登録／削除を行う．本アーキテクチャでは，*Web Load Balancer* (以下WLB)と*Grid Load Balancer*(以下GLB)の2つのロードバランサを利用する．WLBはWebアプリケーションで標準的に使われるが，GLBはGridサービスと同様の機能を持つ．GLBはWebコンテナからのリクエストを適切なGridインス

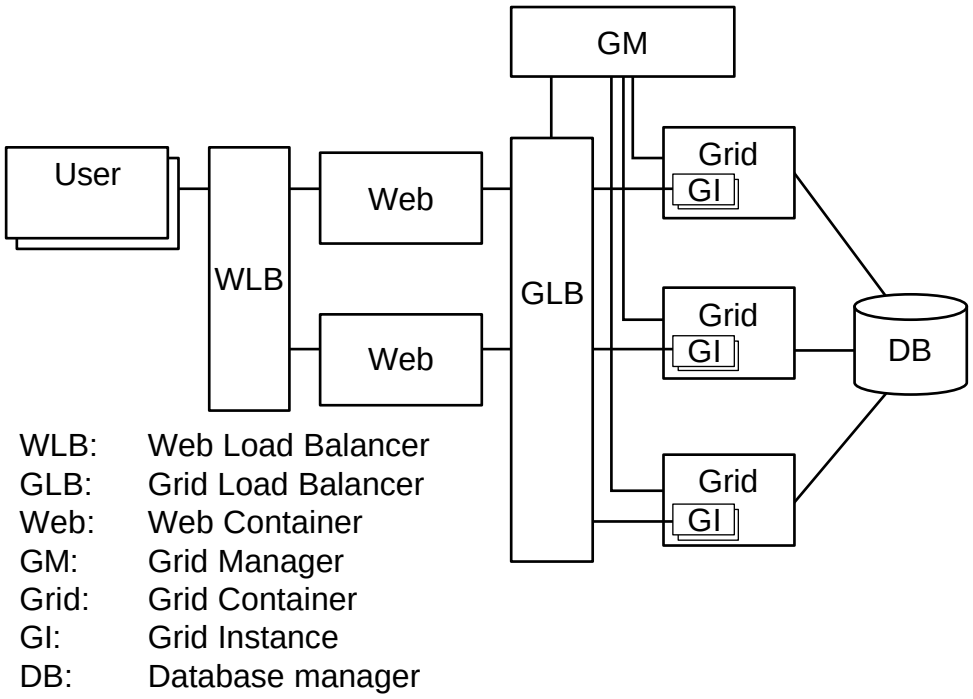

図 3.5: Grid 化 Web アプリケーションのスケーラブルアーキテクチャ

タンスに振り分ける．振り分け先インスタンスは，当該リクエストを処理し，レスポンスを GLB 経由で適切な Web コンテナに返す．このアーキテクチャは，Grid サーバと，Grid インスタンスをシステム内の他のコンポーネントとは独立して管理することを可能にする．

WLB は入力リクエストを Sticky-Session 方式で転送するため，当該ユーザがログアウトするまで当該セッションを維持する．これを実現するために，セッション ID と対応する Web サーバ ID を記録する Cookie を活用した．リクエストを受け取ると，WLB が 2 つの ID を Cookie から抽出し，当該リクエストを担当するサーバに転送する．リクエストに ID が含まれない場合，または，参照先のサーバが存在しない場合，WLB は当該リクエストを処理するサーバを Round Robin 方式で選択する．本実験のアプリケーションでは，Grid インスタンスでの処理は Session に依存しないため，GLB は，単純に Round Robin 方式で，リクエストを Grid インスタンスに転送する．

本章で提案するアーキテクチャのスケーラビリティを検証するために，Web サーバ，Grid サーバ，Grid サーバ上の Grid インスタンス数が異なる構成で，応答時間と経過時間を計測した．本実験のゴールは，第一に，性能の観点で，Web アプリケーションに Grid 技術が適用可能であることの確認，次に，Grid 基盤が提供する機能の優位性を引き出すことができるアーキテクチャの決定，である．

検証のために，前節で述べたGrid版インターネットバンキングアプリケーションをすべてのGridサーバに同じ設定でデプロイした．各Gridサーバ内のコンテナが起動すると，必要な数のGridインスタンスが作成され，Gridマネージャに登録される．性能計測の方法は，前節で，3種類のバージョンのアプリケーションを比較する際に使ったものと同じである．唯一の違いは，送信されるリクエストの数だけである．特に，Gridサーバの数が全体システム性能に大きく影響するため，同じ時間の間に約2倍のリクエストが送信／処理された．

Webサーバと Gridサーバの数，1Gridサーバあたりの Gridインスタンスの数が異なる構成で実験を行った．個々の構成を区別するために以下の表記を使った．

- Wα-Gβ x γ: α台の Webサーバ, β台の Gridサーバ, 1サーバあたりγ個の Gridインスタンス，の構成を表す

表 3.2 に，構成毎のリクエスト種別の応答時間を示す．実験は，前節と同じ実験環境 (Web/Grid/Database サーバには 1.8GHz マシンを，トラフィック生成には 2.4GHz マシンを，100MbpsEthernet で接続) で行った．実験中，各マシンの負荷もネットワーク負荷も軽い状態であった．Javaガベージコレクションと他の制御不可能な影響を軽減するために，上位と下位5%に相当する計測結果は除去した．

表 3.2: リクエスト種別応答時間 (ミリ秒)

	W1-G1x1	W2-G1x1	W1-G1x2	W2-G2x1	W2-G3x1
Login	77.0	57.7	61.7	37.7	46.8
Logout	23.4	12.2	19.6	13.2	16.5
GetAccountList	88.3	73.2	72.8	46.5	56.2
GetTransactionList	115.9	140.7	108.1	81.1	117.4
TransferFund	133.4	143.8	116.3	80.8	101.6

結果は，Webサーバと Gridサーバ数の増加がシステムの応答時間改善に効果がある事を示しているが，Gridサーバの数が3以上になると，応答時間が長くなってしまう．これは，データベースサーバでの競合が増えることが原因と考えている．

Gridサーバ内の Gridインスタンス数だけが異なる2つの構成の比較から，この環境における，1Gridサーバあたりの Gridインスタンス数の上限を計り知ることができる．いくつかのリクエスト種では，Webサーバの数が，Gridサーバの数よりも多いときに，応答時間が長くなってしまう．これは，2台の Webサーバが，1台の Gridサーバがボトルネックとなるのに十分な数のリクエストを処理できるからであると考えられる．詳細な分析がこれを示している．Webサーバ1台の時の，Gridサーバの GetAccountList, GetTransactionList, and TransferFund に対する応答時間はそれぞれ，21.3 ms, 38.1 ms, and 65.9 ms であったにも関

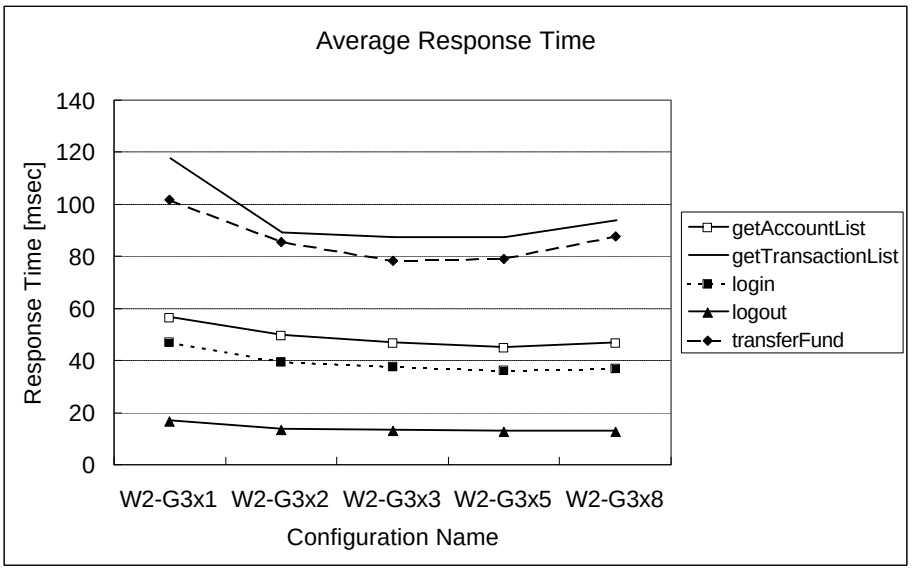

図 3.6: サーバあたりのインスタンス数と応答時間との関連

わらず, Web サーバが 2 台になると，これらが，それぞれ，30.0 ms, 72.6 ms, and 95.3 ms に延びた.

図 3.6 に，1Grid サーバあたりの Grid インスタンス数のみ異なる構成での性能比較を示す. 一般に，インスタンス数が増加すると性能が改善するが，インスタンス数 3 から 5 あたりで性能向上が見られなくなり，インスタンス数が 8 になると性能劣化が起こる. このことから，本アプリケーションにおける，最適インスタンス数は 5 程度であることが分かる. 前回同様，集計時に上位と下位 5%の計測結果を除去した.

Grid サーバの数だけが異なる構成の比較から，サーバ数増減の効果を計ることができる. サーバ数増減の効果を計測するために，稼働中に動的にサーバ数を変更して，応答時間を計測した. 全体で 180 秒の実験で，60 秒経過時にサーバ 1 台を追加し，120 秒経過時にさらに 1 台追加した. 実験開始当初，および，サーバ追加直後の 3 秒分の計測結果を削除し，その後の 40 秒間の計測結果を図 3.7 に示す. 予想通り，Grid サーバの増加は応答性能の改善をもたらし，Grid サーバの現象は逆の効果をもたらした.

3.4 結果と今後の課題

本章では，企業アプリケーションとして重要な位置づけにある，Web アプリケーションに Grid 技術を適用する事の，実現性と適合性の両方を明らかにすることを試みた. はじめに，実現性の検証として，インターネットバンキング Web アプリケーションが，Globus を使って，許容できる程度の工数，かつ，少なくとも本サンプルアプリケーションでは，許容できる程度の性能劣化で，Grid 化できることを示した. 次に，適合性の検証として，

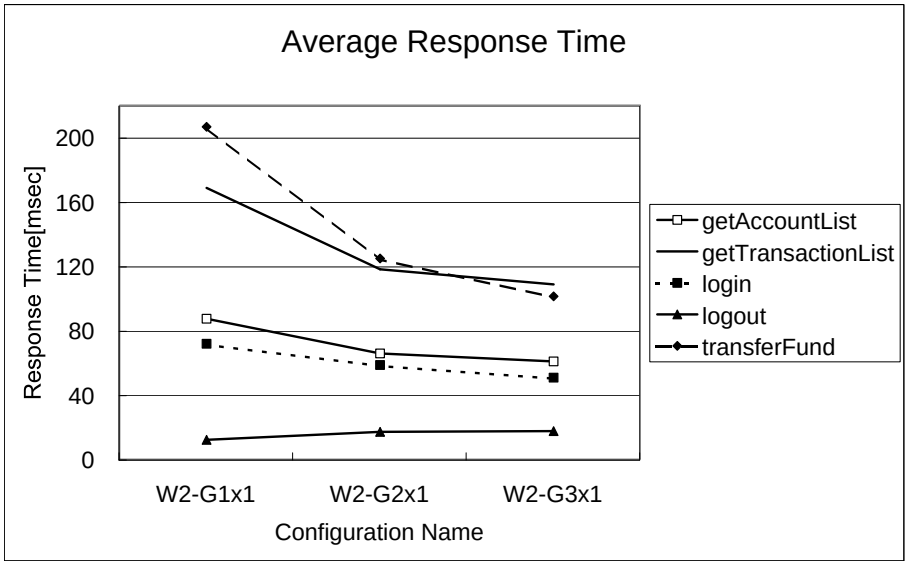

図 3.7: Grid サーバ数と応答時間との関連

Web アプリケーションが Grid の機能を活用する意義がある事を，Grid 基盤の動的リソース割当機能をサポートするスケーラブルアーキテクチャを提案することで示した．基本的な性能検証で，クライアントからのリクエスト量に応じて，Grid サーバの数を増減させる場合も，サーバあたりの Grid インスタンス数を増減させる場合も，システム処理能力がスケールする事を確認した．ここで得られた結果はサンプルアプリケーションによる限定的なもので，より現実的な負荷環境下で，より現実的なアプリケーションにどの程度適用可能かを検証する必要はあるが，Grid ミドルウェアが提供する高度な管理機能は，企業アプリケーションにとって重要な優位性を提供すると信じている．

Grid 技術を典型的な 3 層 Web 企業アプリケーションに適用する類似の研究には，[33] がある．これは，データベースサーバと Web サーバからなり，JSP と Servlet によるプレゼンテーション層と，Java クラスによる業務ロジック層を含む Grid 化 Web アプリケーションのためのアーキテクチャについて記載している．この論文での Grid 化の目的は，業務ロジックの実行を，必要な時に，他の空きサーバにオフロードすることにある．これを実現するために，全てのサーバに監視サービスと利用可能なサーバとその負荷を記録する Grid レジストラサービスを配置するアーキテクチャを採用する．さらに，Grid コントローラ Java クラスが Servlet から業務ロジッククラスへのリクエストを中継し，必要に応じて，リモート業務ロジック Grid サービスに転送する．本章でのアプローチと違い，既存の J2EE ベース Web アプリケーションを Grid 基盤へ移行する事は考慮しないし，Grid 化したアプリケーションの性能測定も行っていない．さらに，Web サーバ層の拡張も考慮していない．

また，関連の研究として，Webアプリケーションでユーザの状態を管理するために一般に利用されるセッション情報をGrid基盤で利用／管理する手法を提案する[34]がある．これは，ひとつのセッション情報にアクセスするイベントの実行を事前に定義したポリシーに従って調整する事で一貫性を維持する方法を提案している．本章での実験では，Gridインスタンスでの処理はセッションに依存しない実装としたため，この技術は不要であったが，より汎用的なWebアプリケーションの実装を考慮すると，本章の手法と補完関係にあると言える．

本章のスケーラブルアーキテクチャと同様のオンデマンドコンピューティング技術は，Gridコンピューティング領域とは異なる領域のWebアプリケーションに適用されることがある．AkamaiやDigital IslandのようなContent Distribution Networks (CDN)は，インターネット上に巧みに配置されCDN企業によって運営されるサーバに，Webサーバの負荷をオフロードするために利用されてきた．典型的には，CDNは画像などの静的コンテンツの配信に活用されてきたが，ストリーミング動画や，動的コンテンツにも利用されるようになってきた．最近，この仕組みが3層eコマースアプリケーションにも拡大適用されてつつある．この技術は，Webサーバの複製，ロードバランシング，管理にフォーカスしており，アプリケーションサーバとデータベース層には対応していないため，本章の技術と補完関係にある．

今後の課題は，より現実的な企業アプリケーションへの適用時に重要となるものが多い．その一つは，ひとつのGrid基盤上での複数のアプリケーションを稼働させることである．このような状況は1企業内でも起こりうる．このようなケースでは，個々のアプリケーションはそれぞれ固有の負荷とリソースの消費パターンを持つため，アプリケーション間のロードバランシングはより高い利用率を達成しうる．

他の課題は，セキュリティである．本章のテストアプリケーションでは，フロンドエンドでのFormベース認証が唯一のセキュリティだったが，一般的にはより厳しい基準が必要である．Webサービスの標準化活動の中で開発が進んでいる技術の適用に期待する．

最後に，Gridマネージャと，GridサーバおよびGridインスタンスの管理方法の改善を提案する．現時点では，Gridサーバと Gridインスタンス数の変更は手動で行うが，これを自動化し，Gridマネージャが計測したシステム状態（サーバの負荷や，応答時間）の変化に応じて自動で行う方法を検討している．

第4章　再帰を含むプログラムのスライス計算方法

4.1　まえがき

本章では, 再帰を含むプログラムの依存関係の静的解析と, その結果に基づいてスライスを求めるための, 対話的システムに組み込みやすいアルゴリズムを提案する. 本研究では, 対話的システムに組み込むことを目標にしているので, このアルゴリズムでは, ソースコードを解析しプログラム依存グラフを作り, その上でスライスを計算するという方法を採用した.

本アルゴリズムが対象とする入力言語では, 再帰的に手続きを定義することができる. 通常, 依存関係の解析には, データフロー方程式が使われるが, ある文に手続き呼び出しがある場合, 陽に現れない変数の定義や参照が発生し, その影響を知ることが容易ではない.

$$
\begin{array}{ll}
 & \vdots \\
1 & g := 0; \\
2 & l := f(5); \\
3 & h := g + 10; \\
 & \vdots
\end{array}
$$

図 4.1: プログラムの一部： g は大域変数で, 関数 f 内で定義可能

例えば, 図 4.1 のようなプログラムを解析する時, 文 1 で定義された変数 g の内容がそのまま文 3 で使われる（文 1 の定義が文 3 に到達するという. 正確には **4.2.** で定義する）かどうかは関数 f の内容に依存する. スライスを計算することを目的とする場合のひかえめな解（無駄なプログラム断片を含むかもしれないが実行に影響を与えるもの全ては必ずスライスとして残る）は, 到達すると考えることであるが, f を注意深く解析すれば, 次のような 3 通りに場合分けすることで, より正確な解を得ることができる.

- f 内で必ず g を定義する時, 1 は 3 には到達せず, 代わりに, f 内の g の定義が 3 へ到達する
- f が g を定義する場合と定義しない場合の両方の可能性がある時, f 内の g の定義と 1 の両方が 3 に到達する

- f が g を定義する可能性が全くない時, 1 が 3 に到達する

また, 再帰的な手続き定義を許すことで問題になるのは, 再帰的に定義された手続きが, ある実行パス中のある文がより前の文へ影響を与える可能性がある, 条件文の条件成立時に実行される文が, 不成立時に実行される文へ影響を与える可能性があるなど, 繰り返し文と似た性質を持っていることや, ある手続きの解析を終える前に, その手続きを呼び出す別の手続きを処理しなければならない場合があることである.

これらに対処するために, 計算に必要な情報をあらかじめ仮定し, その情報をより正しいものへと変化させ, それが収束するまで解析を繰り返すという方法をとった. さらに, 手続きの境界を越えてスライスを計算できるように, 独自の改良を施したプログラム依存グラフを定義した.

図 4.2 にスライスの計算を行なうアルゴリズムの概要を示す. またこのアルゴリズムにしたがってプログラムを解析し, ユーザーの入力に応じてスライスを計算し表示するシステムを試作し, その動作を確認した.

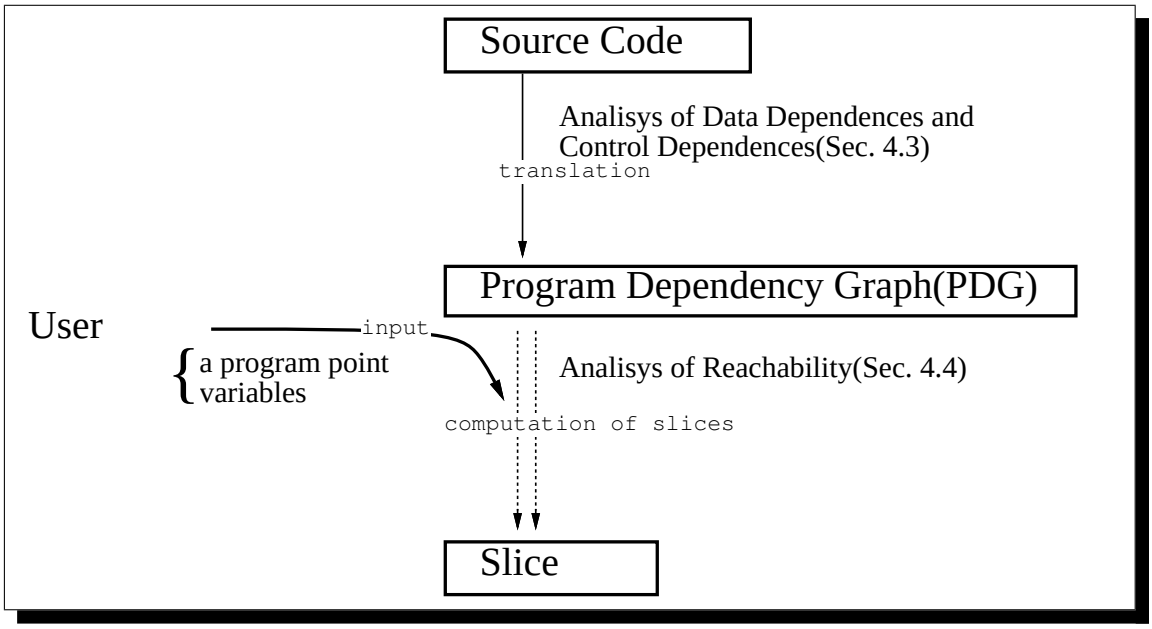

図 4.2: 試作システムの全体像

本論文ではこれ以降, **4.2.** で, 入力言語とプログラム依存グラフに関する定義を, **4.3.** で, 入力プログラムからプログラム依存グラフを作る方法 (図 4.2 の細矢印部分) を, **4.4.** で, プログラム依存グラフ上でスライスを計算する方法 (図 4.2 の破線矢印部分) を, **4.5.** で, アルゴリズムの複雑さについて, それぞれ述べる.

4.2 プログラム依存グラフとスライス

本研究では, Ottenstein, Horwitz と同様に, ソースコードを解析しプログラム依存グラフを作り, その上でスライスを計算するという方法を採用した. この方法は, プログラム依存グラフを作ることにより, 始めの一回の解析を終えた後は, より少ない時間でスライスを計算できる（**4.5.** 参照）もので, 対話的にスライス情報を提供するシステムには適していると考えられる. この節では, 入力言語, プログラム依存グラフおよびその上でのスライスを定義する.

4.2.1 入力言語

ここで紹介する解析アルゴリズムの入力言語として, 以下のような特徴を持つ Pascal 風言語を想定している. この言語は現実の言語に比べ, 簡単化されてはいるが, 本質的な構造をすべて含んでおり, 一般の言語に拡張できる.

その言語には文として条件文 (if), 代入文, 繰り返し文 (while), 入力文 (readln), 出力文 (writeln), 手続き呼びだし文, 複合文 (begin–end) がある. 変数の型としてはスカラ型のみでポインタ型はない. プログラムは, 大域変数宣言, 手続き（および関数）定義, メインプログラムからなり, ブロック構造はない (図 4.7 参照). 手続き内では, 内部で宣言された局所変数と仮引数変数および大域変数のみが参照可能で, 他の手続き内の局所変数は参照できない. 手続きは, 自己再帰的および相互再帰的に定義可能であり, その引数は, 値渡しで扱われる.

4.2.2 制御依存とデータ依存

２つの文の間の関係として制御依存 (Control Dependence, 以下 CD) 関係とデータ依存 (Data Dependence, 以下 DD) 関係がある. 文 s が if 文のような条件判定を含む文であり, かつ, s の条件判定部分 s_1 の実行結果が文 s_2 の実行の有無に直接影響を与える時, s_1 と s_2 の間に CD 関係が発生する. これを CD 関係辺 $(s_1 \dashrightarrow s_2)$ と表す. また, 文 s_1 がある変数 v を定義し, v を参照している文 s_2 にこの定義が **到達する**[1] 時, s_1 と s_2 の間に DD 関係が発生する. これを DD 関係辺 $(s_1 \xrightarrow{v} s_2)$ と表す.

4.2.3 プログラム依存グラフ (PDG)

プログラム依存グラフ (Program Dependence Graph, 以下 PDG) とは, プログラム内の文の間の依存関係を表す有向グラフであり, その節点は, プログラムに含まれる条件判定部分, 代入文, 入出力文, 手続き呼びだし文を表し, その有向辺は２つの節点の間の制御依存および, データ依存関係を表す. ただし, 手続きの境界を越えて, スライスを計算できるよう

[1]「s_1 から s_2 への, v の再定義を含まないような実行パスが存在する」と同義

にするために, 表4.1に挙げるような, いくつかの特殊節点も存在する. 例えば, 大域変数gと, 仮引数pを取る関数fを持つプログラムには図4.3にあるように, exit節点f–$exit$, in

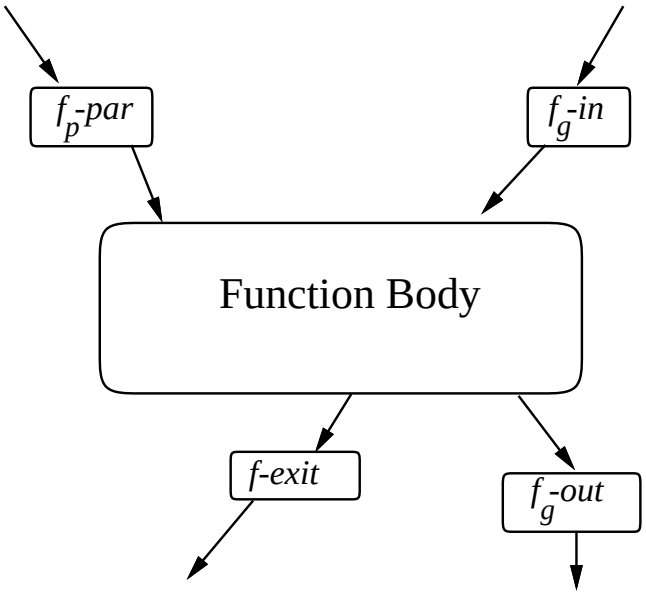

図4.3: 関数fに対するPDGの概略

節点f_g–in, out節点f_g–out, 引数節点f_p–parが存在する.

表4.1: 特殊節点

	特殊節点	表記例
exit節点	関数の戻り値を通して伝わる影響を検出するための節点で, 各関数にひとつずつある	f–$exit$
in節点	手続き外からの大域変数の影響を内部へ伝えるための節点で, 手続きに, 個々の大域変数に対して, ひとつずつある	f_g–in
out節点	手続き内で定義された大域変数の影響をその外へ伝えるための節点で, 手続きに, 個々の大域変数に対して, ひとつずつある	f_g–out
引数節点	手続きの引数を通して伝わる影響を検出するための節点で, その引数それぞれにひとつずつある	f_p–par

4.2.4 スライス

プログラムをPDGで表現し,「文sにおける変数vに関するスライスとは, PDG上でCD関係辺またはDD関係辺を辿って文sの変数vに到達できる節点集合に対応する文の集合である」と定義する.

例えば, 図 4.4 のプログラムの文 writeln(c) に関するスライスは, 図 4.5 の様になり, この部分だけを実行しても指定した文での n の値は同じものになる.

```
program atoi(input,output);
    var c:integer;
        n:integer;
        ch:char;
begin
    n := 0;
    c := 0;
    readln(ch);
    while (ch >= '0') and (ch <= '9') do begin
        n := n * 10 + (ch - '0');
        c := c + 1;
        readln(ch);
    end;
    writeln(n);
    writeln(c);
end.
```

図 4.4: プログラム atoi：入力文字列を整数に変換してその数と桁数を表示する

```
program atoi(input,output);
    var c:integer;
        ch:char;
begin
    c := 0;
    readln(ch);
    while (ch >= '0') and (ch <= '9') do begin
        c := c + 1;
        readln(ch);
    end;
    writeln(c);
end.
```

図 4.5: atoi の文 writeln(c) に関するスライス：桁数を表示する部分が抽出されている

4.3 PDG への変換

ソースコードを PDG に変換するには, プログラム中の各文の間の CD 関係および DD 関係を知る必要がある. CD 関係は入力言語仕様より容易に求めることができるので, ここでは, DD 関係の求め方を詳しく説明する.

4.3.1 諸定義

変数vと節点nとの組dを$\langle v, n\rangle$と表し,変数部をd_{var}で,節点部をd_{vertex}で表す.プログラムのある領域S(連続する0個以上の文の集合)に対する到達定義集合(Reaching Definition Set) $RD_{in}(S)$を,次のように定義する.

$$RD_{in}(S) \equiv \{\langle v, n\rangle \mid \text{節点}\, n\, \text{で変数}\, v\, \text{が定義される} \wedge\ n\, \text{が}\, S\, \text{の先頭へ到達する}\}$$

また,Sを出て次の領域へ到達する定義集合を$RD_{out}(S)$と書く.一般に式4.1ような関係が成り立つ.ただし,$kill(S)$はSで**消滅する**[2]定義の集合を,$gen(S)$はSで新たに発生する定義の集合を意味する.

$$RD_{out}(S) = RD_{in}(S) - kill(S) \cup gen(S) \tag{4.1}$$

プログラムのある領域Sについて,Sを実行した時に(どのパスが実行されても)必ずその値が定義される変数とその定義節点との組の集合を確実定義集合と呼び,$SuDEF(S)$と表す.また,定義される可能性のある変数とその定義節点との組の集合を潜在定義集合と呼び,$PoDEF(S)$と表す.定義より,二つの定義集合間には次のような関係がある.

$$SuDEF(S) \subseteq PoDEF(S)$$

さらに,次のように定義する.

$$\begin{aligned} SuDEF_{var}(S) &\equiv \{d_{var} \mid d \in SuDEF(S)\} \\ PoDEF_{var}(S) &\equiv \{d_{var} \mid d \in PoDEF(S)\} \end{aligned}$$

この確実定義集合と潜在定義集合を使うと,式4.1は次のように書き換えることができる.

$$RD_{out}(S) = RD_{in}(S) - \{\langle v, n\rangle \mid v \in SuDEF_{var}(S)\} \cup PoDEF(S) \tag{4.2}$$

また,次に示す条件をすべて満たす変数vを「手続きpで暗使用される変数」と呼び,その集合を$ImUSE(p)$と表す.

- vは大域変数である
- p内でのvの参照地点にp外のvの定義が到達する

$ImUSE(p)$に含まれる変数は手続きpの呼び出し地点sには直接現れないが,pの呼び出しによって参照されると解釈して,$RD_{in}(s)$に含まれる定義節点からsへのDD関係を作るために使われる.

[2] S以降の文に到達しない

4.3.2 プログラム全体の解析

プログラムは手続きをひとつの単位として解析される. 手続きpの解析にはp自身の構造に関する情報以外にpが直接呼び出す手続きqの確実定義集合(SuDEF(q)), 潜在定義集合(PoDEF(q)), 暗使用される変数集合(ImUSE(q))に関する情報が必要となる. しかし, 再帰を含むプログラムを解析する際には, qを解析する前にqを呼び出す別の手続きpを解析しなければならない状況が起こり得る. よって, 手続きの解析を開始する前に, すべての手続きに対して, その SuDEF, PoDEF, ImUSE の初期値を与える必要がある.

手続きrが別の手続きsを呼び出すが, sはrを呼び出さない時, rの定義集合はsの定義集合に依存するので, sの解析の後に, rの解析をする方が, rの定義集合が早く収束することは明らかである. この様に, 手続き間の呼び出し関係を解析し, 手続きの解析順序をうまく選ぶことにより, 手続きの総解析回数を少なくできる.

解析順序を決定するためには, 手続き間の「呼ぶ—呼ばれる」関係を知る必要があるので, 入力プログラムに対して, 手続き名を節点で, また, 「呼ぶ—呼ばれる」関係を, 呼ぶ側から呼ばれる側への有向辺で表した呼び出しグラフ (Call Graph, 以下 CG) をつくる. あるプログラムの CG が有向無閉路グラフ (Directed Acyclic Graph, 以下 DAG) になる時, そのプログラムには, 相互/自己再帰呼び出しがない.

手続きが再帰的に定義されている時, その解析に使われる情報が不正確な場合があり, 通常, その手続きを一度解析しただけでは, その定義集合は収束していない. そこで, 関連する再帰呼び出しを含む手続き集合を, その他の手続き集合と分離して, それぞれの手続きの定義集合が収束するまで, その集合内の手続きのみを解析し続ける. 集合内の解析順序は, 要素間の「呼ぶ—呼ばれる」関係を考慮して, 決定される.

関連する再帰呼び出しを含む手続き集合は, 直接または間接的に相互に呼び出す可能性のある手続きの集合であるから, CG 上では強連結成分 (Strongly Connected Component) として現れる. 再帰呼び出しを含まない手続きは, その手続きひとつだけからなる強連結成分として抽出される. CG 上での強連結成分の抽出方法については参考文献 [35] の 10 章を参照されたい.

次に, 強連結成分間の解析順序を決めなければならない. 強連結成分をひとつの節点として, 強連結成分間をつなぐ CG 上の辺のみをその辺としてもつ縮約グラフ (Reduced Graph) CG′ を作ると, これは強連結成分の定義から DAG になる. CG′ 内の辺は, 強連結成分間の「呼ぶ—呼ばれる」関係を表しているので, CG′ 上をその根から深さ優先探索をして, ポストオーダーで順序付けすれば, これが強連結成分間の最適な解析順序になる.

以上をまとめると, 次のようになる.

アルゴリズム ANALYZEPROGRAM

Input: プログラム P

Output: P の PDG

Step 1. プログラム P の CG 上で強連結成分を抽出して, その縮約グラフを作り, その根から深さ優先探索して, ポストオーダーで順序付けする（ただし, 一回の探索ですべての節点に到達できない時は, 未到達の節点がなくなるまで繰り返し, すべての節点を順序付けする）.

Step 2. 各強連結成分内の要素を任意の要素から深さ優先探索して, ポストオーダーで順序付けする

Step 3. 各手続きの SuDEF, PoDEF, ImUSE を以下のように初期化する (ただし, 変数 SuDEF(f) は手続き f 全体に対する確実定義集合を意味する).

$$\begin{aligned} \texttt{SuDEF}(f) &\leftarrow \phi \\ \texttt{PoDEF}(f) &\leftarrow \phi \\ \texttt{ImUSE}(f) &\leftarrow \phi \end{aligned}$$

Step 4. Step 1 で決定した順序で強連結成分 S をひとつずつ選び以下の手順で解析する

(a) S 内の各手続きを Step 2 で決定した順序で一回ずつアルゴリズム ANALYZEPROCEDURE を使って解析する（アルゴリズム ANALYZEPROCEDURE は **4.3.3.** で説明する）.

(b) S 内のすべての手続きの SuDEF, PoDEF, ImUSE が変化しなくなるまで (a) を繰り返す.[3]

Step 5. メインプログラムを解析する

4.3.3 ひとつの手続きの処理

ひとつの関数 f は, 一般に図 4.6 のような構造で表される. ここではその本体の領域を B とする. この f は以下の手順で解析される（手続きの場合, 関数と違って戻り値がなく, exit 節点が不要なので, この節点に関する辺を作らないところを除けば関数の場合と同じである）.

function $f\,(\ldots) : \cdots ;$
var $\cdots$
begin
$\vdots$ } B
end;

図 4.6: 関数定義の概略

[3]ただし, S が再帰呼び出しを含まない手続きひとつだけからなる時は, 繰り返し解析してもその手続きの SuDEF, PoDEF, ImUSE は変化しないので, その手続きを一回解析するだけで S の解析を終える

表 4.2: 各文での確実定義集合の計算方法

	各文の確実定義集合
代入文	左辺の変数が確実定義集合に入る
条件文	then 節と else 節それぞれの確実定義集合の積集合が文全体の確実定義集合となる
繰り返し文	確実定義はない
複合文	各文の確実定義集合の和集合が全体の確実定義集合となる
手続き呼び出し文	呼び出される手続きの確実定義集合が文全体の確実定義集合となる
入力文	入力値を代入される変数が確実定義集合に入る
出力文	確実定義はない

表 4.3: 各文での潜在定義集合の計算方法

	各文の潜在定義集合
代入文	左辺の変数が潜在定義集合に入る
条件文	then 節と else 節それぞれの潜在定義集合の和集合が文全体の潜在定義集合となる
繰り返し文	条件成立時に実行される文の潜在定義集合が全体の潜在定義集合となる
複合文	各文の潜在定義集合の和集合が全体の潜在定義集合となる
手続き呼び出し文	呼び出される手続きの潜在定義集合が文全体の潜在定義集合となる
入力文	入力値を代入される変数が潜在定義集合に入る
出力文	潜在定義はない

ただし, 複合文において, 同じ変数に対する定義がある場合, より後の定義が全体の定義集合に含まれる (式 4.2 参照). また, 文が含む式に関数呼び出しがある時, 呼び出される関数の定義集合をその文の定義集合に加える. また, 呼び出される関数で暗使用される変数がその文で参照されたように DD 関係辺をつなぐ.

アルゴリズム ANALYZEPROCEDURE

Input: 手続き f

Output: $\mathtt{SuDEF}(f)$, $\mathtt{PoDEF}(f)$, $\mathtt{ImUSE}(f)$ の変化の有無

Step 1. $\mathtt{RD}_{\mathtt{in}}(B) \leftarrow \{\langle u, f_u\text{–}par\rangle \mid u \text{は} f \text{の仮引数変数}\} \cup \{\langle v, f_v\text{–}in\rangle \mid v \text{は大域変数}\}$

Step 2. B 内の文を表 4.2 , 表 4.3 に基づいて解析する[4]. これによって, f 内の CD 関係辺, DD 関係辺が作られ, $\mathtt{RD}_{\mathtt{out}}(B)$, $\mathtt{SuDEF}(B)$, $\mathtt{PoDEF}(B)$, $\mathtt{ImUSE}(B)$ の値が計算される (ただし, $\mathtt{ImUSE}(B)$ は f が呼び出す手続きの現在の ImUSE の値を使って計算し直した結果を意味する) .

Step 3. 次のようにして, 各変数の変化を調べる (ただし, *Global* は変数集合の中の大域変数だけを選ぶ事を意味する) .

$$\begin{aligned} \mathtt{SuDEF}_{\mathtt{var}}(f) &== Global(\mathtt{SuDEF}_{\mathtt{var}}(B)) \\ \mathtt{PoDEF}_{\mathtt{var}}(f) &== Global(\mathtt{PoDEF}_{\mathtt{var}}(B)) \\ \mathtt{ImUSE}(f) &== \mathtt{ImUSE}(B) \end{aligned}$$

[4] より詳しくは, 参考文献 [36] を参照されたい

左辺の各変数は, 初期値として与えたもの, または, この解析が二度目かそれ以降の時は, 前回の解析時に代入されたものである. これらの内, ひとつでも変化したものがある時は, 次の操作をして定義集合が変化したことを記録しておく.

$$\begin{aligned}
\mathtt{SuDEF}(f) &\leftarrow \{\langle v, f_v\text{–}out\rangle \mid v \in Global(\mathtt{SuDEF}_{\mathtt{var}}(B))\} \\
\mathtt{PoDEF}(f) &\leftarrow \{\langle u, f_u\text{–}out\rangle \mid u \in Global(\mathtt{PoDEF}_{\mathtt{var}}(B))\} \\
\mathtt{ImUSE}(f) &\leftarrow \mathtt{ImUSE}(B)
\end{aligned}$$

Step 4. $\mathtt{RD}_{\mathtt{out}}(B)$ の中から, 関数の戻り値に関する定義を探し, その定義節点 n から exit 節点への DD 関係辺 $(n \xrightarrow{f} f\text{–}exit)$ を作る. また, 同様に, 各大域変数 g に関する定義を探し, その定義節点 m から, out 節点への DD 関係辺 $(m \xrightarrow{g} f_g\text{–}out)$ を作る.

4.3.4 解析例

ここでは, **4.3.2.** で示したアルゴリズム ANALYZEPROGRAM を使って, 図 4.7 の左のプログラムを解析する様子を示す.

はじめに, Step 1 でプログラム progression の CG を作り, 強連結成分を抽出すると (図 4.8), 手続きの解析順として, $\{f\} \Rightarrow \{nth\}$ が得られる. 次に, Step 2 を行ない, Step 3 で各変数に初期値を代入する. その後, Step 1 で得られた解析順に従って, Step 4(a) で $\{f\}$ 内の手続き f を ANALYZEPROCEDURE を使って解析するとその本体 B での定義集合が次のようになる.

$$\begin{aligned}
\mathtt{SuDEF}_{\mathtt{var}}(B) &= \{\mathtt{f}, \mathtt{g}\} \\
\mathtt{PoDEF}_{\mathtt{var}}(B) &= \{\mathtt{f}, \mathtt{g}, \mathtt{l}\} \\
\mathtt{ImUSE}(B) &= \phi
\end{aligned}$$

f は大域変数 g を文 9 で参照するが, この参照の前に文 8 の手続き呼び出しで, g を必ず定義するので, $\mathtt{ImUSE}(B)$ に g は含まれない. この時, $\mathtt{SuDEF}_{\mathtt{var}}(B)$ と $\mathtt{PoDEF}_{\mathtt{var}}(B)$ の値は初期値から変化しているので, 各変数の値は以下のように更新され, Step 4(b) によって再度 $\{f\}$ に対して Step 4(a) が実行される.

$$\begin{aligned}
\mathtt{SuDEF}(f) &\leftarrow \{\langle \mathtt{g}, f_g\text{–}out\rangle\} \\
\mathtt{PoDEF}(f) &\leftarrow \{\langle \mathtt{g}, f_g\text{–}out\rangle\} \\
\mathtt{ImUSE}(f) &\leftarrow \phi
\end{aligned}$$

($\{f\}$ はひとつの要素しか持たないが, その手続き f が再帰呼び出しを含むので再度解析しなければならない.)

二回目の $\{f\}$ の解析では定義集合が変化しないので $\{f\}$ の解析が終る. 次に Step 4(a) により, $\{nth\}$ の解析をする. $\{nth\}$ の一回目の解析により, その定義集合は次のように更新

```
 1  program progression(input,output);
 2      var     g:integer;
 3
 4      function f(p:integer):integer;
 5          var     l:integer;
 6      begin
 7          if p>1 then begin
 8              l := 2 * f(p-1);
 9              g := g + l;
10              f := l;
11          end
12          else begin
13              g := 1;
14              f := 1;
15          end;
16      end;
17
18      procedure nth;
19          var     n,a:integer;
20      begin
21          g := 1;
22          readln(n);
23          a := f(n);
24          writeln(a);
25      end;
26  begin
27      g := 0;
28      nth;
29      writeln(g);
30  end.
```

```
 1  program progression(input,output);
 2      var     g:integer;
 3
 4      function f(p:integer):integer;
 5          var     l:integer;
 6      begin
 7          if p>1 then begin
 8              l := 2 * f(p-1);
 9
10              f := l;
11          end
12          else begin
11
14              f := 1;
15          end;
16      end;
17
18      procedure nth;
19          var     n,a:integer;
20      begin
21
22          readln(n);
23          a := f(n);
24          writeln(a);
25      end;
26  begin
27
28
29
30  end.
```

図 4.7: スライスの計算例

される.

$$\begin{aligned} \texttt{SuDEF}(nth) &\leftarrow \{\langle \mathrm{g}, nth_g\text{–}out\rangle\} \\ \texttt{PoDEF}(nth) &\leftarrow \{\langle \mathrm{g}, nth_g\text{–}out\rangle\} \\ \texttt{ImUSE}(f) &\leftarrow \phi \end{aligned}$$

$\{nth\}$は再帰呼び出しを含まない手続きひとつからなる強連結成分なので解析を繰り返してもその定義集合は変化しない. よって, $\{nth\}$の解析は一回で終る.

最後に, Step 5でメインプログラムを解析してプログラム全体の解析を終える.

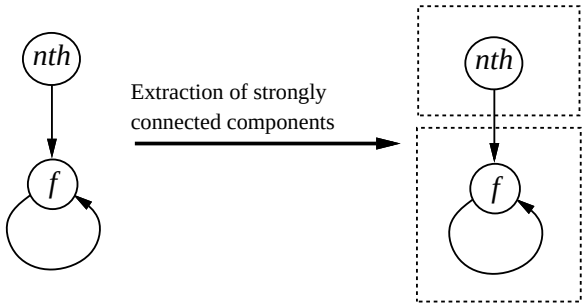

図 4.8: プログラム progression に対する CG

4.4 スライスの計算

この節では PDG 上でスライスを計算する方法を示す（図 4.2 の破線矢印部分）. 本アルゴリズムでは,「文 s における変数 v に関するスライスとは, PDG 上で CD 関係辺または DD 関係辺を辿って文 s の変数 v に到達できる節点集合に対応する文の集合である」と定義したので, PDG 上でのスライスの計算は節点間の到達可能性の判定と同じである. ただし, 本論文で定義した PDG には手続きの境界を越えてスライスを計算できるようにいくつかの拡張がなされているので, たどることのできる辺に制限が付く.

プログラム内の文 s での変数 v に関するスライスを表す節点の集合 V は, 以下に示すような手順で計算できる. ただし, n_s は s に対応する節点である.

アルゴリズム GETSLICEONPDG

Input: PDG, 文 s, 変数 v

Output: スライスを表す節点の集合 V

Step 1. $V \leftarrow \{n_s\}$

Step 2. $N \leftarrow \{n \mid n \xrightarrow{v} n_s\}$[5] $\cup \{m \mid m \dashrightarrow n_s\}$

Step 3. $N \neq \phi$ の間, 以下の操作を繰り返す

(a) $n \in N$ をひとつ選ぶ

(b) $N \leftarrow N - \{n\}$

(c) $V \leftarrow V \cup \{n\}$

(d) n が in 節点でも引数節点でもないなら次の操作をする

$N \leftarrow N \cup \{m \mid m \notin V \wedge (m \longrightarrow n \vee m \dashrightarrow n)\}$

（ただし $m \longrightarrow n$ は m から n への任意の変数の DD 関係辺を表す）

スライスを計算する対象を文 s 全体にするには Step 2 を次のように変更すれば良い.

[5] より正確には $\{n \mid \langle v, n\rangle \in RD_{in}(s)\}$ であるが, 一般に, ユーザーは変数 v の参照を含む文を s に指定する場合が多いのでここでは, このわかりやすい表現を用いた.

Step 2′.　　$N \leftarrow V$

また, 文の集合Sに対するスライスを計算する時は, さらに, Step 1を次のように変更すれば良い.

Step 1′.　　$V \leftarrow \bigcup_{s_i \in S} (s_i$を表す節点$)$

節点とプログラムの断片は一対一に対応しているので, Vが求まると, それに対応したプログラムを作り上げることは簡単である. 例えば, 図4.9のPDG上で文`writeln(a)`での変数`a`についてのスライスを計算した結果が図4.9中の破線四角形で示された節点であり, この節点集合に対応するソースコードの部分を集めたものが図4.7の右のプログラムとなる.

4.5 アルゴリズムの複雑さ

この章では**4.3.**, **4.4.**で紹介したアルゴリズムの複雑さについて述べる. 本論文のアルゴリズムは, ソースコードをPDGに変換する部分(AnalyzeProgram)と, PDG上でスライスを計算する部分(GetSliceOnPDG)とに分けられるので, 複雑さについても両者を分けて考える.

4.5.1 PDGを作るコスト

ソースコードからPDGを作るための解析に関わる要素を以下の表の様に定義する.

PDG作成に関わる要素	
P	手続きの総数
G	大域変数の総数
L	手続きの局所変数の数の最大値
S_i	手続き呼び出しの総数
S_t	文の総数

補題1 プログラム中のすべての手続きが再帰呼び出しを含む時が解析に最も時間がかかる

> **(証明)**　すべての手続きを少なくとも一回は解析しなければならない事は明らかである. 再帰呼び出しを含まない手続きは, その手続きひとつだけからなる強連結成分に含まれる. この強連結成分の解析はそれが含む手続きを一回解析するだけで終るので, このような手続きが存在しない時に, すなわち, プログラム中のすべての手続きが再帰呼び出しを含む時に, プログラム全体の解析に最も時間がかかる. □

補題2 手続きの`SuDEF`, `PoDEF`, `ImUSE`に含まれる要素は, 解析の繰り返しによって単調に増加する

(証明) 手続きp内での, 手続き呼び出しによらない直接的な, 確実定義集合をS_dとすると, SuDEF(p)はS_dとpが直接呼び出す別の手続きのSuDEFだけを変数として持つ, 否定を含まない積と和だけからなる論理式で表現される. この論理式とS_dの値はプログラムが変化しない限り変化せず, 各手続きのSuDEFは最小(空集合)に初期化されるので, SuDEF(p)に含まれる要素は単調に増加する.

同様の事がPoDEF(p)とImUSE(p)についてもいえるので, これらに含まれる要素も単調に増加する. □

補題3 アルゴリズムANALYZEPROGRAMではひとつの強連結成分Sに含まれる手続きの解析(*Step 4(a)*)の繰り返しは高々$|S|$回で終る(ただし, $|S|$はSに含まれる手続きの数を表す)

(証明) 強連結成分Sに含まれる手続きpのSuDEF$_{\mathtt{var}}(p)$に大域変数gが含まれている事を$\mathrm{g}_p = true$で, 含まれていない事を$\mathrm{g}_p = false$で表現する.

SuDEF(p)は, 手続きp内での, 手続き呼び出しによらない直接的な, 確実定義集合とpが直接呼び出す別の手続きのSuDEFだけを変数として持つ, 積と和だけからなる論理式で表現される.

本論文の入力言語仕様では別名*(Alias)*が発生しないので, どの大域変数も互いに影響を及ぼす事はない. よって, SuDEF(p)を変数ごとに分解する事ができ, g_pはpが直接呼び出す手続き$q_1, q_2, \ldots$に関する$\mathrm{g}_{q_1}, \mathrm{g}_{q_2}, \ldots$を変数として持つ, 積と和だけからなる論理式で表現される. この$\mathrm{g}_{q_1}, \mathrm{g}_{q_2}, \ldots$の内, Sに含まれない手続きに関するものはすでに値が決定しており, 変化する事はない.

また, 補題*2*より, あるg_pが初期値$false$から一旦$true$に変化すると再び$false$に変化する事はない.

S全体を一回解析して, Sに含まれる手続きpのどのg_pも変化しなければ, その後の解析で変化する事はないので, S全体を一回解析するごとにSに含まれるひとつの手続きpのg_pにだけ変化が起こる時, Sの解析の回数が最大となる. これは, SuDEFだけでなく, PoDEF, ImUSEに対しても同様である. これはANALYZEPROGRAMが停止する事と同時に, Sに含まれる手続きの解析は高々$|S|$回で終る事を示している. □

定理1 アルゴリズムANALYZEPROGRAMの時間計算量は$O(P + S_i + P \cdot S_t \cdot (G + L))$である

(証明) アルゴリズムANALYZEPROGRAMの*Step 1*, *Step 2*に要する時間はプログラムの*CG*の節点と有向辺の数の和に比例する. 節点の数はPで, 有向辺の数はS_iで, それぞれ抑えられる. *Step 3*は手続きの数に比例する時間で終る.

ANALYZEPROGRAM の繰り返し部分 *Step 4(a)* は, 補題 *1, 3* より, プログラム中のすべての手続きがひとつの強連結成分に含まれている時, 最も時間がかかる. この時, プログラム中のすべての手続きが最大 P 回ずつ解析される.

プログラム中のすべての手続きを一回ずつ解析する, すなわち, プログラム中のすべての文を一回解析するのに要する時間は, 文の数に比例する. ひとつの文の解析には定数回の集合演算が必要である. 一回の集合演算にかかる時間が集合に含まれる要素の数に比例すると仮定すると, ひとつの文の解析の時間は大域変数と局所変数の数の和に比例する.

ゆえに, アルゴリズム ANALYZEPROGRAM の時間計算量は $O(P+S_i+P\cdot S_t\cdot(G+L))$ である. □

4.5.2 PDG 上でスライスを計算するコスト

PDG に関わる要素を以下のように定義する.

PDG 上でのスライスの計算に関わる要素
V　PDG の節点の総数
E　PDG の有向辺の総数

アルゴリズム GETSLICEONPDG では, PDG 上でスライスを計算する際, すべての節点とすべての有向辺を多くとも一回しか辿らないので, そのコストは $O(V+E)$ である.

4.5.3 解析全体にかかるコスト

ソースコードから始めてスライスを計算するまでに $O(P+S_i+P\cdot S_t\cdot(G+L)+V+E)$ の時間がかかるが, その後ソースコードに変更がなければ, $O(V+E)$ の時間でスライスの再計算ができる.

ただし, これはプログラム中のすべての手続きが互いに呼び合う可能性がある場合で, この様なプログラムは, 非常に稀にしか存在しないと考えられる. よって, 一般的なプログラムに対しては, $O(P+S_i+S_t\cdot(G+L)+V+E)$ の時間でスライスの計算ができる.

4.6 むすび

本論文では, 再帰を含むプログラムにおいて, 文の間の依存関係を解析し, プログラムを PDG に変換し, その上でのスライスの計算するアルゴリズムを紹介した. 本アルゴリズムは, 実際の解析が行なわれる前に, ひかえめな解を初期値として定義しておき, 必要に応じて解析を繰り返して真の解に収束させるというもので, 再帰を含むプログラムの解析には適したものであると思われる.

再帰を含むプログラムのスライスを計算するアルゴリズムは, 他に, Hwang によるもの [18] と Horwitz によるもの [21] がある. 前者のアルゴリズムは, 再帰呼び出しの深さ毎に,

スライスを計算し,それらの和集合が収束するまでその計算をするというもので,PDGを作らない. よって,プログラムを解析してPDGで表し,その上でスライスを計算する本論文のアルゴリズムに比べると,スライスを何度も計算する時に非常に不利になり,実際のデバッグ環境を考えると満足できるものであるとは言いがたい. また,ソースコードの変更の影響が及ぶ箇所を知るためなどに使われる順方向スライス(Forward Slice)を計算するために,PDGをそのまま使う事ができる点でも本論文のアルゴリズムは有利である.

後者のアルゴリズムに比べると,本論文のアルゴリズムでは,プログラム内の各文の到達定義集合を計算しているので,プログラムの任意の地点の任意の変数に対するスライスが計算できる点,および,ソースコードが変更された時に,再解析が必要になる部分をより少なくでき,インクリメンタルにPDGを更新する方法をとる事ができる点で有利であると考えられる.

本アルゴリズムに従ってスライスを計算し,表示する試作システムを作成し,本アルゴリズムが正しく動作する事を確認した. このシステムはSUN SPARCstation ELC上で動作し,13個の手続き,7個の大域変数を含む再帰呼び出しのない249行のプログラムを0.80秒で,また,8個の手続き,9個の大域変数を含み,すべての手続きがお互いに呼びあう可能性のある275行のプログラムを1.82秒で,それぞれ解析する事ができる. 今後,このシステムをより使いやすいユーザーインターフェイスを備えた本格的なシステムにする予定である.

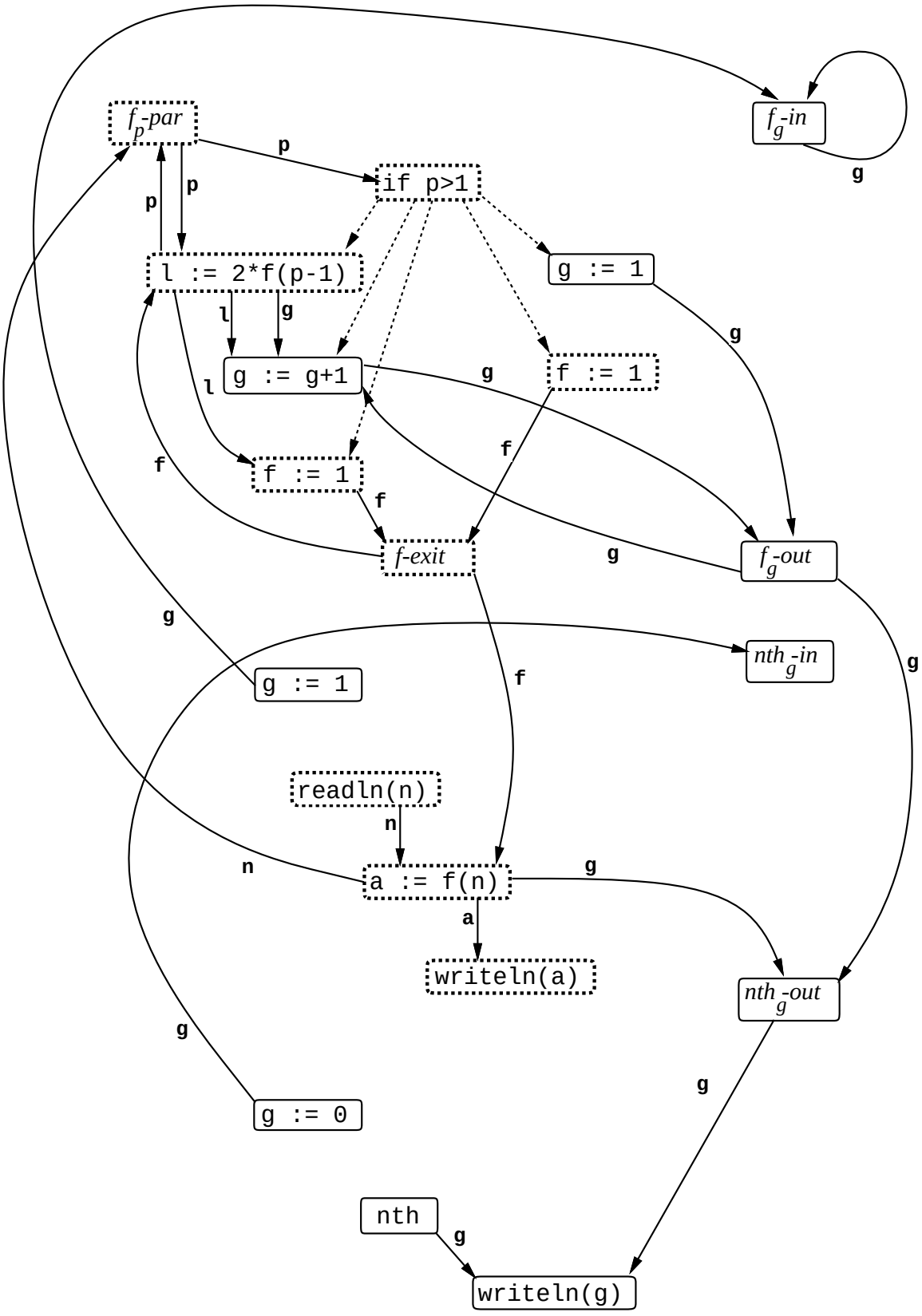

図 4.9: プログラム progression に対する PDG：破線四角形は文 `writeln(a)` での変数 a に関するスライスを計算した時の節点集合に含まれる節点である

第5章　まとめ

5.1　これまでの研究成果

本論文では，オンラインサービスシステムの安定的な稼働に資する以下の3つの手法を提案／評価した．

(1) Webサービスシステムの応答性能劣化診断のための学習データ自動選定方法

計算機システムのCPU／ネットワーク等の利用状況のメトリクス値を入力として，機械学習を組み合わせてシステムの状態を診断する際に，学習データを自動的に選定する手法を考案した．オンライン販売システムを使い，あらかじめ用意した初期学習データに加えて，提案手法で自動選定した学習データを追加した場合の診断結果と，全データによる診断結果の類似性を判定する評価実験を行った結果，10種中7種で，より少量のデータで全データ学習と同等の学習効果が得られることを確認した．

(2) Grid技術のWebサービスシステムへの適用方法

Webサービスシステムの処理性能を必要に応じて拡大／縮小させるために，Grid技術を適用する手法を考案した．Grid基盤の動的リソース割当機能を活用するための，2層のロードバランサを持つスケーラブルなソフトウェアアーキテクチャを設計し，J2EEインターネットバンキングアプリケーションをGrid化，元のバージョンと性能比較することで，その有効性を検証した．その結果，Grid技術がWebサービスシステム全体の処理性能を拡大／縮小する方法として有効であることが分かった．

(3) 再帰を含むプログラムのスライス計算方法

再帰を含むプログラムに対して，プログラムのバグ発見に役立つスライスを計算する手法を考案した．手続きの境界を越えてデータフローを解析可能とするために，独自の改良を施したプログラム依存グラフを定義し，さらに，再帰を含むプログラムに対応するために，ひかえめな解を初期値から開始し，必要に応じて解析を繰り返して真の解に収束させる方法を採用した．本手法に従ってスライスを計算／表示する試作システムを実装し，本手法が，対話的にデバッグを行うシステムに組み込むために許容可能な時間で，正しく動作する事を確認した．

オンラインサービスシステムの安定稼働を達成するために，研究成果(1)の学習データ自動選定方法は，システムレベルのマクロな視点から，システムの応答性能が劣化した際に，それが，リクエスト過多に起因するものなのか，それとも，システムの障害に起因するものなのか，を判定する際に役立つ．

システムの性能劣化がリクエスト過多に起因するものであった場合，研究成果(2)のGrid技術を適用して，オンラインサービスシステムの処理能力を動的に拡大することでこれに対処する事ができる．

また，システムの性能劣化がシステムの障害に起因するものであった場合，今度はミクロの視点で研究成果(3)のスライス計算方法を活用することで，システム障害の原因のひとつとなるプログラム内のバグ発見を支援する事ができると考える．

データベース層がボトルネックとなり，Grid技術の適用だけではシステムの処理能力が拡大できない場合や，システム障害の原因がプログラムのバグではない場合については，これらの研究では十分な対策ができないが，これは今後の課題としたい．

5.2 今後の研究方針

今後，本論文で述べた研究成果を活用し，計算機システム，特に，オンラインサービスシステムの安定的な稼働に役立つ技術開発をしていきたいと考えている．

具体的には，機械学習を組み合わせて計算機システムの状態を診断する手法に関して，研究成果(1)で提案した手法では10種中7種のデータのみ有効性が確認できたが，残り3種については有効な結果が出せなかった．この点については，追加の実験から，今回の実験では固定長5分を採用したデータ区画の分割方法を，可変長に変更するという改良のヒントを得ており，この観点での検証を進める計画である．

さらに，実際のオンラインサービスシステムから得られた実データに提案手法を適用することで，精度の追加検証を進めるとともに，実験では遭遇しなかった課題を見つけ，本手法を改良していきたいと考えている．

また，将来は本手法を，自動スケールアウト等でシステム構成が動的に変化する状況にも対応可能とする拡張や，システム障害の検知だけではなく，[9, 10]同様，根本原因究明に役立つ情報を提供する拡張等を行い，システムの安定稼働に貢献する事を考えている．

参考文献

[1] 総務省. 情報通信白書平成27年版. http://www.soumu.go.jp/johotsusintokei/whitepaper/ja/h27/pdf/index.html, 2015.

[2] Nikkei. みずほ銀行、大規模障害を招いた「夜間バッチ処理」. http://www.nikkei.com/article/DGXZZO25303720Z10C11A3000000/, 2011.

[3] 八嶋俊介. 障害事例から学ぶ高信頼化へのアプローチ. Technical report, 独立行政法人情報処理推進機構（IPA）, Oct. 2015.

[4] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. In *The International Conference on Dependable Systems and Networks*, pp. 644–653, Yokohama, Japan, Jun. 2005.

[5] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J.S. Chase. Correlating instrumentation data to system state: A building block for automated diagnosis and control. In *USENIX Association OSDI'04: 6th Symposium on Operating Systems Design and Implementation*, pp. 231–244, 2004.

[6] 鈴木英明, 内山宏樹, 湯田晋也. データマイニングによる異常検知技術. オペレーションズ・リサーチ経営の科学, Vol. 57, No. 9, pp. 506–511, Sep. 2012.

[7] E. Stehle, K. Lynch, M. Shevertalov, C. Rorres, , and S. Mancoridis. Diagnosis of software failures using computational geometry. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 496–499, Lawrence, KS, USA, Nov. 2011.

[8] E. Stehle, K. Lynch, M. Shevertalov, C. Rorres, and S. Mancoridis. On the use of computational geometry to detect software faults at runtime. In *7th International Conference on Autonomic Computing, ICAC 2010*, pp. 109–118, Washington, DC, USA, Jun. 2010.

[9] S. Iwata and K. Kono. Clustering performance anomalies based on similarity in processing time changes. *IPSJ Transactions on Advanced Computing Systems*, Vol. 5, No. 1, pp. 1–12, Jan. 2012.

[10] T.H.D. Nguyen, M. Nagappan, A.E. Hassan, M. Nasser, and P. Flora. An industrial case study of automatically identifying performance regression-causes. In *MSR'14*, pp. 232–241, Hyderabad, India, May 2014.

[11] 南波幸雄. 企業情報システムアーキテクチャ. 翔泳社, Apr. 2009.

[12] 独立行政法人情報処理推進機構（IPA）. クラウド・コンピューティング社会の基盤に関する研究会報告書. http://www.ipa.go.jp/files/000009362.pdf, Mar. 2010.

[13] Capers Jones, Olivier Bonsignour : 小坂恭一監訳. ソフトウェア品質の経済的側面. 共立出版, Dec. 2013.

[14] M. Weiser. Program slicing. In *Proceedings of the Fifth International Conference on Software Engineering*, pp. 439–449, 1981.

[15] B. Korel and J. Laski. Dynamic program slicing. Technical report, Information Processing Letters, 1988.

[16] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. pp. 392–411, 1992.

[17] 下村隆夫. Program slicing 技術とテスト, デバッグ, 保守への応用. 情報処理, Vol. 33, No. 9, pp. 1078–1086, 1992.

[18] J.C. Hwang, M.W. Du, and C.R. Chou. Finding program slices for recursive procedures. In *Proceedings of the IEEE COMPSAC '88*, pp. 220–227, 1988.

[19] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM SIGPLAN Notices 19(5)*, pp. 177–184, 1984.

[20] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 35–46, 1988.

[21] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 1, pp. 26–60, 1990.

[22] bnlearn - An R package for Bayesian network learning and inference. http://www.bnlearn.com/.

[23] JPetStore. http://sourceforge.net/projects/ibatisjpetstore/files/jpetstore4/4.0.5/iBATIS_JPetStore-4.0.5.zip/download.

[24] Worldcup98 site access data. http://ita.ee.lbl.gov/html/contrib/WorldCup.html.

[25] The R Project for Statistical Computing. http://www.r-project.org/index.html.

[26] Globus Project. Globus toolkit 3.2 documentation. http://toolkit.globus.org/toolkit/docs/3.2/.

[27] B. Sotomayor. The Globus toolkit 3 programmer's tutorial. http://www.casa-sotomayor.net/gt3-tutorial/, May 2004.

[28] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications*, Vol. 15, No. 3, 2001.

[29] S. Bodoff, D. Green, K. Haase, E. Jendrock, M. Pawlan, and B. Stearns. *The J2EE Tutorial*. Addison-Wesley, Mar. 2002.

[30] C. Bauer and G. King. *Hibernate in Action*. Manning Publishing Company, Aug. 2004.

[31] S. Stark and the JBoss Group. *JBoss Administration and Development, 3rd Edition (3.2.x Series)*. Jun. 2003.

[32] D. Davis and M. Parashar. Latency performance of SOAP implementations. In *The 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID2002)*, pp. 377–382, 2002.

[33] L. Schwartz, L. Letoutneaut, N. Carey, and S. Kaufer. A grid-enabled electronic commerce application for small and medium business. In *GlobusWORLD Conference*, Jan. 2004.

[34] H. Wang, H. Jin, F. Yuan, and L. Pang. Session management for collaborative applications in interactive grid. In *IEEE International Conference on Computer Systems and Applications*, pp. 712–715, Mar. 2006.

[35] J.H. Kingston. *Algorithms and Data Structures : Design, Correctness, Analysis*. Addison-Wesley, 1990.

[36] 植田良一, 練林, 井上克郎, 鳥居宏次. 再帰を含むプログラムの依存関係解析とそれに基づくプログラムスライシング. Technical Report SS93–24, 信学技報, Sep. 1993.