先進的基盤ソフトウェア 4 巻 1 号（通号 4 号）オンライン版 2010 年 10 月 22 日発行
ISSN 1882-4196

先進的基盤ソフトウェア

Joint Symposium for Advanced System Software 2010
(JSASS2010)

2010 年 8 月 30 日(月)・31 日(火)

於 名古屋工業大学 (愛知県名古屋市)

JSASS 実行委員会

巻頭言

名古屋工業大学　齋藤彰一

JSASS も今年で 11 年となりました。本会は、普段の研究室だけでは得られない大勢の前での発表や、他の大学の先生からの御意見を頂戴する場として大変貴重な場を提供していると感じています。学会の研究会とは違い、先生方と学生がより近い場で、やや言いにくいこともコメントできるのではないかと思っております。また、研究室だけではどうしてもこぢんまりとまとまってしまいかねないところです。それを、打ち破り、さらなる飛躍を得るためのきっかけとして活用していただければと考えております。

JSASS は「Advanced System Software」をメインテーマにあげております。「System Software」分野は、情報システムやネットワークシステムを支える非常に重要な分野でありますが、裏方的な存在でもあります。そのため、アプリケーションやインターネット（ここでは Web という意味）のように人々の注目を集めることは珍しいでしょう。逆に、正しく動いて当たり前であるために、トラブル時のマイナスの注目のほうが目立つのではないでしょうか。しかし、System Software はなくてはならない基盤技術です。System Software の発達は、アプリケーションの可能性を広げ、さらなる情報化社会の発展や安全な社会の実現に貢献できるのではないでしょうか。そのために従来の System Software を超える Advanced System Software を議論する場として、JSASS を今後も発展させていこうと考えております。

さて、今年の発表を見てみると、オペレーティングシステムに関するテーマが核となっておりますが、セキュリティや広域分散環境、センサーネット、アーキテクチャと多様な内容となり、合計 24 件もの発表がありました。多様な分野の発表が同時に行われることは、新しい刺激やヒントを得るチャンスです。特に学生のみなさんには、特定の分野にこだわらずに様々な分野に触れてほしい

と思いますので、JSASS を活用してほしいと思います。また、発表の完成度も様々であり、完成度の高い発表もあれば、発展途上の発表もありました。参加者からは、多数の質問やコメントがあり活発な議論ができました。発表者のみなさん、特に発展途上の研究の発表者には、これらのコメントを活かしていただき、良い結果をまとめていただきたいと強く願っております。

最後になりましたが、今年の会場は名古屋工業大学で 3 年ぶり 2 度目です。私のローカルアレンジとしては、第 3 回の和歌山大学を含めて 3 回目となります。いつもですが、会場の準備や懇親会の手配など気を使うことが多く大変です。同時に、これまで会場のお世話を引き受けていただいた皆様に、感謝の気持ちでいっぱいになります。この場を借りてお礼を申し上げます。さらに、幹事の皆様、特に龍谷大学の芝公仁先生、ポストプロシーディング編集の立命館大学毛利公一先生に感謝いたします。また、すべての皆様に、今後の御協力をお願い申し上げます。

2010 年 9 月 30 日記

Joint Symposium for Advanced System Software 2010 (JSASS2010)

2010 年 8 月 30 日(月)・31 日(火)

名古屋工業大学 (愛知県名古屋市)

プログラム

■ 8 月 30 日(月)

○ オープニング: 13:00～13:10

○ セッション 1: 13:10～14:25 セキュリティ(1)

○ セッション 2: 14:40～15:55 仮想化

○ セッション 3: 16:10～17:50 分散システム

○ 懇親会: 18:10～20:00

■ 8 月 31 日(火)

マルウェア検知における誤検知率削減手法の提案

古屋　雄介 †
† 名古屋工業大学

1　はじめに

現在，インターネットの普及とともにPC上で悪さを行うプログラムであるマルウェアが横行している．各セキュリティ企業により報告されるマルウェアの発見件数は増加の一途を辿っており，近年ではマルウェアは愉快犯的なものから金銭目的なものへと移り変わっており，個人情報の流出等の深刻な被害が多く発生している．

一般にマルウェアへの対策としてアンチウイルスソフトが用いられている．アンチウイルスソフトの多くはパターンマッチングによるマルウェアの検知を行う．この手法は発見された既知のマルウェアのバイナリコードからパターンを作成し，このパターンとPC上のプログラムの比較を行うことでマルウェアを検知する．しかし，検知を行うためにはマルウェアを収集しパターンを作成しておくことが必要であるため，未知のマルウェアを検知することができない．そこで，現在ビヘイビア検知と呼ばれる手法が注目されている．ビヘイビア検知はマルウェアの挙動をより抽象的に定義し，実際に検査対象のプログラムを動作させて挙動が一致するか調べる手法である．

以降，第2章で既存のビヘイビア検知手法とその問題点を説明した後，第3章で本提案手法，第4章でシステムの実装，第5章で安全性の考察を述べ，最後にまとめる．

2　既存手法とその問題点

本章では既存のビヘイビア検知手法とその問題点を順に述べる．

2.1　既存のビヘイビア検知手法

ここで酒井らによる既存のビヘイビア検知手法 [1] を取り上げる．この手法ではマルウェアの挙動として実行プロセスの自己ファイルのREADとDELETEを定義し，検査対象プロセスにこれらの挙動が見られた場合に検知を行う．この定義は一般にマルウェアがシステムに侵入後，自身の存在を隠すために自己ファイルをシステムフォルダに移動させるマルウェア独自の挙動に基づく．酒井らが行った実験では使用したマルウェア9検体のうち8検体を検知することができているため，非常に有効な手法であると言える．

2.2　問題点

酒井らの手法は高い検知精度を誇るが，正規のプログラムを検知する誤検知の問題がある．例えば，アンインストーラの誤検知が挙げられる．アンインストーラは不要になったファイルを削除する目的を果たした後，自身を削除するが，このときの挙動がマルウェアの挙動定義と一致するため誤検知が起こる．そこで，本研究ではアンインストーラを誤検知することを解消し，マルウェア検知において誤検知率を削減することを目的とする．

3　提案手法

本章では既存手法の問題への解決手法を述べる．

3.1　基本的なアイデア

既存手法はマルウェアの挙動のみを観測して検知を行っている．そこで，本手法ではアンインストーラの挙動にも着目し，マルウェアには見られずアンインストーラに見られる挙動を別途定義する．アンインストーラに自己ファイルREAD/DELETEの挙動が見られた場合でも，同時にアンインストーラの挙動も観測されるため誤検知を防ぐことができる．

3.2　検知処理の流れ

提案手法追加後の検知の流れを図1に示す．システムはまず全プロセスの自己READ/DELETE，ウィンドウ生成の3つの挙動を監視し待機する．そして，自己READまたはウィンドウ生成の挙動が観測された場合，そのプロセスの挙動記録に登録する．その後，自己DELETEの挙動が観測された場合，そのプロセスの挙動記録を参照し，自己READのみが登録されていれば検知し，CreateWindowも登録されていれば検知はしない．このような順で挙動を観測するのはアンインストーラが自己READ，ウィンドウ生成を行った後に自己DELETEを行うはずだからである．

3.3　定義するアンインストーラの挙動

本手法ではアンインストーラの挙動としてGUIウィンドウの生成を定義する．これは一般にアンインストーラがユーザとのインタフェースとしてGUIを備えていることに基づく．

A Proposal for Reduction of Mis-detection ratio on Malware Detection
† Furuya Yusuke
Nagoya Institute of Technology (†)
Gokiso-cho, Showa-ku, Nagoya, Aichi, 466-8555 Japan

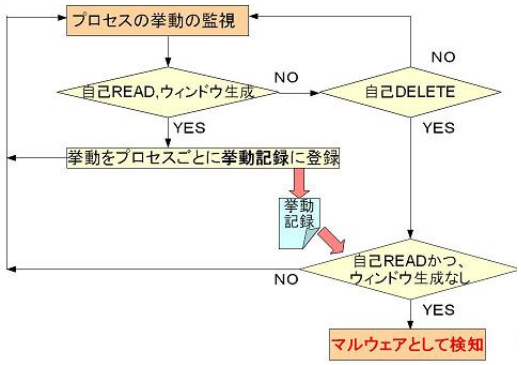


図 1: 検知の流れ

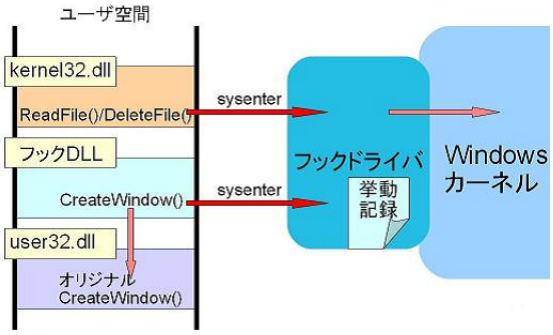


図 2: API フック後の処理の流れ

4 実装

本章では提案システムの実装について述べる．なお，対象とした OS は Windows XP SP2 である．

4.1 挙動の観測

検査対象プログラムの自己 READ/DELETE，ウィンドウの生成の挙動の観測を行うためにログラムが発行する API のフックを行う．具体的には自己 READ/DELTE を行う API はそれぞれ ReadFile, DeleteFile であり，ウィンドウの生成を行う API は CreateWindow であり，これらのフックを行う．フック機構を追加した後のプログラムの処理の流れを図 2 に示す．

4.2 API のフック

API のフックは API が sysenter 命令を発行するか否かで 2 つの方法に分類されるため以下に順に説明する．

4.3 ReadFile/DeleteFile API のフック

ReadFile/DeleteFile API は最終的に sysenter 命令を発行する．sysenter 命令はユーザモードとカーネルモードを切り替える命令である．そこで，Windows システム内に 1 箇所 sysenter 命令をフックする処理を加える．これにより，全プロセスにより発行される sysenter 命令をすべてフックできる．sysenter 命令をフックするために sysenter 命令発行時に参照される MSR レジスタの内容をフック処理を行うフックドライバ内のフック関数のアドレスに書き換える．

4.4 CreateWindow API のフック

CreateWindow API は sysenter 命令を発行しないため，フック処理は各プロセスのユーザ空間内で行う．各 API は DLL として実装されており，API 発行時は DLL 内の API 関数が呼ばれることになる．そこで，CreateWindow API が実装されている user32.dll と同じエクスポート関数を持つフック DLL を作成し，user32.dll と置き換えることでフックを行う．フック DLL 内の CreateWindow の処理を変更し，sysenter 命令を発行させることでフックドライバと情報を共有する．

5 安全性の考察

本章ではマルウェアによる本提案システムへの攻撃とその対策について述べる．

5.1 マルウェアによる本提案システムへの攻撃

マルウェアが本提案システムを回避する攻撃を行うことが考えられる．具体的にはマルウェアが CreateWindow API を発行し，自身をアンインストーラに見せかけ，検知を回避する攻撃である．この場合，本システムではマルウェアを正しく検知することができなくなる．

5.2 対策

一般にマルウェアは自身の存在を隠すことをひとつの目的とする．そのため，マルウェアはウィンドウを小さく表示したり，画面外に表示したり，表示後すぐに消去するといった挙動を見せると考えられる．そこで，CreateWindow API が発行された場合，その引数まで確かめることでマルウェアによる発行かどうかを調べることができる．これにより，マルウェアによる検知の回避を防ぐことができると考えられる．

6 まとめ

本稿ではマルウェア検知における誤検知率の削減手法を提案した．提案手法はマルウェアの挙動だけでなく，アンインストーラの挙動にも着目することで誤検知を解消するものであった．今後，本提案システム上で検知実験を行い，誤検知率が削減されるかを検証する．

参考文献

[1] 酒井崇裕, 長谷巧, 竹森敬祐, 西垣正勝: 自己ファイル READ/DELETE の検出によるボット検知の可能性に関する一考察, MWS2008(2008 年 10 月).

マルウェア検知における誤検知率削減手法の提案

古屋雄介

名古屋工業大学

目次

- 背景
- 既存手法
- 既存手法の問題点
- 提案手法
- システムの実装
- 実験(予定)
- まとめと今後の課題

2

背景

- 悪意あるプログラム(マルウェア)の横行
  - マルウェアの報告件数は急増している
  - アンチウイルスソフトが使用されている
- 検知の限界
  - 従来のパターンマッチング検知ではすべてのマルウェアへの対応はできない
  - ビヘイビア検知が有効である

3

代表的な検知手法

- パターンマッチング検知
  - **検知方法**
    - マルウェアをコードのパターンで定義し、静的に調べる
  - **長所**
    - パターンと一致していれば、ほぼ確実にマルウェアと断定できる
  - **短所**
    - 未知のマルウェアに対応できない
- ビヘイビア検知
  - **検知方法**
    - マルウェアのビヘイビア(挙動)をより抽象的に定義し(ファイルの読み書き等のAPI呼出し)、動的に調べる
  - **長所**
    - 未知のマルウェアに対応できる
  - **短所**
    - 定義した挙動と一致していても、確実にマルウェアと断定できない

4

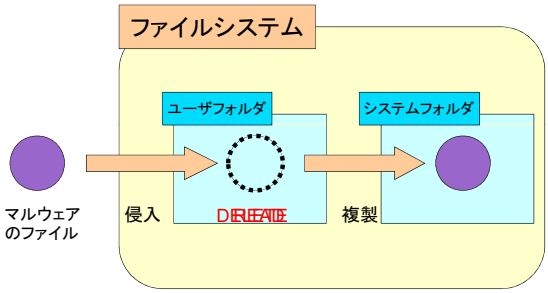
既存のビヘイビア検知手法(1

- 検知手法
  - 自己ファイルのREAD/DELETEをビヘイビアとして検知
    - マルウェアが侵入後、自身の存在を隠すためにシステムフォルダに移動する独自の挙動に基づく
- 検知精度
  - 検知率　約9割
    - 9検体の内、1検体のみ耐検知機能を有するマルウェアであったため検知できなかった

5

(1 酒井ら, 自己READ/DELETEの検出によるボット検知手法(MWS 2008)

マルウェアによるファイル操作の挙動



6

既存手法の問題点

- 正常なプログラムであってもマルウェアと判断する誤検知の発生
  - 例： アンインストーラが自身を削除するために自己をREADし、DELETEする挙動の誤検知

目的

アンインストーラを誤検知するのを解消する

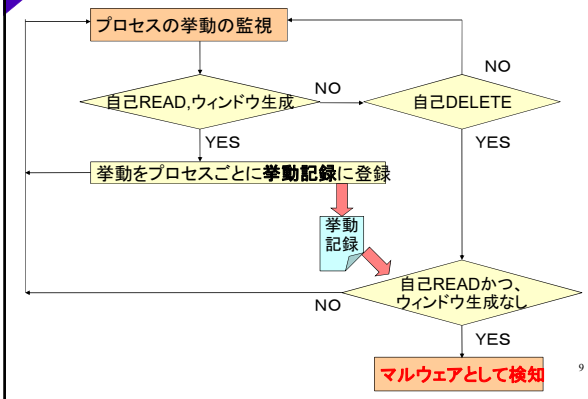
7

提案手法

- 基本的なアイデア
  - マルウェアの挙動だけでなく、アンインストーラの挙動にも着目した検知手法
    - マルウェアの挙動に見られず、アンインストーラに見られる挙動を定義
    - 当該挙動が見られた場合は検知対象から除外
- 着目したアンインストーラの挙動
  - GUIウィンドウの生成
    - 一般にアンインストーラはGUIインタフェースを持つ

8

提案手法による検知の流れ



9

システムの実装

- 対象OS
  - Windows XP
    - 既存研究での対象とされているOS
- 挙動の監視
  - Windows APIのフックによる監視
    - ファイルの読込み/削除→ ReadFile/DeleteFile API
    - ウィンドウの生成 → CreateWindow API

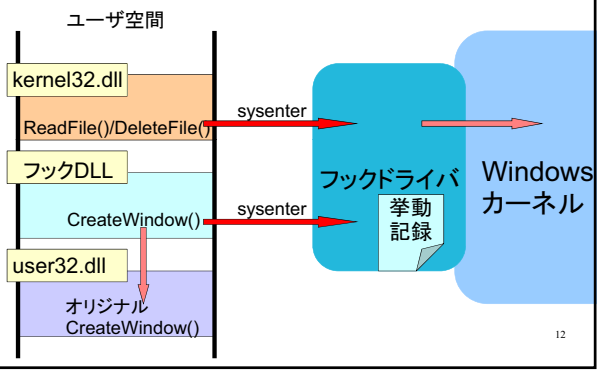
10

APIフック方法

- ReadFile/DeleteFile API
  - APIが使用するSysenter命令のフックにより実現
    - MSRレジスタを書換え、独自のフックドライバを経由させる
- CreateWindow API
  - DLL(user32.dll)の置換えにより実現
    - user32.dllと同じエクスポート関数を持つフック用のDLLを経由させる
      - 独自CreateWindow関数内でsysenter命令を発行し、フックドライバに呼ばれたことを通達する

11

APIフック後の処理の流れ



12

実験(予定)

- 実験目的
  - アンインストーラを誤検知しないかの検証
- 実験内容
  - マルウェア検体とアンインストーラに対する検知実験
    - マルウェアの検知率、アンインストーラの誤検知率の調査
- 実験環境
  - OS Windowx XP SP2

13

使用するマルウェア検体

- マルウェアの入手先
  - Cyber Clean Center
- マルウェア種別
  - トロイの木馬　9検体

14

安全性の考察

- マルウェアによる本提案システムの回避
  - マルウェアによるCreateWindowの発行により、本システムが回避される可能性がある
- 対抗策
  - ウィンドウの表示サイズ、表示される時間、表示される座標を確認することで対抗可能と考えられる
    - → CreateWindowが呼ばれた時に引数のチェックを行う

15

まとめ

- マルウェア検知における誤検知率の削減手法を提案した
  - アンインストーラの挙動に着目することで誤検知率を低下させる

今後の課題

- 本システムを用いた実験を行い、有効性を検証する

16

データフローを主体としたアクセス制御を実現する *DF-Salvia* の設計と開発

井田章三　6171090003-4　*sida@asl.cs.ritsumei.ac.jp*
立命館大学大学院理工学研究科　毛利研究室

1　はじめに

近年，計算機が普及し，プライバシデータはデジタルデータとして保存，管理されている．これによって，作業の効率化や企業の提供するサービスの品質向上につながったが，同時に記憶媒体やネットワークを通じて個人情報が流出する事件が頻繁に発生し，深刻な社会問題となっている．

文献 [1] では，2008 年に報道された個人情報漏洩事件の原因の調査について述べられており，「誤操作」，「管理ミス」，「紛失・置き忘れ」，「盗難」の比率が最も高いという結果が報告されている．具体的には，「誤操作」では顧客に他の個人の情報を電子メールに添付して送信してしまったという事例が多数報告されている．また，「管理ミス」，「紛失・置き忘れ」，「盗難」では，顧客などの個人情報を記憶媒体に保存し，外部に持ち出したことが原因となっている事例が多い．これらから，データ漏洩の多くが不当アクセスによるものではなく，正当なアクセス権限を持つユーザが原因となっていることが分かる．このようなデータ漏洩は，暗号化や認証などの外部からの攻撃を防ぐことを目的としたセキュリティ技術では防ぐことが難しい．

以上の背景より，我々はこれまで，正当なアクセス権限を持つユーザによる情報漏洩を防ぐことを目的としたオペレーティングシステム *Salvia*[2] の開発を行ってきた．*Salvia* は，プロセスによる情報漏洩の可能性がある計算機資源に対する操作を制御することにより，情報漏洩を防ぐ．すなわち，計算機資源に対する操作の制御をプロセス単位としている．本稿では，*Salvia* のアクセス制御の粒度を更に細かくし，データごとにより正確なアクセス制御を行うことを可能とする *DF-Salvia* を提案する．これにより，*Salvia* では発生する可能性のある保護データに対する過剰なアクセス制限や保護が不要なデータまで制限してしまうことを防ぐことが可能となり，ユーザビリティが向上する．

以降，本稿では，2 章で従来のアクセス制御手法について述べる．次に，3 章で *DF-Salvia* の概要について述べる．4 章で *DF-Salvia* の実装について述べる．

2　従来のアクセス制御手法

Salvia では，データ提供者の意図する保護方針をデータ保護ポリシ (以下，ポリシ) として定義し，保護対象のファイル (以下，保護ファイル) と組にして管理する．*Salvia* は，保護ファイルのデータ (以下，保護データ) を読み込んだプロセスに対し，ポリシに基づいたアクセス制御を課すことにより，プライバシを考慮したデータ保護を実現している．具体的には，read システムコールの前処理として対象の保護ファイルに設定されているポリシを読み出し，保護データを読み込んだプロセスを監視対象とする．監視対象となったプロセスでは，各計算機資源へのアクセスがポリシに基づき制限される．計算機資源へのアクセスは，システムコールによって行われ，システムコールの実行の可否をコンテキストとポリシから判断することでアクセス制御を実現している．

しかし，このアクセス制御手法では，過剰にアクセスを制限してしまう可能性がある．図 1 は，保護ファイルに対する read 後に他のファイルへ write を行う例である．書き込まれるデータに保護データが含まれているか否かに関わらず，その write システムコールは禁止されてしまう．

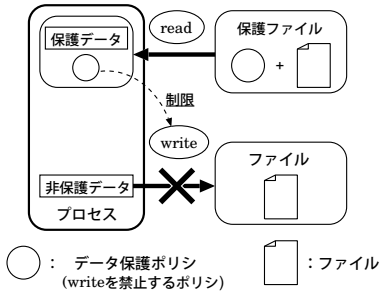


図 1　*Salvia* による過剰なアクセス制限

3　*DF-Salvia* のアクセス制御モデル

2 章で述べた課題を解決するためには，アクセス制御の主体であるプロセスを分割し，より小さいものにする必要がある．そこで，コンパイラと協調し，プロセスをデータフロー単位で管理・制御する *DF-Salvia* を提案する．以下，コンパイラのデータフロー解析と *DF-Salvia* のアクセス制御手法を述べる．

3.1　データフロー解析

データフロー解析は，コンパイラがコード最適化の際に行うプログラム中の変数定義の流れ (以下，データフロー) の解析である．*DF-Salvia* では定義-使用連鎖を作成するデータフロー解析を利用する．定義-使用連鎖は，変数の値を定義する命令文 (定義文) と，定義された変数がどの文で使用されるかを，その変数を使用する文 (使用文) の集合で表したものである．具体的には，ファイルからデータを読み込むライブラリ関数を定義文，そこで定義された変数を用いて計算機資源に書き込むライブラリ関数を使用文として定義-使用連鎖を作成する．保護データを読み込んだ定義-使用連鎖をアクセス制御の対象とすることで，計算機資源への書き込みをデータごとに制御可能となる．

3.2　アクセス制御手法

DF-Salvia のアクセス制御手法を図 2 に示す．*DF-Salvia* は，以下に示す手順でアクセス制御を行う (手順番号は，図 2 内の番号に対応している)．

1. コンパイラが生成したデータフロー解析結果に基づいて，データフロー ID を割り当てる．
2. read システムコールが発行された時，読み込み対象が保護ファイルなら，システムコールを発行したライブラリ関数が属するデータフロー ID (図 2 では ID は 1) に対して，対象保護ファイルのポリシを適用する．
3. write システムコールが発行された時，システムコールを発行したライブラリ関数が属するデータフロー ID に対して適用されたポリシの有無を確認する．
4. ポリシが適用されている場合は，そのポリシに従ってシステムコールの実行の可否を判断する．

OS が実行時に上記の処理をするには，手順 (2)，(3) におい

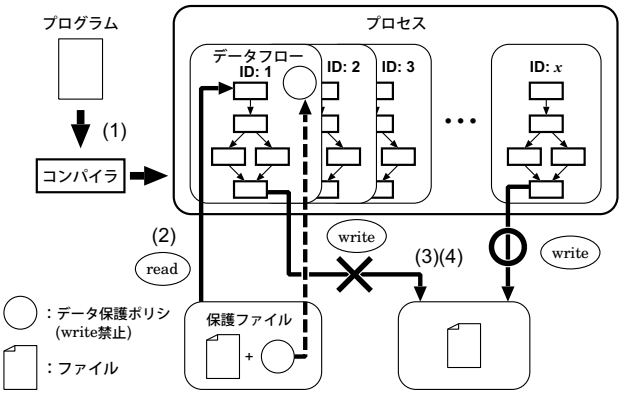


図 2　*DF-Salvia* のアクセス制御モデル

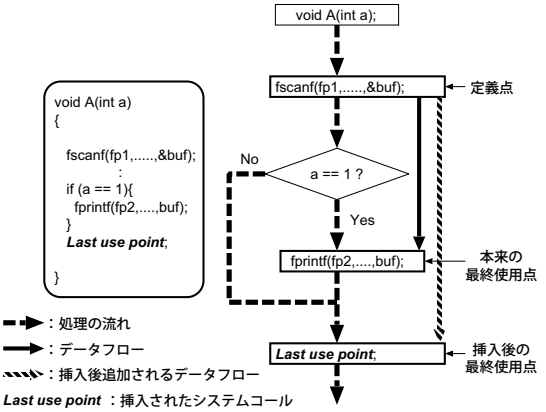


図 3　コンパイラによる最終使用点の挿入

て，システムコールを発行したライブラリ関数が属するデータフロー ID を特定できなければならない．これは，ライブラリ関数コールの命令アドレスからデータフロー ID を求めることができる．具体的には，データフロー ID 表を利用する．データフロー ID 表は，次の要素から成る．

- ユーザプログラムのライブラリ関数コール命令アドレス
- データフロー ID
- 最終使用点 (後述) かどうかを示すフラグ

システムコールが発行された際に，OS がライブラリ関数コールの命令アドレスを検出できれば，その命令アドレスとデータフロー ID 表に記録してある命令アドレスを比較し，そのシステムコールを発行したライブラリ関数が属するデータフロー ID を求めることができる．データフロー ID 表は，コンパイラとリンカが持つ情報から生成できる．OS 実行時のライブラリ関数コールの命令アドレスは，システムコール発行時にプロセスのスタックを解析することで求める．

3.3　データフロー ID の管理

■データ保護ポリシの削除　*Salvia* では，プロセスに適用されたポリシは，そのプロセス終了時に削除される．*DF-Salvia* では，データフローの最後の処理の終了時に削除する．ポリシは，それが適用されたデータフローのアクセス制御にのみに必要なものであるため，当該データフローの最後の処理が終了すれば，以降のアクセス制御に不必要となる．データフローの最後の処理を行う命令文を最終使用点と呼ぶ．以上をまとめると，最終使用点でアクセス制御を行った後，当該最終使用点が属するデータフローに適用されているポリシを削除する．

しかし，最終使用点が if 文やループ文に囲まれている場合，上記だけでは不十分である．最終使用点が if 文に囲まれている場合，最終使用点を通過しない分岐へ処理が進むと，ポリシの削除が行われない．最終使用点がループ文に囲まれている場合は，1 ループ目の最終使用点でポリシが削除され，2 ループ目以降の最終使用点実行時には，本来参照すべきポリシが存在しなくなり，正しいアクセス制御を行うことができない．

これらの問題を解決するには，if 文やループ文の終了直後に最終使用点をコンパイルの段階で埋め込むといった手法を用いることを考えている．具体的には if 文の場合，図 3 のように，if 文の終了直後に最終使用点を埋め込む．こうすると if 文による処理の分岐に関わらず必ず最終使用点を通過するため，fscanf で作成されたポリシの消去が可能となる．最終使用点は，プログラム中すべての if 文の終了直後に埋め込む必要がある．ループ文の場合も同様に問題を解決できる．

■識別番号の追加　*DF-Salvia* では，保護データが読み込まれると，ポリシはポリシ管理リスト (後述の Policy List) へ当該プロセス ID とデータフロー ID をキーとして登録され，アクセス制御時はこれらの値から適切なポリシが検索される．これにより，データフロー ID ごとに異なるポリシを適用可能となる．

しかし，同一データフロー ID の処理を区別してアクセス制御しなければならない場合がある．例えば，再帰関数内のデータフローは，最終使用点を通らずに再び同一データフローの定義点に到達する可能性がある．この場合，処理は独立しているがデータフロー ID は同一であるデータフローを区別して制御しなければならない．そこで，*DF-Salvia* は，データフロー ID に動的に識別番号を割り当てることとする．具体的には，データフローの定義点が実行される度に同一データフロー ID を識別するための番号 (データフロー ID x-1，x-2 の ”1”，”2”) を割り当て，別データフローとして OS に認識させることとする．

4　*DF-Salvia* の実装

DF-Salvia におけるデータフローに基づくアクセス制御を行うシステムの構成を図 4 に示す．プロセスがシステムコールを発行すると *DF-Salvia* が用意したアクセス制御を行う疑似システムコールである Alternative System Call Module に処理が移る．Alternative System Call Module は，システムコールの実行の可否を判定する際，History Repository で管理しているシステムコールの履歴情報を参照する．Policy List は，読み出したポリシを格納するリストである．Data Flow ID List は，データフロー ID 表を格納するリストである．Stack Backtracer は，ライブラリ関数コールの命令アドレスと Data Flow ID List を比較し，データフロー ID を求める．その際，History Repository からスタックのベースポインタのアドレスを指す EBP レジスタの値を取得し，スタック解析を行う．*DF-Salvia* は，保護データの read 時にポリシを読み出し，Stack Backtracer から取得したデータフロー ID を用いて Policy List に格納する．write システムコールが発行された場合は，Stack Backtracer から取得したデータフロー ID を用いて Policy List から適当なポリシを検索し，そのポリシに基づいて write システムコールの実行の可否を判断する．その際，write システムコールを発行したライブラリ関数が最終使用点ならば，アクセス制御に使用したポリシを Policy List の登録から削除する．

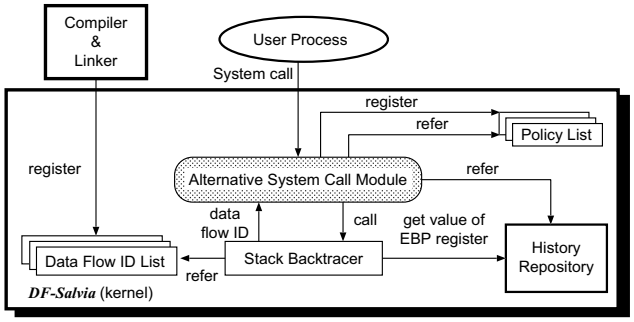


図 4　*DF-Salvia* のソフトウェア構成

5　おわりに

本稿では，従来の *Salvia* より粒度の細かいデータアクセス制御を可能とする *DF-Salvia* について述べた．*DF-Salvia* は，コンパイラのデータフロー解析を利用することにより，アクセス制御の主体をプロセスより小さい単位であるデータフローとし，保護すべきデータを正確に識別し制御することが可能となる．これにより，従来の *Salvia* で発生する可能性がある過剰なアクセス制限を防ぐことができる．今後は，コンパイラのデータフロー ID 表の自動生成機能と OS との連携部の実装を進める予定である．

参考文献

[1] NPO 日本ネットワークセキュリティ協会: “2008 年情報セキュリティインシデントに関する調査報告書 Ver.1.3,” http://www.jnsa.org/result/2008/surv/incident/2008incident_survey_v1.3.pdf, 2009.

[2] 鈴来 和久, 一柳 淑美, 毛利 公一, 大久保 英嗣: “Privacy-Aware OS *Salvia* におけるデータアクセス時のコンテキストに基づく適応的データ保護方式,” 情報処理学会論文誌: コンピューティングシステム, Vol.47, No.SIG3(ACS 13), pp.1-15, 2006.

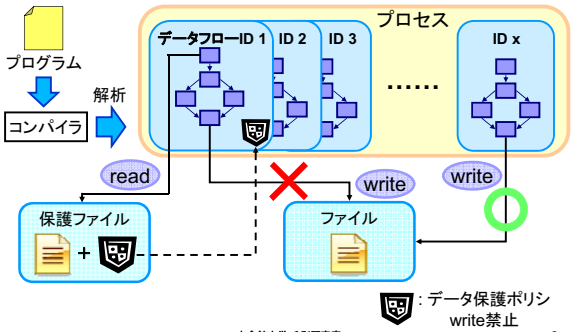
データフローを主体としたアクセス制御を実現するDF-Salviaの設計と開発

立命館大学大学院
毛利研究室
井田 章三

発表内容

- DF-Salviaの概要
- データ保護ポリシ管理手法
- 実装
- デモ
- おわりに

DF-Salviaの概要



DF-Salviaの課題

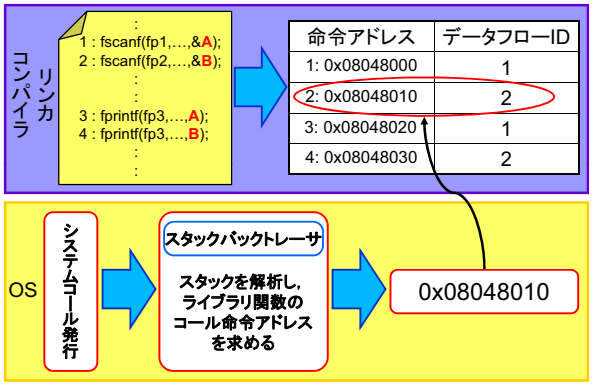
- 発行されたシステムコールが属するデータフローIDの動的な特定
- データフローIDに基づくデータ保護ポリシの管理
  - データ保護ポリシの削除
  - 処理が独立している同一データフローIDの識別

データフローIDの特定

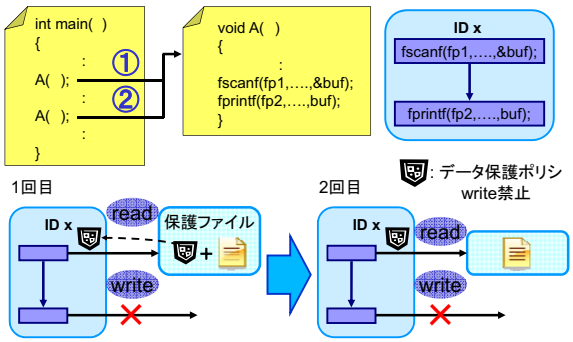
システムコールを発行したライブラリ関数コール命令のアドレスからデータフローIDを求める

- スタックバックトレーサ
  - プロセスのスタックを解析し，システムコールを発行したライブラリ関数コール命令のアドレスを求める
- データフローID表
  - ライブラリ関数コール命令のアドレスとデータフローIDを対応付けした表
  - コンパイラとリンカで事前に作成

データフローIDの特定の流れ

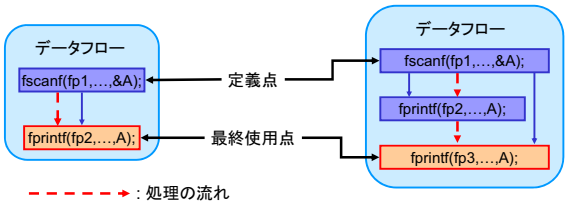


データ保護ポリシの管理

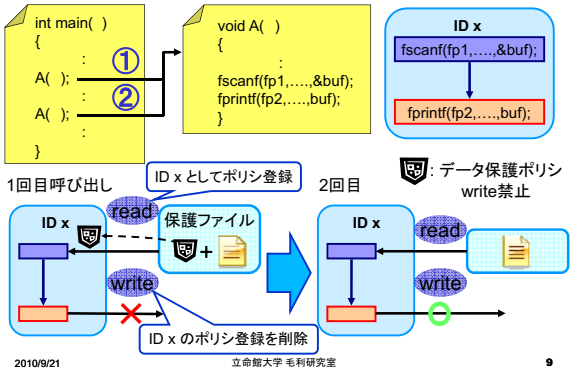


ポリシの削除(1/2)

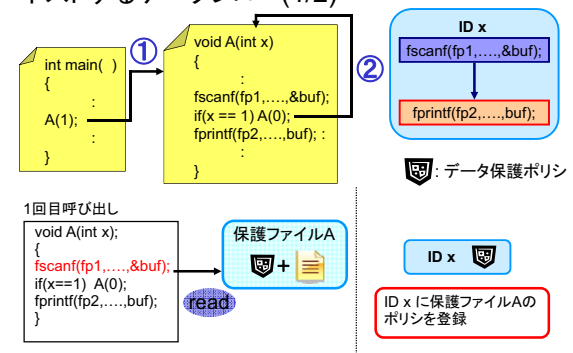
- データフローの最後の使用点のアクセス制御後は，そのデータフローのデータがプログラム中で使われることはない
- よって，データフローの最後の使用点を通過後，データ保護ポリシを削除
- このような使用点を最終使用点と呼ぶ



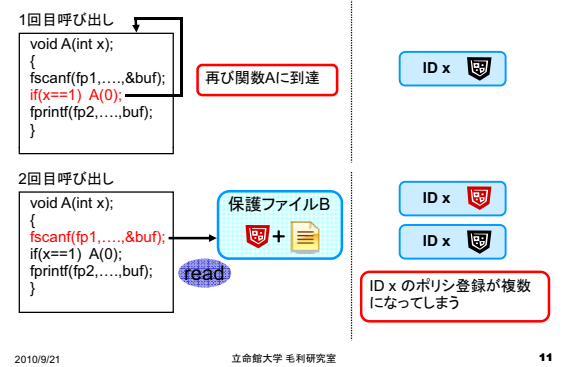
ポリシの削除(2/2)



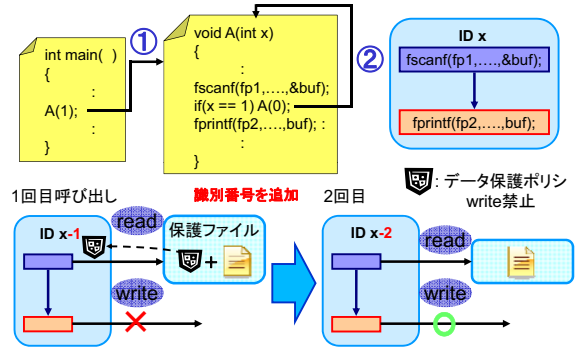
ネストするデータフロー(1/2)



ネストするデータフロー(2/2)

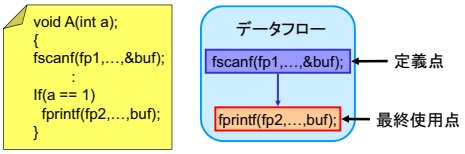


識別番号の追加



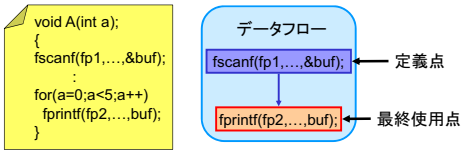
最終使用点の問題 (1/2)

- 最終使用点が if 文で囲まれている場合
  - 最終使用点を通らず，そのデータフローの処理をすべて終了する可能性
  - 最終使用点を通過しないので，ポリシの削除が行われず前述の問題が発生する



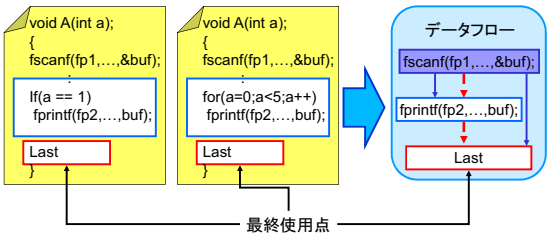
最終使用点の問題 (2/2)

- 最終使用点が ループで囲まれている場合
  - 最終使用点を通るが，そのデータフローの処理をすべて終了していない可能性
  - fprintfは5回発行されるが，1回目のfprintfで参照すべきポリシを削除してしまう

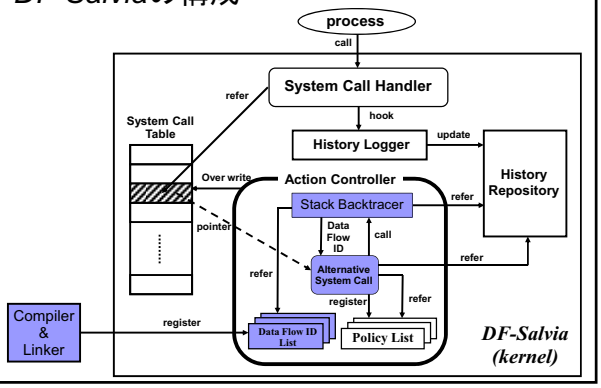


解決案

If 文とループ文の終了直後に最終使用点を埋め込む

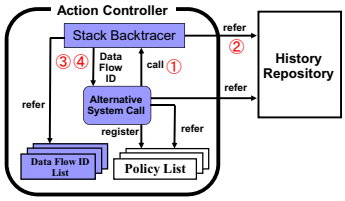


DF-Salviaの構成



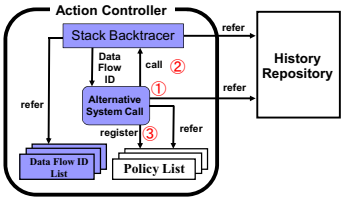
データフローIDの特定

1. Alternative System Call Module がStack Backtracer を呼び出す
2. Stack Backtracer は，History Repository からEBPレジスタの値を取得する
3. EBPレジスタの値を使い，プロセスのスタックを解析し，ライブラリ関数コールの命令アドレスを求める
4. 求めた命令アドレスとData Flow ID List に格納されているデータフローID表を比較してデータフローIDを求め，Alternative System Call Module に返す



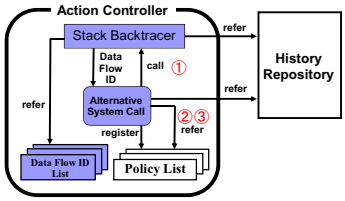
データ保護ポリシの登録(read)

1. readシステムコールが発行されると，対応したAlternative System Call Module がread実行の可否を対象保護ファイルのポリシから判断する
2. readシステムコールが実行された場合は，Stack Backtracer を呼び出し，データフローIDを求める
3. データフローIDを用いてPolicy Listに読み込んだポリシを格納する



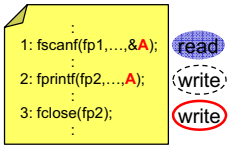
アクセス制限・ポリシ削除 (write)

1. writeシステムコールが発行されると，対応したAlternative System Call Module がStack Backtracer を呼び出し，データフローIDを求める．この際，最終使用点なのか調べる
2. データフローIDを用いてPolicy List からポリシを検索し，システムコールを制御する
3. 手順1において，最終使用点だった場合，手順2で参照したポリシをPolicy List から削除する



課題

- 入出力バッファリングによる使用点のズレ
  - writeシステムコールが発行されるのは，fcloseのバッファフラッシュ時
  - コンパイラが作成した定義-使用連鎖に含まれない命令文からwriteシステムコールが発行されることとなり，制御できない
  - fcloseも使用文に含める，バッファリングを切る，ライブラリ関数もデータフロー解析する，といった工夫が必要



定義文	使用文	ID
1: fscanf	2: fprintf	1

デモ

- ファイルコピープログラム
  ファイルAとファイルBの内容をファイルCに書き込む
  - SalviaとDF-Salviaで実験
    - 河島コンパイラで定義-使用連鎖を作成
    - 定義-使用連鎖とobjdumpの結果からデータフローID表を手書きで作成

おわりに

- *DF-Salvia*の概要
  データフローを主体としたアクセス制御
  - データフローIDの動的な特定
    - スタック解析とデータフローID表
  - データフローIDに基づくデータフ保護ポリシ管理
    - 最終使用点によるポリシ削除
    - 識別番号による処理が独立した同一データフローIDの区別
- 実装
  - スタックバックトレーサ
  - readによるポリシ登録・write時のアクセス制限，ポリシ削除

情報漏洩防止に着目したデータフロー解析を行うコンパイラの構築

河島 裕亮　6171090021-2　*ykawasima@asl.cs.ritsumei.ac.jp*
立命館大学大学院理工学研究科　毛利研究室

1　はじめに

近年, 計算機や情報通信技術の発達に伴い, プライバシ情報を電子データとして取り扱う事例が増加している. それに伴い, 電子化された顧客情報などのプライバシ情報が, データ保有者の意図に反して漏洩する事件が多発している. 文献 [1] では, 情報漏洩を引き起こす要因として, プライバシ情報に対する正当なアクセス権限を有する者の管理ミスによる流出や紛失, 外部への持出し, アプリケーションの誤操作, 盗難が挙げられる. 特に, 正当なアクセス権限を有する者による情報漏洩が発生原因の多くを占めることを示している. しかし, 暗号化や認証といった既存のセキュリティ技術は, 外部からの攻撃を防止することを目的とするため, 人為的なミスや内部犯による情報漏洩を防止することは困難である.

以上の背景より, これまで我々は, 上記のような人為的ミスや内部犯による情報漏洩を防止することを目的としたオペレーティングシステム Privacy-aware OS *Salvia*[2] の開発を行ってきた. *Salvia* では, 保護すべきデータを有するファイル (以下, 保護ファイルと記す) に対して, そのデータの保護方針を定義した保護ポリシを設定する. *Salvia* は, 保護ファイルを読込んだプロセスに対して保護ポリシの内容に応じたアクセス制御を課すことでプライバシ情報を保護する.

本稿では, Salvia のアクセス制御の粒度をより細かくすることで, データごとのより正確なアクセス制御を実現する *DF-Salvia*[3] について述べる. *DF-Salvia* は, アクセス制御を課す単位をデータフローとすることで, プロセス内で計算機資源に書き込むデータとデータの元となったファイルの対応付けが明確になるため, 制御すべき処理をより正確に識別することが可能となる. *DF-Salvia* によるアクセス制御を実現するためには, プロセス内に存在するデータフロー情報を得る必要がある. そのため, コンパイラが監視対象プロセス内で実行されているプログラムのデータフローを予め解析し, *DF-Salvia* に解析結果を提供する. そのため, *DF-Salvia* によるアクセス制御を実現するためには, データフロー情報を解析するコンパイラと解析結果に基づいてアクセス制御を課す制御機構の 2 つの機構が必要となる.

以降, 本稿では, 2 章でデータフロー解析および *DF-Salvia* によるアクセス制御手法の概要について述べる. また, 3 章で COINS コンパイラを用いたプロトタイプ実装について述べ, 4 章で今後の課題を述べ, 本稿をまとめる.

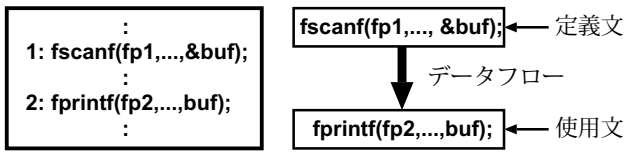


図 1　定義使用連鎖

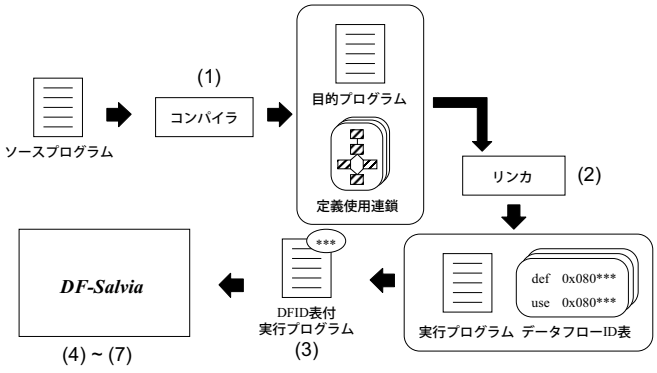


図 2　DFID 表付き実行プログラム生成過程

2　*DF-Salvia*

2.1　データフロー解析

DF-Salvia では, データフローを主体としたアクセス制御方式を実現するために, 予めプログラム中のデータフローを把握する必要がある. このデータフローは, コンパイラを用いてソースプログラムを解析することで求めることが可能となる. コンパイラが解析するデータフローは様々だが, *DF-Salvia* では. 定義使用連鎖を生成するデータフロー解析を用いる.

定義使用連鎖とは, 変数に値を定義する命令文 (定義文) とその変数を使用する命令文 (使用文) の集合で表したものである (図 1). 定義使用連鎖を解析することで, 保護ファイルから読み出されたデータが, どの命令で使用されるかを把握することが可能となる. すなわち, 保護すべきデータフローを把握することで, プロセスに対して過剰なアクセス制御を課すことなく, データを保護することが可能となる.

2.2　アクセス制御手法

DF-Salvia では, アクセス制御の対象をデータフローとするために, コンパイラを用いてプログラム中のデータフローの有無を解析する. *DF-Salvia* では, コンパイラ

によるデータフロー解析から実行プログラム生成までを, 図 2 に示す手順で処理を行う. また, *DF-Salvia* によるアクセス制御の手順を以下に示す (以下の手順番号は, 図 2 中の番号にしている).

1. ソースプログラムをコンパイルし, 定義使用連鎖を生成する.
2. リンカが持つ情報をもとにして定義使用連鎖と各連鎖に関連するライブラリ関数コールの命令アドレスを対応付ける (データフロー ID 表の生成).
3. データフロー ID 表と実行プログラムを一組にして管理する.
4. 手順 (3) で生成されたプログラムが実行された際, データフロー ID 表を読み込み, 各データフローに対して ID を割り当てる.
5. read システムコールによって, 保護ファイルからデータが読み出された場合, システムコールを発行したライブラリ関数が属するデータフロー ID に対して読み出し元保護ファイルに対応する保護ポリシを適用する.
6. write システムコールが発行された際, システムコールを発行したライブラリ関数が属するデータフローにポリシが適用されているか否かを確認する.
7. ポリシが登録されている場合, ポリシの内容に応じてシステムコールの実行可否を判断する.

OS が実行時にアクセス制御を課すためには, 手順 (5), (6) において各システムコールで使用されている変数が属するデータフロー ID を特定する必要がある. これは, システムコールが発行された時点の命令アドレスを起点としてスタックを解析することで, そのシステムコールを呼び出したライブラリ関数コールの命令アドレスを検出することでデータフロー ID を求める. そのために, 手順 2 において, リンカが持つ情報を元にして定義使用連鎖と命令アドレスの対応付けを行う.

3 実装

DF-Salvia で用いる定義使用連鎖を生成するために, プロトタイプとして COINS[4] を用いた. COINS は, 新しいコンパイラ方式を作成し, 実験・評価を容易に実行可能にする並列化コンパイラ向け共通インフラストラクチャである. COINS は, ソースコードを高水準中間表現 (HIR) から低水準中間表現 (LIR) に変換し, LIR を基にして目的コードを生成する. また, COINS は, HIR または LIR を用いて最適化やデータフロー解析を行うことも可能である. プロトタイプでは, HIR を用いて定義使用連鎖を生成した.

本研究では, 情報漏洩防止に着目したデータフローを生成することを目的とするため, ファイルから読み出したデータを変数に格納する点を定義点の一種とした. つまり, fscanf() や fgets() のようなファイルからデータを読み出すために用いられるライブラリ関数コールを定義点として検出するよう実装した (図 3).

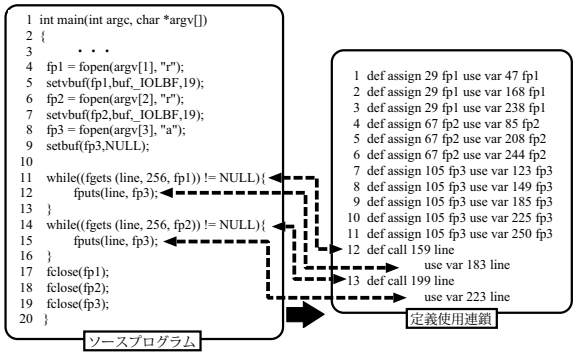


図 3 プロトタイプを用いて生成した定義使用連鎖

今後は, 生成した定義使用連鎖とリンカが持つ情報をもとにして, 定義使用連鎖とライブラリ関数コールの命令アドレスを対応付ける機能と生成したデータフロー ID 表と実行プログラムを一組で管理するための機能を設計・実装する必要がある.

4 おわりに

本稿では, データフローを単位としてアクセス制御を課す *DF-Salvia* で用いるコンパイラについて述べた. 今後は, 定義使用連鎖とライブラリ関数コールの命令アドレス対応付けの自動化と, 生成したデータフロー ID 表と実行プログラムを一組で管理するための手法について検討する.

参考文献

[1] NPO 日本ネットワークセキュリティ協会: JNSA 2007 年情報セキュリティインシデントに関する調査報告書, http://www.jnsa.org/result/2007/pol /incident/2007incidentsurvey_v1.32.pdf, 2008.

[2] 鈴来和久, 一柳淑美, 毛利公一, 大久保英嗣: “Privacy-Aware OS *Salvia* におけるデータアクセス時のコンテキストに基づく適応的データ保護方式,” 情報処理学会論文誌, コンピューティングシステム (ACS 13), Vol. 47, No. SIG3, pp. 1–15, 2006.

[3] 井田章三, 岩永真幸, 毛利公一: “Privacy-aware OS Salvia におけるデータフローを主体としたアクセス制御手法,” 第 71 回全国大会講演論文集, Vol. 3, pp. 353–354, 情報処理学会, 2009.

[4] 中田育男, 渡邉坦, 佐々政孝, 森公一郎, 阿部正佳: “COINS コンパイラ・インフラストラクチャの開発,” コンピュータソフトウェア, Vol. 25, No. 1, pp. 2–18, 日本ソフトウェア科学会, 2008.

情報漏洩防止に着目したデータフロー解析を行うコンパイラの構築

立命館大学大学院
毛利研究室
河島　裕亮

発表内容

- はじめに
- Privacy-aware OS *Salvia*
- *DF-Salvia*
  - データフロー解析
  - データフローID表
- 実装
- 実験
- おわりに

はじめに

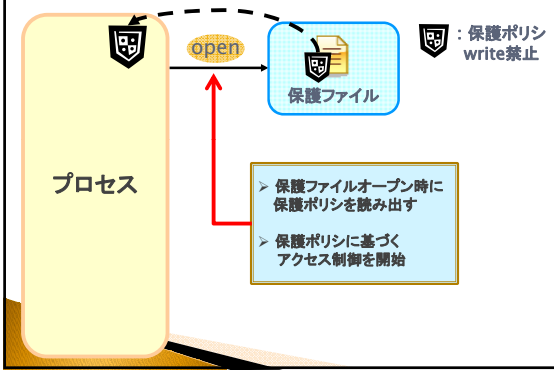
- 近年，機密情報が漏洩する事件が多発している
  - 計算機，情報通信技術の発達伴い，
    電子化された機密情報の漏洩頻度，規模が拡大
- 情報漏洩事件のおもな原因
  1. 正当な利用者の不注意による流出や紛失
  2. 正当な利用者による機密情報の持出し
  3. アプリケーションの誤操作
  4. 盗難

Privacy-aware OS *Salvia* の開発

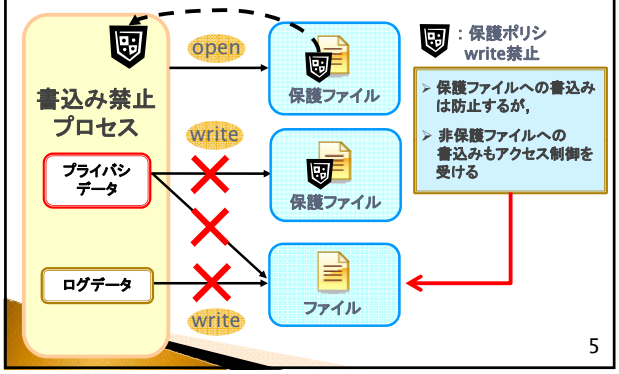
Privacy-aware OS *Salvia* の概要

- アクセス制御機構を備えたOS
  - アプリケーションの信頼度に関わらず統一的に制御可能
- 保護方式
  - 制御対象：プロセス
    - プロセスがデータを漏洩させる動作を制御
    - プロセスが実行するシステムコールをチェックすることで制御
  - 保護単位：ファイル
    - 保護したいファイルと保護ポリシを一組にして管理
  - コンテキストの利用
    - ユーザや計算機の状況のこと
    - ユーザ，時間，計算機の場所，IPアドレス　など

*Salvia*のアクセス制御方式



*Salvia*のアクセス制御方式



原因

6

DF-Salvia の概要

- アクセス制御機構を備えたOS
  - Privacy-aware OS *Salvia* を基にしたアクセス制御機構
  - アプリケーションの信頼度に関わらず統一的に制御可能
  - データフローを対象にアクセス制御を課す
- 保護方式
  - 制御対象：データフロー
    - コンパイラのデータフロー解析によって事前に求める
    - 制御対象システムコールの発行時，対応するデータフローと保護ポリシに基づいてアクセス制御を課す
  - 保護単位：ファイル
    - 保護したいファイルと保護ポリシを一組にして管理
  - コンテキストの利用
    - ユーザや計算機の状況のこと
    - ユーザ，時間，計算機の場所，IPアドレス　など

7

データフロー解析

- プログラム中の様々な位置で、取りうる値の集合を解析する技法
- 定義使用連鎖を求めるデータフロー解析を用いる
  - 変数に値を定義する命令文(定義文)と定義された変数が使用される文(使用文)の集合
- 代表的な定義文

```
a = 1;                  // aの定義文
scanf("%d", &b);        // bの定義文
```

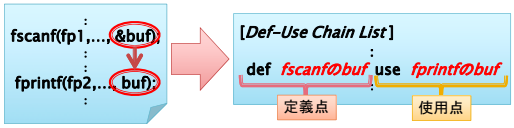
- 代表的な使用文

```
b = a + 1;              // aの使用文
                        (bの定義文)
printf("%d", b);        // bの使用文
```

8

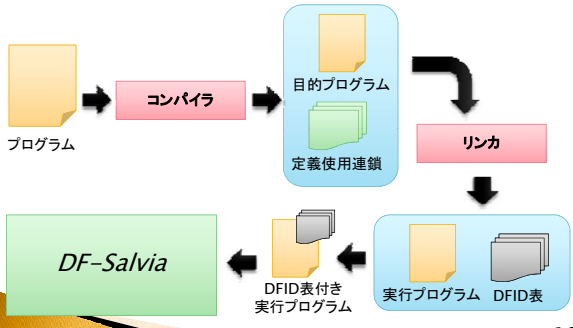
定義使用連鎖

- 保護ファイルのデータ読込み以降，データが漏洩する可能性があることに着目
- ファイルからデータを読込むためにreadを発行するライブラリ関数コールから派生する定義使用連鎖を求める



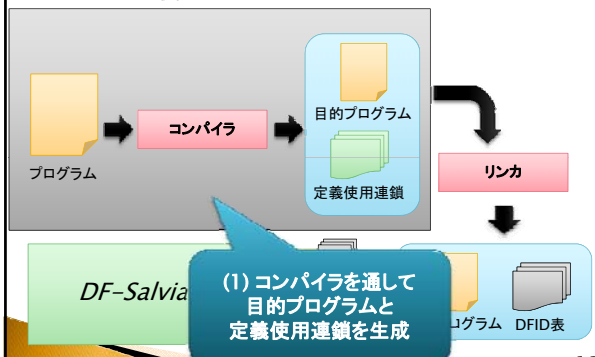
9

処理手順

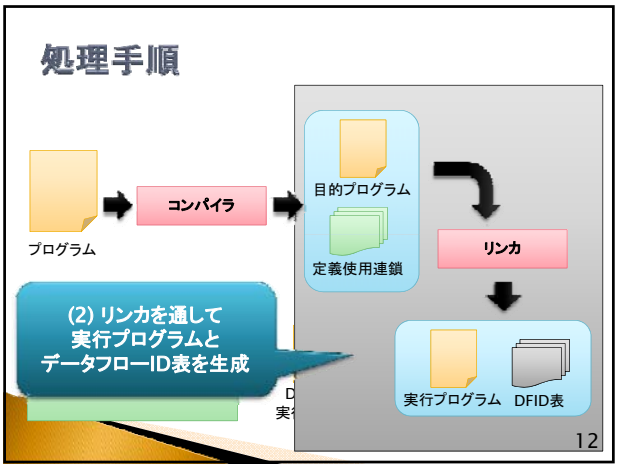


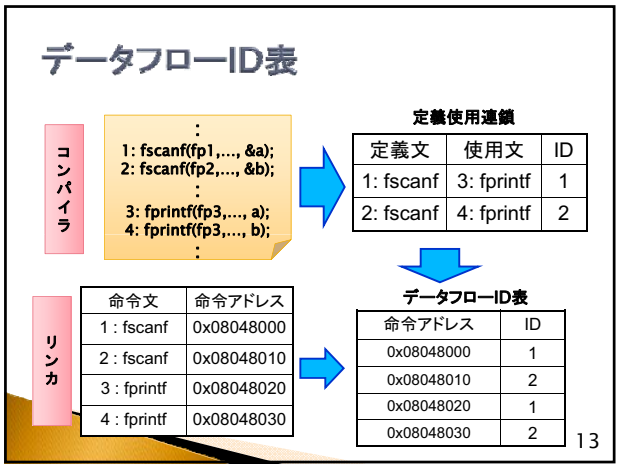
10

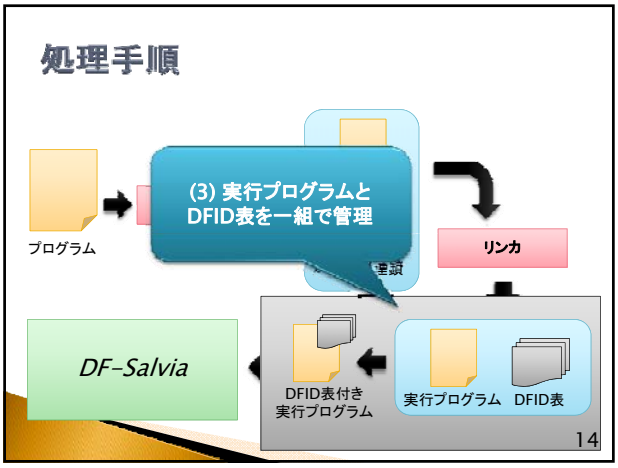
処理手順

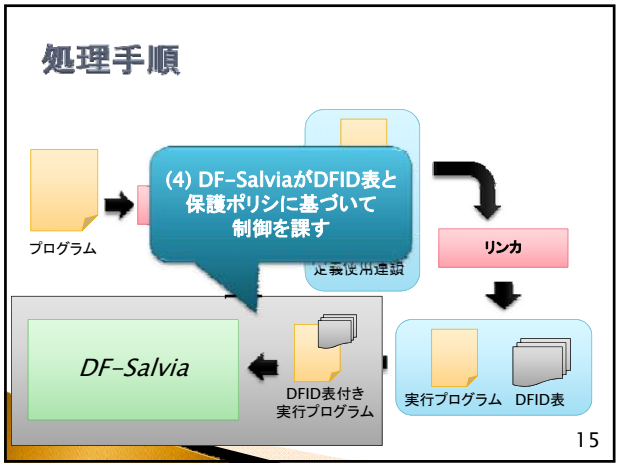


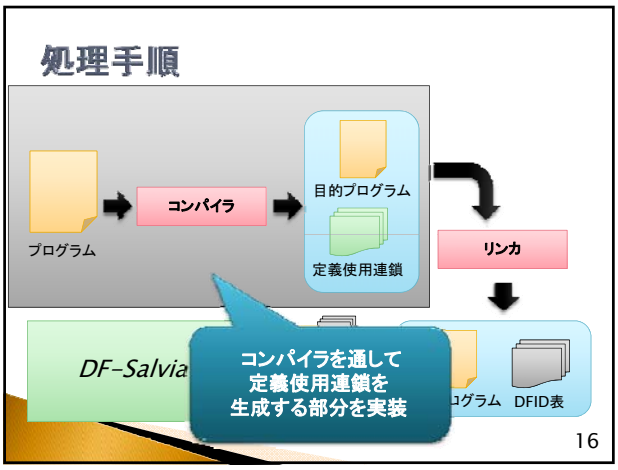
11









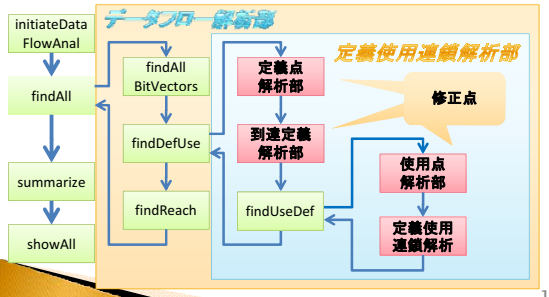


実装(1/2)

- プロトタイプとしてCOINSコンパイラをもとに実装中
- COINSで不完全な部分
  - ポインタや配列における定義使用連鎖解析
    - 定義点–使用点の関連付け
    - ブロック間解析
      - ⇒ 定義点解析部，到達定義解析部，使用点解析部を修正することで解決
  - 引数を定義点とするライブラリ関数の解析
    - fgets(buf) ⇒ fputs(buf) : buf からbufへのフロー
      - ⇒ 引数を定義点とするようなライブラリ関数をCOINSに登録することで解決

17

実装(2/2)



実験

- コンパイラで求めたデータフロー情報をDF-Salviaに登録することで適切な制御が行われるかを確認する
- 簡単なコピープログラムを対象に実験
  - 2種類のファイルを他のファイルに書き込むプログラム
  - 保護ファイルと非保護ファイルの2種類を書き込む
    - 保護ファイル：保護ポリシに基づいてアクセス制御が課される
    - 非保護ファイル：通常の処理が実行される
- データフローID表の登録は未完成のため，手動で登録

実験(定義使用連鎖生成)

ソースプログラム

```
int main(int argc, char *argv[])
{
        fp1 = fopen(argv[1], "r");
        fp2 = fopen(argv[2], "r");
        fp3 = fopen(argv[3], "a");
    while((
        fgets (line, 256, fp1))
        != NULL)
    { fputs(line, fp3); }
    while((
        fgets (line, 256, fp2))
        != NULL)
    { fputs(line, fp3); }
        fclose(fp1);
        fclose(fp2);
        fclose(fp3);
}
```

定義使用連鎖

def *call 159 line* (fgetsのline)
use *var 183 line* (fputsのline)

def *call 199 line* (fgetsのline)
use *var 223 line* (fputsのline)

- 定義使用連鎖は，COINSの中間表現を介して生成する
- def *** use *** で一つの連鎖を表現
- def *** ：定義点
- use *** ：使用点

実験(データフローID表生成)

定義使用連鎖

def *call 159 line* (fgetsのline)
use *var 183 line* (fputsのline)

def *call 199 line* (fgetsのline)
use *var 223 line* (fputsのline)

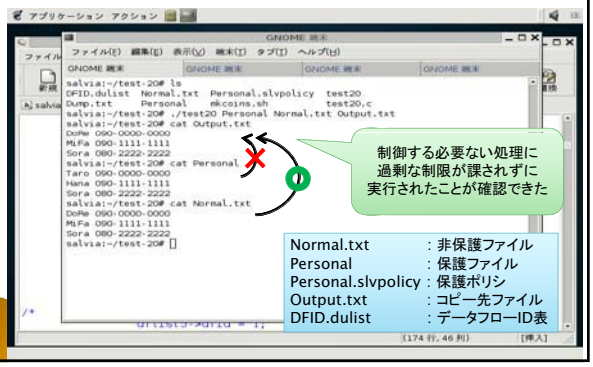
リンカ

- 定義使用連鎖とリンク後の情報をもとに，DFID表を生成する
- DFID表の情報を*DF-Salvia* に登録する

データフローID表

LibFnc	address	DFID
fgets	0x0804861d	1
fputs	0x08048638	1
fgets	0x08048658	2
fputs	0x08048673	2

実験(実行結果)



今後の課題

- 実装済み
  - 定義使用連鎖の自動生成
- 課題
  - 定義使用連鎖とライブラリ関数コールの命令アドレス対応付けの自動化
  - 実行プログラムへのデータフローID表の埋込み
    - ELFファイルにひとつのセクションとして組み込む方向で検討中

おわりに

- 情報漏洩防止に着目したデータフロー解析を行うコンパイラの構築
  - read, write を発行するライブラリ関数に着目して定義使用連鎖を解析
  - リンカの持つ情報と定義使用連鎖をもとにデータフローID表を生成し, 実行プログラムと一組にして管理
- 今後の課題
  - 定義使用連鎖とライブラリ関数コールの命令アドレス対応付けの自動化
  - データフローID表をELFヘッダへの追加
  - Low Level Virtual Machineコンパイラの調査

24

完全仮想化環境における高信頼性タイマの実現

若林 大晃† 毛利 公一†

†立命館大学情報理工学部

1 はじめに

近年，組込みシステムの分野では，リアルタイム性や信頼性だけでなく，GUI やマルチメディア処理といった高い機能性が求められている．リアルタイム性を求めるシステムでは，μITRON や VxWorks などに代表されるリアルタイム OS（以下，RT-OS と記す）が利用され，これらの RT-OS では，機器制御を主要とした必要最小限の機能提供によって，リアルタイム性と信頼性を保証している．また，高機能性を求めるシステムでは，Windows や Linux などの高機能 OS が利用される．これらの OS は，高度な情報処理を主としており，複雑なシステム構成により実現されている．リアルタイム性・信頼性の保証と，高機能性の実現という異なる特性をもつ要求を同時に実現する方法として，仮想計算機モニタを利用することが考えられる．

仮想計算機上で RT-OS を動作させる場合の課題として，リアルタイム性の保証があげられる．仮想計算機上で動作させる OS（以下，ゲスト OS と記す）のタイマ管理は，リアルタイム性の保証に重要な要素である．しかし，仮想計算機では，タイマ管理に用いられるタイマデバイスが仮想化されており，他のゲスト OS や仮想計算機モニタの動作に影響を受ける．このため，仮想タイマデバイスが生成する仮想タイマ割込みの周期にジッタが発生し，ゲスト OS 内のタイマの制度が低くなるという問題がある．

本稿では，完全仮想化環境において，一定の周期でゲスト OS に仮想タイマ割込みを配送する手法について述べる．

2 完全仮想化

完全仮想化の利点は，ゲスト OS の修正が不要なことである．完全仮想化の仮想計算機モニタとして，Linux と KVM [1] を用いている．KVM（Kernel-based Virtual Machine）は Linux を仮想計算機モニタとして利用するための，Linux カーネルモジュールである．仮想計算機を動作させるには，この他に，ハードウェアのエミュレートを行うソフトウェアが必要でり，QEMU [2] を用いている．

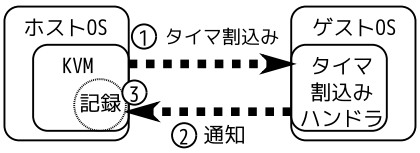


図 1: ジッタの測定方法

2.1 KVM と QEMU の関係

KVM は，ユーザアプリケーションに仮想計算機を資源として提供する．ユーザアプリケーションである QEMU は，システムコールを用いて KVM に仮想計算機の操作を依頼する．KVM は，移行したカーネルコンテキストで処理を行う．QEMU が KVM に仮想マシンの実行を依頼するため，QEMU がディスパッチされていることが，ゲスト OS の動作に必要な条件となっている．QEMU は他のユーザプロセスと同列であるため，他のプロセスと同じようにスケジュールされる．

3 現状の問題点

3.1 ジッタの測定方法

タイマの性能の指針として，ゲスト OS に通知されたタイマ割込みのジッタを使用する．ジッタの測定方法を，図 1 に示す．あらかじめ，ゲスト OS のタイマ割込みハンドラを修正し，KVM に割込みを受け取ったことを通知するようにする．通知を受けた KVM は，その時刻を記録し，1 つ前の時刻との差分をとり，ジッタとする．

3.2 問題点

表 1 に示す環境でゲスト OS を動作させたときのジッタを，図 2 に示す．ジッタは，負荷をかけた場合と，かけていない場合を測定した．負荷は，ゲスト OS から HDD に対して書き込みを行うものである．表 1 の環境

Development of high reliability timer on full virtualization

Hiroaki WAKABAYASHI†and Koichi MOURI†

†Department of Computer Science, Ritsumeikan University 525-8577, Shiga, Japan

表 1: 実行環境

環境	項目	内容
ホスト	CPU	Intel Core i7 920 2.67GHz
	メモリ	6GB
	OS	Linux 2.6.33.3
ゲスト	メモリ	1GB
	OS	Linux 2.6.34-rc5

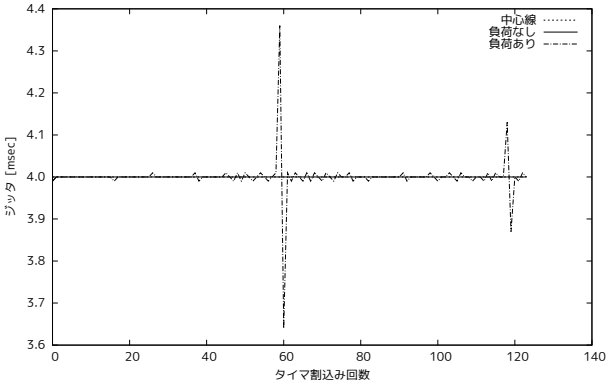


図 2: 問題のある環境でのジッタの測定

におけるタイマ割込み周期は 4ms であるため，4.0 の中心線に近い方がジッタが少なく，よいタイマといえる．図 2 のグラフから，負荷をかけた場合には，最大で約 4.35msec のジッタが発生していることがわかる．

ジッタの発生は，KVM の仕組みによるところが大きい．KVM によるタイマ割込みの配送の流れを，図 3 に示す．KVM は 4ms ごとにカーネル内部のタイマをセットし，タイマが起動したときにタイマ割込みを配送する．タイマが起動すると，KVM はそのことを記憶し，仮想計算機の実行が再開されたときに，割込みを配送する．仮想計算機の再開は，ユーザプロセスである QEMU が行うため，QEMU がディスパッチされなければ，仮想計算機の実行を再開できず，タイマ割込みを配送できない．

4 改善手法

本章では，タイマ割込みを一定の周期で送るための，3 つの改善手法について述べる．

4.1 CPU 占有方式

CPU 占有方式は，QEMU のプロセスに CPU を独占させることで，タイマ割込みの配送時に，ただちにディスパッチされるようにする手法である．この手法を適用したときのジッタを図 4 に示す．手法適用前の図 2 と比較すると，ジッタが減少していることがわかる．

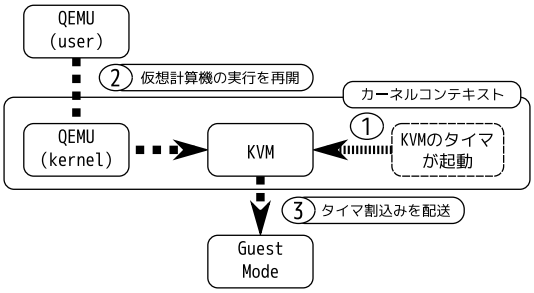


図 3: タイマ割込みの配送の流れ

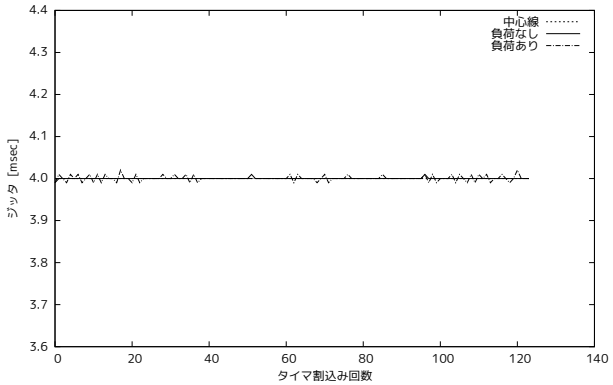


図 4: CPU 占有方式でのジッタの測定

4.2 ディスパッチ方式

ディスパッチ方式では，CPU を占有させる代わりに，KVM から QEMU をスケジュールする．タイマ割込みを配送する際に，QEMU がただちにディスパッチされるようにすることで，ジッタを抑える手法である．

4.3 ショートカット方式

ショートカット方式は，タイマ割込みの配送に，QEMU を介さない手法である．前述の 2 つの手法と比べると，実現が難しいという問題がある．

5 おわりに

本稿では，完全仮想化環境における高信頼性タイマの実現において，現状の問題点とその改善手法について述べた．現状では 3 つの改善手法のうち 1 つのみ評価を行えている．今後は，ディスパッチ方式の評価を行い，また，ショートカット方式の有効性を検証し，有効と考えられれば実装および評価を行いたい．

参考文献

[1] KVM: http://www.linux-kvm.org/page/Main_Page, 2010.

[2] QEMU: http://wiki.qemu.org/Main_Page, 2010.

完全仮想化環境における高信頼性タイマの実現

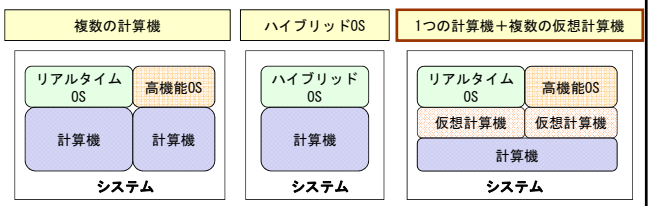
立命館大学　情報理工学部
毛利研究室
若林 大晃

発表内容

- はじめに
- KVMとQEMUとは
- Jitterの測定方法
- 現在のタイマの問題点とその原因
- 改善案
- おわりに

はじめに (1/2)

- 組込みシステムにおいて、信頼性と機能性を両立するため、
  - 複数の計算機を用いて、リアルタイム用とその他用に分ける
  - ハイブリッドOSを用いて1つのハードウェアで実現する
  - ひとつの計算機上に複数の仮想計算機を作成する

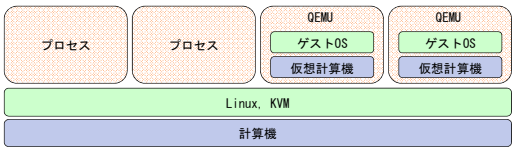


はじめに (2/2)

- リアルタイム仮想マシンモニタ
  - リアルタイム性を保障する必要がある
  - 時間を計るためのタイマデバイスが仮想化されている
  - タイマの制度が落ちる可能性
    - タイマ割込みの周期に揺らぎが発生
- 高信頼性のタイマの実現をめざす
  - ゲストOSにタイマ割込みを 遅れず／早まらず 通知する
  - ゲストOSに修正の必要がない、完全仮想化で行う
  - Linux, KVM, QEMUを使用

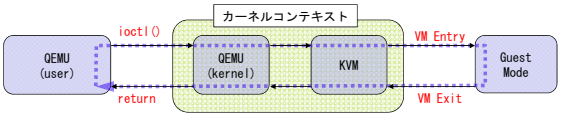
KVMとQEMUとは

- KVM (Kernel-based Virtual Machine)
  - Linuxを仮想マシンモニタにする，カーネルモジュール
  - 仮想計算機とその上で動作するソフトウェアを，１つのプロセスとして抽象化して扱う
  - 機能
    - 仮想計算機の作成
    - ゲストOSをCPUで直接実行
- QEMU
  - BIOSやハードウェアのエミュレータ



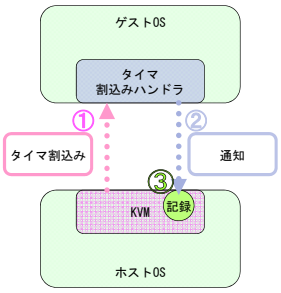
QEMUとKVMの関係

- QEMUが仮想計算機の状態を設定し，実行を開始する
  - QEMU(user)：初期化など，準備
  - KVM：CPUで直接ゲストOSを実行（ioctlシステムコールより）
  - KVM：CPUで直接実行できない命令に到達（→ioctlシステムコール終了）
  - QEMU(user)：エミュレーション
  - QEMU(user)：実行再開
- QEMUは普通のプロセスと同じようにスケジュールされる
  - ディスパッチされないと，ゲストOSが動作できない



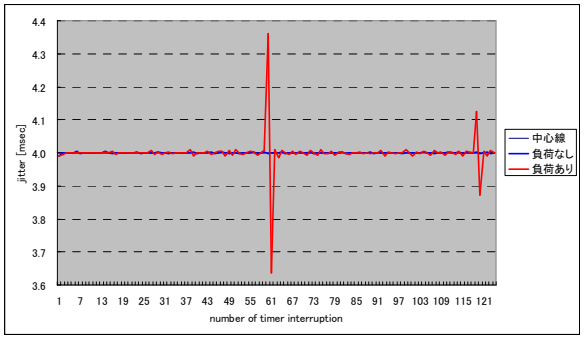
Jitterの測定方法

- タイマ割込みの間隔を測定
  - ゲストのタイマ割込みハンドラから、ホストへ通知し、ホストでこの時刻を記録
  - 前回の割込み時刻から、何秒後に割込みがきたかをグラフ化
  - タイマの性能の指針に
- 環境
  - ホスト
    - CPU: Intel Core i7 920 2.67GHz
    - メモリ: 6GB
    - OS: Linux 2.6.33.3
  - ゲスト
    - CPU: 仮想CPUが1つ
    - メモリ: 1GB
    - OS: Linux 2.6.34-rc5



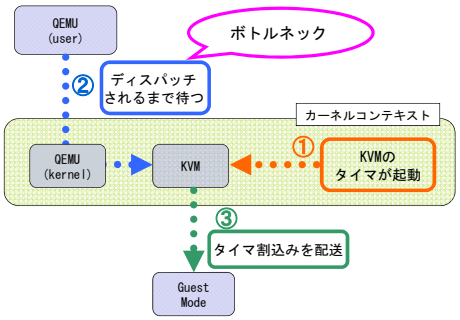
現状の問題点

- 負荷をかけると、タイマ割込みが遅れる場合がある
- 負荷：ゲストOSでHDDに書き込み



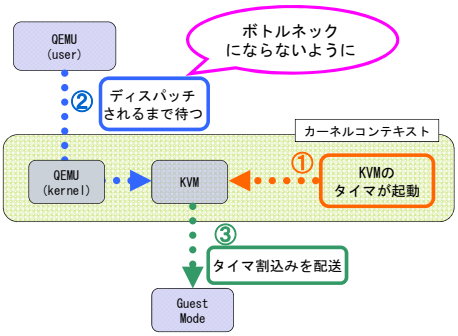
現在のタイマ割込みの通知方法

- ゲストOSはQEMUがディスパッチされないと，動作できない



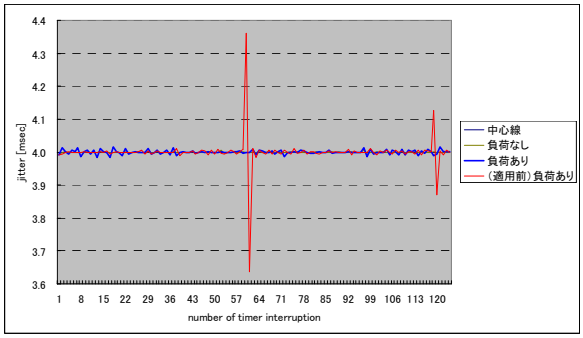
改善案その1（CPU占有方式）

- ゲストOSを動かすQEMUが，すぐにディスパッチされるように，CPUを占有させる



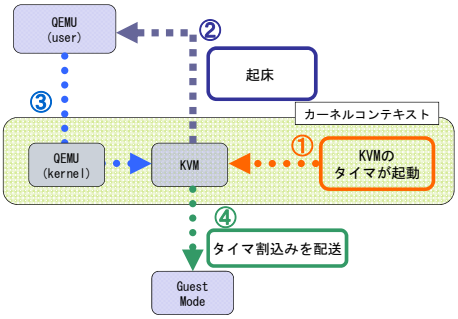
改善案1の結果

- 負荷をかけても、大きな遅延はみられない



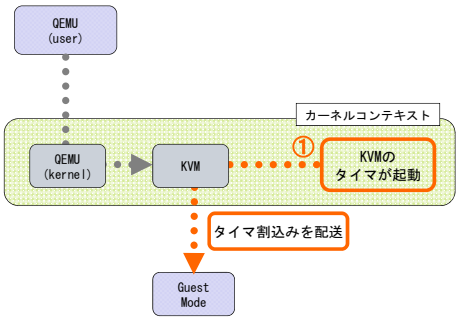
改善案その2（ディスパッチ方式）

- タイマ割込みを送りたいときに，KVMからQEMUがすぐにディスパッチされるようにスケジュールする



改善案その3（ショートカット方式）

- 難点：本来の遷移とは異なる遷移が必要



おわりに

- 完全仮想化環境でリアルタイムOSを動作させる
  - 計算機の数が少なくて済み、動作させるリアルタイムOSも変更なしで利用可能なため、コストパフォーマンスがよい
- 高信頼性のタイマの実現手法
  - 現状：KVMの仕組み上、ユーザ空間とカーネルの繋がりがボトルネックになっている
  - 改善案1：CPUを占有させ，ゲストOSを常に動作させる
  - 改善案2：割込み配送時に，ただちにディスパッチされるようにする
  - 改善案3：KVM内部で割込み配送処理を完結する
- 今後の予定
  - 改善案1（CPU占有方式）を評価
  - 改善案2（プリエンプション方式）の実装・評価
  - 改善案3（ショートカット方式）の検証

ご静聴ありがとうございました

リアルタイム仮想計算機モニタ Natsume における割込み通知モデルの提案

渡邉 和樹 †　　毛利 公一 ††
† 立命館大学大学院理工学研究科　†† 立命館大学情報理工学部

1　はじめに

近年，FA をはじめとした組込みシステムでは，信頼性や応答性に加えて，高い機能性が求められている．組込みシステムでは，主に VxWorks に代表されるリアルタイム OS(RT-OS) が利用される．RT-OS は，機器制御を主目的とした最小限の機能提供により，処理時間の予測可能性を高め，リアルタイム性を保証する．そのため，RT-OS を機能拡張し，高機能化した場合，処理時間の予測可能性が低下し，リアルタイム性の保証が困難になる．一方，高度な情報処理を行うシステムには，Linux に代表される高機能 OS が利用される．高機能 OS は，高い機能性を実現するために，複雑なシステム構造をしており，処理時間の予測可能性は低い．そのため，高機能 OS によるリアルタイム処理は困難である．

先述の要求に応える既存の手段として，システムに複数の計算機を搭載し，RT-OS と高機能 OS を同時動作させる手法と，機能性と信頼性を兼ね備えたハイブリッド OS を用いる手法がある．しかし，前者は省電力化・省スペース化を困難にし，ハードウェアの開発コストを増加させ，後者はシステム毎に最適化を施す必要があり，ソフトウェアの開発コストを増加させる．

この問題を解決するために，仮想化技術により複数の OS を共存させる手法が考えられる．RT-OS が動作する計算機と高機能 OS が動作する計算機を，ひとつの計算機に統合することで，ハードウェアの製造コストを削減できる．また，仮想化により，既存のソフトウェア資源を有効活用することが可能になり，ソフトウェアの開発コストを削減できる．

以上の背景から，仮想化技術により，単一の計算機上で RT-OS と高機能 OS を同時稼動させる，リアルタイム仮想計算機モニタ Natsume の研究を行っている．

2　リアルタイム仮想計算機モニタ Natsume

2.1　概要

Natsume は，仮想化技術により RT-OS と高機能 OS を共存可能にするシステムソフトウェアである．Natsume の構造を図 1 に示す．Natsume は，各 VM の管理用インターフェースを提供する特権 Domain，RT-OS を動作させ，リアルタイム性を保証する RT-Domain，高機能 OS が動作する Domain-U，各環境に対してハードウェア資源を提供する RT ハイパーバイザで構成される．

Natsume は，信頼性が求められる処理を RT-OS で実行し，高い機能性が必要な処理を高機能 OS で実行することで，信頼性と応答性を保証しつつ，高い機能性を実現する．また，複数の OS を単一計算機上に共存させることで，ハードウェア開発のコスト削減を実現する．さらに，Natsume は既存の VMM である Xen[1] を基に開発しており，Xen に対応した豊富なソフトウェア資源の有効利用を可能にする．これにより，ソフトウェア開発のコスト削減も実現する．

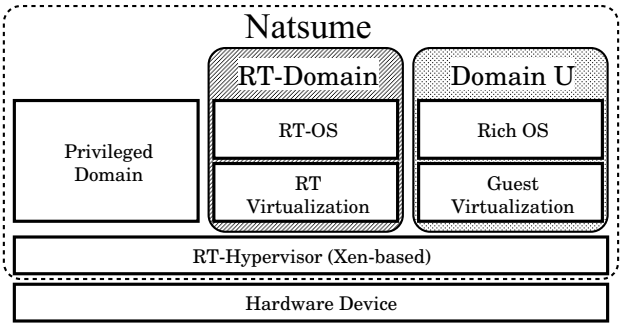


図 1　Natsume の構造

2.2　Natsume における資源管理

単一の計算機上で複数の VM が動作する場合，計算機資源は各 VM で共有される．これは，あるゲスト OS の処理性能が，他のゲスト OS の状況に影響されることを意味する．そのため，複数の VM が動作する環境では，RT-OS によるリアルタイム性の保証が困難になる．Natsume では，RT-Domain に主要な資源を占有させることで，この問題を解決する．対象とする資源は以下に挙げるものを想定している．

CPU 資源

主記憶資源

入出力資源

CPU 資源と主記憶資源は，Xen が提供する XenTools によって特定の Domain よる占有が可能である．入出力資源は，PCI Pass-through を基にした資源管理を行う．

2.3　PCI Pass-through とその問題点

PCI Pass-through は，PCI デバイスを Domain に排他的に割り当てる機構である．対象となるデバイスは，他の Domain から隠蔽され，占有したゲスト OS のデバイスドライバによる直接アクセスが可能になる [2].

Xen では，デバイスからの物理割込みを Event Channel を介し，仮想割込みとして Domain に通知する．Event Channel は，Xen の汎用的な通信機構であり，全ての Event を平等に通知する．すなわち，全てのデバイスは，割込み通知において平等に扱われ，割込みが発生した順に Domain に通知される (割込みの直列化)．これにより，PCI Pass-through により，あるデバイスの I/O 処理を占有した場合においても，割込み通知において他のデバイスと平等に扱われるため，そのパフォーマンスは，他のデバイスや Domain の処理の影響を受ける．これは，RT-OS によるリアルタイム性の保証を困難にする．

3 Xen における割込み通知

Xen における，Event Channel を介した割込み通知の全体像を図 2 に示す．Xen では，これらの機構を以下の手順で用いて，割込み通知を実現する．

1. システム初期化時：全デバイスの割込みハンドラを Xen の do_IRQ() に設定する
2. Domain 初期化時：ゲスト OS が Xen に，callback 用エントリポイント (hypervisor_callback:) の CS と EIP の値を登録する (HYPERVISOR_callback_op())
3. 割込み発生時：ゲスト OS に通知する割込みと判断した場合は，send_guest_pirq() を呼び出す
4. send_guest_pirq()：発生した IRQ・Vector から Event に変換し，その Event がマスクされていないか確認する．マスクされていなければ，割込み処理中であることを表す 3 つの pending bit をセットする (evtchn_upcall_pending，evtchn_pending_sel，evtchn_pending)
5. vcpu_kick()：Event をハンドルする VCPU の pause_flag を確認し，pause 状態なら解除する
6. send_IPI_mask()：Event をハンドルする VCPU の処理を行う CPU に対して IPI を発行し，割込み復帰コード (ret_from_intr:) を実行させる
7. ret_from_intr：pending bit がセットされていれば，登録済みの CS と EIP の値を基にして，ゲスト OS 用のスタックに，スタックフレームを作成し，割込み処理から復帰する
8. hypervisor_callback：ゲスト OS 側に制御が移り，ゲスト OS の割込みハンドラが動作する．この時，pending bit をクリアする

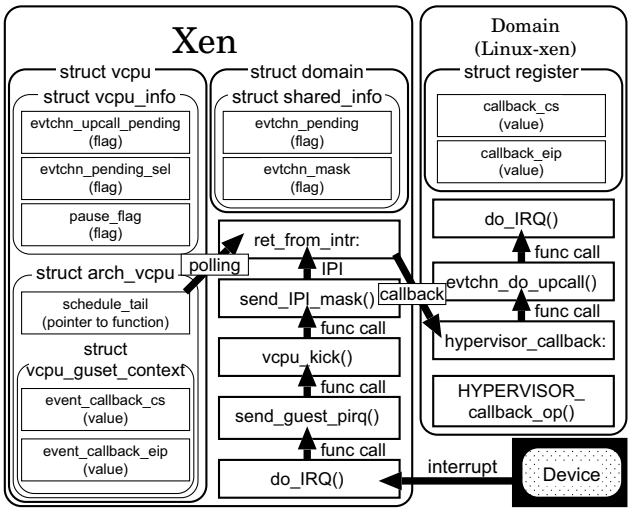


図 2　Xen の割込み通知の全体像

9. evtchn_do_upcall()，do_IRQ()：割込み処理中に新たに割込みが発生し，pending bit がセットされた場合，Xen へ制御を返す前に，繰り返しゲスト OS の割込みハンドラを起動する
10. schedule_tail：割込み通知先 VCPU を処理する CPU がスケジューリングされていないことが原因で，IPI による callback に失敗することがある．この場合，VCPU スケジューリングの際に，shecule_tail から ret_from_intr が実行され，ポーリングによる callback が実行される．

4 提案する割込み通知モデル

RT-OS によるリアルタイム性の保証を実現するには，I/O の占有を実現する PCI Pass-through に加えて，割込み通知経路の占有を実現する割込み通知モデルが必要になる．提案する割込み通知モデルを以下に示す．

1. システム初期化時に，占有デバイスを特定する
2. 占有デバイスに専用の Vector を割り当て，他のデバイスと区別する
3. 占有デバイスからの割込みを専用ハンドラでハンドルし，ハンドルした割込みを即座に通知する

4.1 占有デバイスの特定

あるデバイスの割込みを，他のデバイスと異なる経路で通知するためには，対象となるデバイスを予め特定する必要がある．提案手法では，Xen のオプションを利用することで，これを実現する．

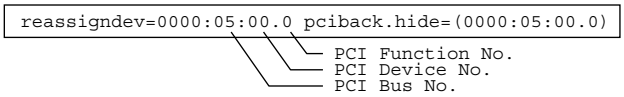


図 3　PCI Pass-through を有効にするオプションの例

Xen で PCI Pass-through を利用する際，システム起動時のオプションとして，対象デバイスの情報を記述する必要がある．記述例を図 3 に示す．この情報を予め取得しておくことで，割込みを占有させるデバイスを一意に判別できる．

4.2　MSI によるデバイスの区別

割込みを検出した際，対象デバイスとその他のデバイスを区別する必要がある．提案手法では，これを Message Signaled Interrupts(MSI)[3] を利用して実現する．

MSI は，PCI2.2 以降で導入された割込み通知機構である．通常，デバイスからの割込みは，割込みコントローラの割込み信号線を用いて通知されるが，MSI では，割込みコントローラに対するメモリアクセスを行うことで実現する．デバイスがメモリアクセスにより書き込む値は，各デバイスの機能毎に設定が可能である．これにより，割込みを検知した際，割込みを発生させたデバイスと，その要因を判別できる．

MSI を用いて，割込みを占有するデバイスに対して，固有 Vector を割り当てることで，対象デバイスからの割込みを，専用の割込みハンドラで処理できる．また，x86 アーキテクチャでは，MSI を用いて，割込み通知先の CPU を限定することも可能であり，通知先 CPU を，RT-OS が動作する CPU に限定することにより，割込み転送のオーバヘッドを削減することも可能になる．

4.3　専用ハンドラによる割込み通知

割込みを占有するデバイスから割込みが発生した場合，専用の割込みハンドラが割込み通知を行う．このハンドラでは，発生した割込みを通常のデバイス割込みとは異なった通知機構を用いて，割込み通知を行う．提案手法では，図 4 と以下に示す 3 通りの割込み通知機構を検討している．

(1) 割込みの横取りと callback を実行する機構　割込みハンドル後，即座にゲスト OS 用のスタックに callback 用のスタックフレームを構築する．その後，実際に callback を行うことで，処理を横取りした状態で，ゲスト OS に割込み処理を実行させることができると考えられる．また，専用の pending bit を用意しておくことで，横取りによる割込み処理中に，さらなる横取りが発生することを防止する．

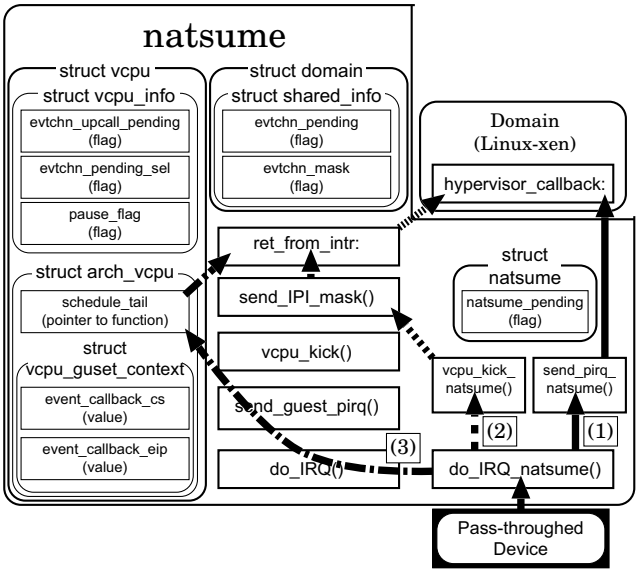


図 4　提案する割込み通知経路

(2) IPI を繰り返し実行する機構　まず，通常の割込み通知機構と同様に処理を行う．その後，割込みを処理する CPU に対して IPI を発行する時点で，callback に成功するまで IPI を繰り返す．これにより，ポーリングによる割込み通知に移行し，割込み通知が遅延することを防ぐことができると考えられる．

(3) VCPU スケジューラを利用する機構　IPI による手法と同様に，通常の割込み通知機構と同様の処理を行う．その後，IPI による callback を実行する直前に，VCPU スケジューラを用いて，callback 可能な状態に VCPU を再スケジューリングする．これにより，ポーリングによる割込み通知に移行し，割込み通知の遅延を防止できると考えられる．

5　考察

提案した割込み通知モデルの中で，最も割込みの占有に近く，高いパフォーマンスが期待できるものは，割込みの横取りと callback を実行する機構を用いたものであると考えられる．しかし，この手法は既存の割込み通知機構や，通知先ドメインの動作状況を意識する必要があり，他の手法と比較して，実装が困難であると考えられる．一方，IPI を繰り返し利用する機構と，VCPU スケジューラを利用する機構は，既存の割込み通知機構を応用する分，実装が容易であると考えられる．しかし，これらの機構は，割込み通知の直列化を解消することはできないため，根本的な問題は解決できないと考えられる．

6 おわりに

本論文では，仮想計算機技術を用いて，RT-OS と高機能 OS を共存させるシステムである，リアルタイム仮想計算機モニタ Natsume の概要と，その資源管理機構について述べた．次に，Natsume の資源管理機構の基である PCI Pass-through は，I/O 処理の占有を実現する一方，割込み通知経路の占有を実現できないため，RT-OS によるリアルタイム性の保証を困難にすることについて述べた．そして，この問題を解決するために，PCI Pass-through 対象デバイスと，その他のデバイスを区別し，専用の割込みハンドラによって，割込みの占有を実現する割込み通知モデルをいくつか提案した．その考察として，割込みの横取りを行う手法は，割込み占有を高いパフォーマンスで実現できると考えられるが，実装は比較的困難であるとし，IPI による手法と VCPU スケジューラを利用する手法は，実装が容易だが，割込みの直列化という問題を解決できないため，根本的な解決にはならないことを説明した．

今後，提案した割込み通知モデルの詳細な設計とプロトタイプ実装を行い，評価を行う予定である．

参考文献

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield: “Xen and the Art of Virtualization,” ACM Symposium on Operating Systems Principles (2003).

[2] William von Hagen: “Professional Xen Virtualization,” Wiley Pubnlishing, Inc, pp. 109-115 (2008).

[3] PCI-SIG: “PCI Local Bus Specification Revision 3.0,” `http://www.pcisig.com/specifications/`, pp. 237-254 (2002).

リアルタイム仮想計算機モニタNatsumeにおける割込み通知モデルの提案

立命館大学大学院　理工学研究科
毛利研究室
渡邉　和樹

2010/9/21

発表内容

- はじめに
- リアルタイム仮想計算機モニタ Natsume
- Natsumeにおける資源管理
- 割込み通知モデルの提案
  - Xenにおける割込み通知の全体像
  - MSIによる割込み発生デバイスの区別
  - 専用ハンドラによる割込み通知モデル
- 考察
- 今後の予定
- おわりに

はじめに（1/2）

- 近年，組込みシステムにおいて，開発コスト削減や，信頼性と機能性の両立が求められている

RT-OS
限定した機能提供で
リアルタイム性を保証

Rich-OS
構造が複雑で
リアルタイム化が困難

- 既存の解決手段

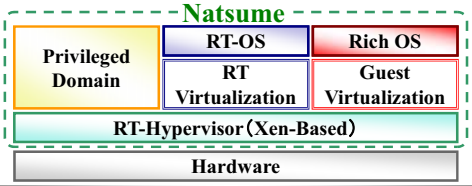
はじめに（2/2）

- 複数の計算機や最適化したOSを用いる手法は，ソフトウェア・ハードウェア開発コスト増加の原因
- 仮想化技術を活用し，単一の計算機上で，複数のOSを動作させるアプローチが考えられる
  - ハードウェア資源の製造コストを削減できる
  - OSやアプリケーションといった，既存のソフトウェア資源を有効活用できる

仮想計算機技術を用いて，2種類のOSを共存させるシステム
“Natsume” の研究と開発

リアルタイム仮想計算機モニタ Natsume

- RT-OSと高機能OSを一つの計算機で共存
- RT-OSが動作する仮想化環境（ドメイン）において，リアルタイム性を保証できる資源管理を行う
- Xen 3.4.x の準仮想化を基にして開発
  - 既存のXenに対応した OS やアプリケーションといった，豊富なソフトウェア資産の有効活用が可能



Natsumeにおける資源管理

- Natsume では，RT-OSに主要な資源を占有させることでリアルタイム性の保証を実現
  - CPU資源
  - 主記憶資源
  - 入出力資源

  → Xen付属のXenToolsで設定可能（CPU資源，主記憶資源）
  → PCI Pass-through を基にした割当て手法（入出力資源）

- PCI Pass-through
  - Xen 3.2.x 以降で利用可能な資源管理機構
  - 特定のドメインによるPCIデバイスの占有を実現
  - 割り当てたデバイスは他のドメインから隠蔽され，デバイスドライバによる直接アクセスが可能

PCI Pass-through の課題点

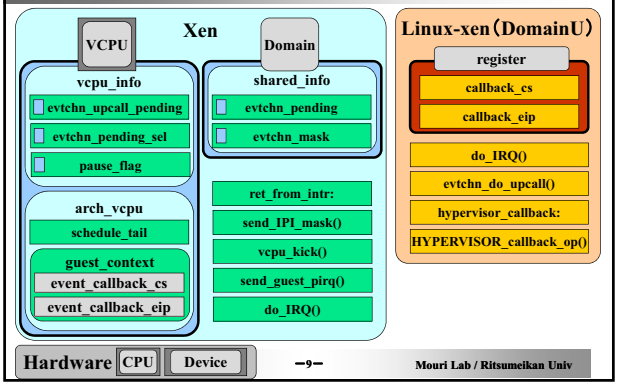
- Xenでは，デバイスから通知される割込みは，VMM内のEvent Channelを介してドメインに通知される
- Event Channel では，全てのデバイスが平等に扱われ，順に通知される（割込みの直列化・横取りの禁止）
- PCI Pass-throughでI/Oを占有したとしても，割込み経路の共有によって，パフォーマンスが低下する
  - 同時に稼動するデバイスやドメインの影響を受ける

特定のデバイスをI/Oだけでなく，割込み通知経路も含めて，占有可能にする資源管理機構が必要

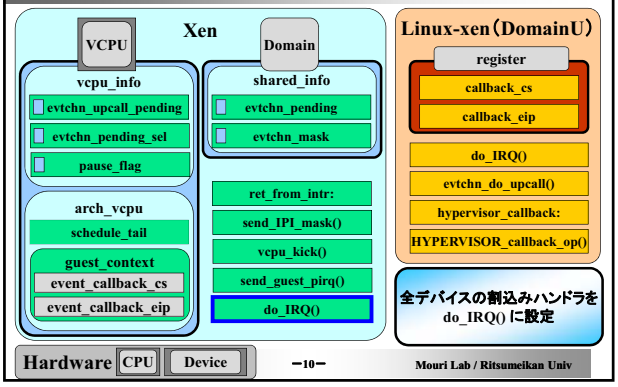
提案する割込み通知モデル

- 通常のデバイス割込みとは異なった割込み通知経路を用意
- システム初期化時に，割込みを占有させるデバイスを特定
- 特定したデバイスに専用の割込みベクタを用意し，他のデバイスと明確に区別
- 特定したデバイスからの割込みを，専用の割込みハンドラを用いてハンドリング

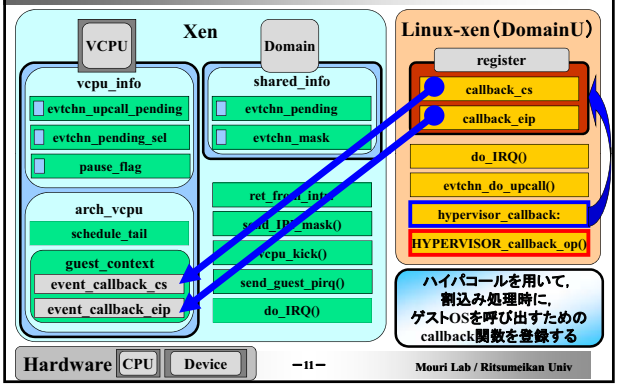
Xenにおける割込み通知の全体像



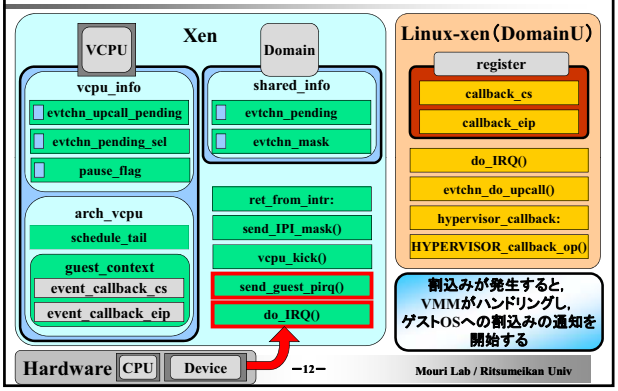
VMM側の初期化（システム起動時）



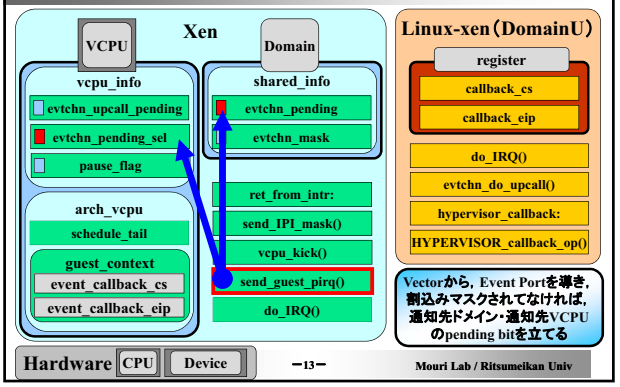
ドメイン側の初期化（ドメイン起動時）



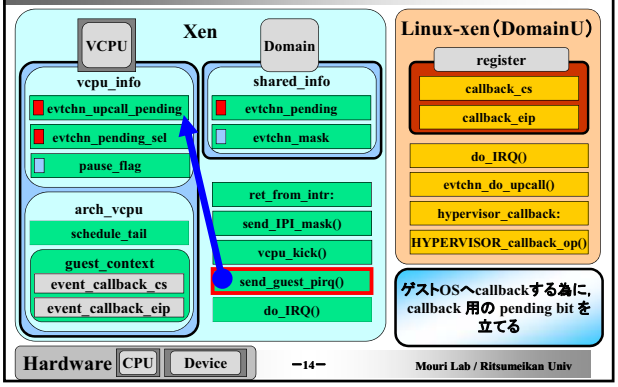
割込みの発生



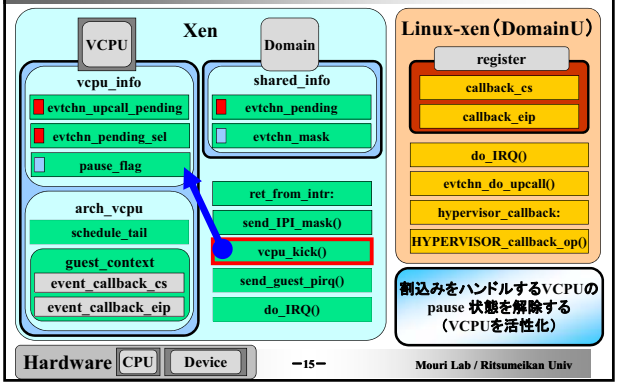
割込み発生の通知



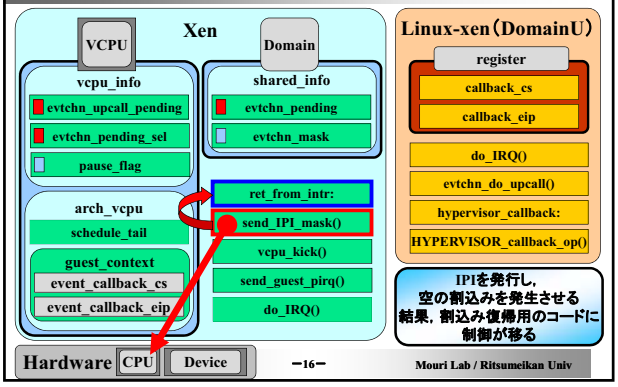
callback（upcall）の準備（1/2）



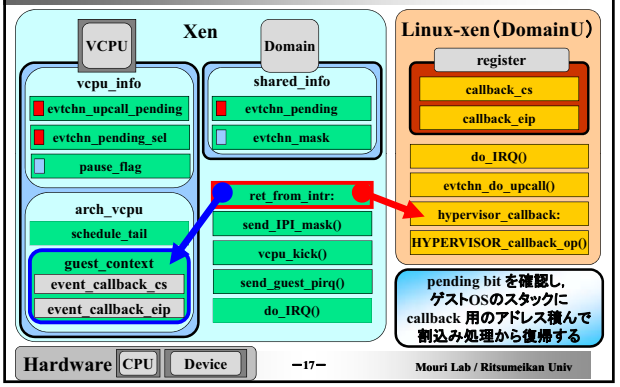
callback（upcall）の準備（2/2）



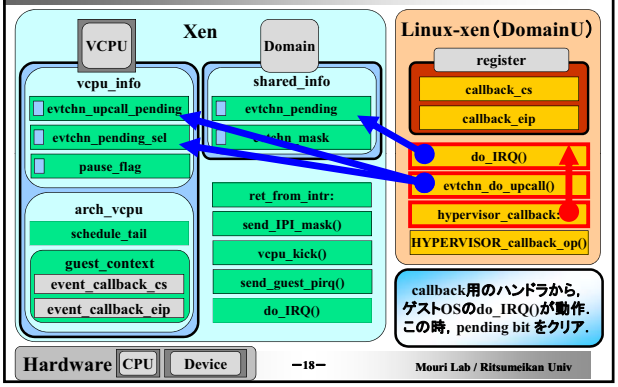
callbackの実行（1/2）



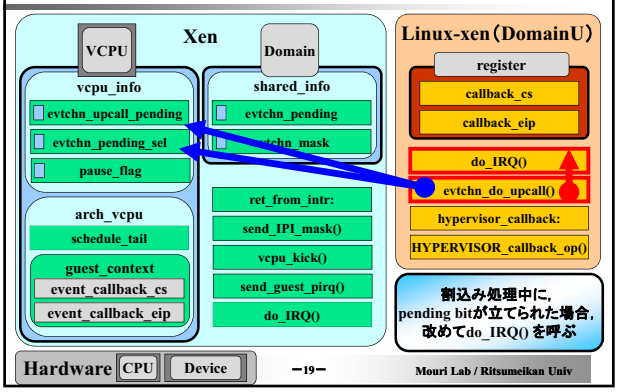
callbackの実行（2/2）



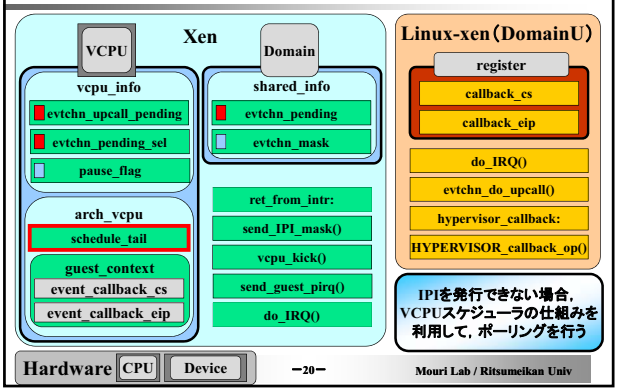
ドメインによる割込み処理



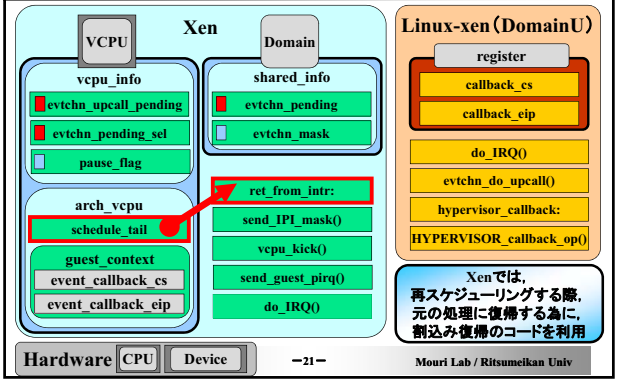
処理中に発生した割込みの処理



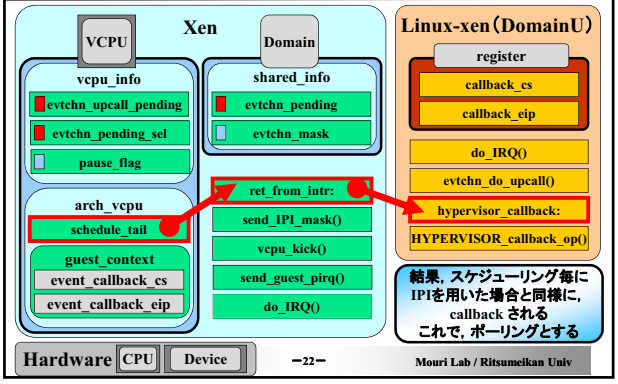
ポーリング（1/3）



ポーリング（2/3）



ポーリング（3/3）



提案する割込み通知モデル（再掲）

- 通常のデバイス割込みとは異なった
  割込み通知経路を用意
- システム初期化時に，
  割込みを占有させるデバイスを特定
- 特定したデバイスに専用の割込みベクタを
  割り当て，他のデバイスと明確に区別
- 特定したデバイスからの割込みを，
  専用の割込みハンドラを用いてハンドリング

占有デバイスの特定

- 現状のXenでは，PCI Pass-throughを利用する際，
  起動時にXenのオプションとして，対象デバイスの
  情報を記述する

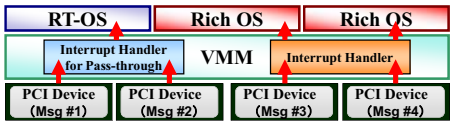
```
reassigndev=0000:05:00.0 pciback.permissive pciback.hide=(0000:05:00.0)
```

0000 : PCIバス番号 : PCIデバイス番号 : PCI機能番号

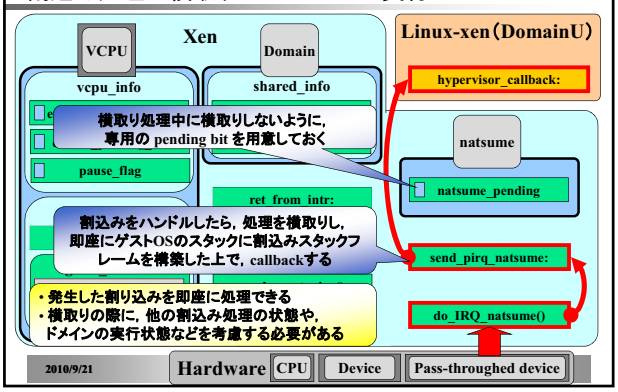
- この情報から，Pass-throughの対象と
  なるデバイスを一意に判別できる
- システム初期化の際，Pass-through デバイスに，
  特定のIRQ・Vectorを占有させることで，
  割込み発生時に，デバイスの特定が可能

MSIによるデバイスの区別

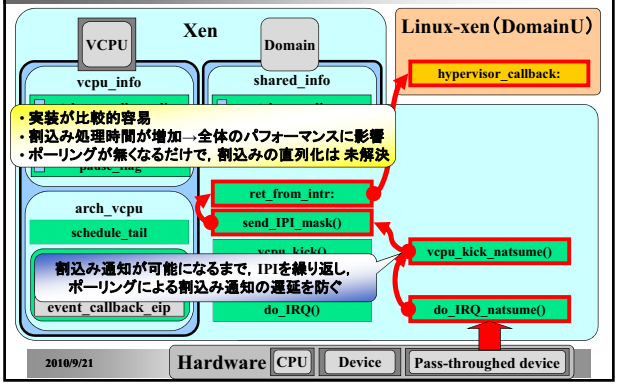
- MSI（Message Signaled Interrupts）は，
  割込みコントローラのINTx信号線を利用せずに，
  コントローラへのメモリアクセスによって割込みを通知する方式
  - IRQ（0-15/23）の代替に，任意のVector（0-255）番号が利用可能
  - x86 アーキテクチャでは，割込み通知先の物理CPUを限定可能
    - 割込み転送のオーバヘッドを削減
- デバイス毎にVector番号を割当てることで，
  割込みが発生した際にそのデバイスを区別可能
  - Pass-throughデバイスからの割込みを専用ハンドラで処理する



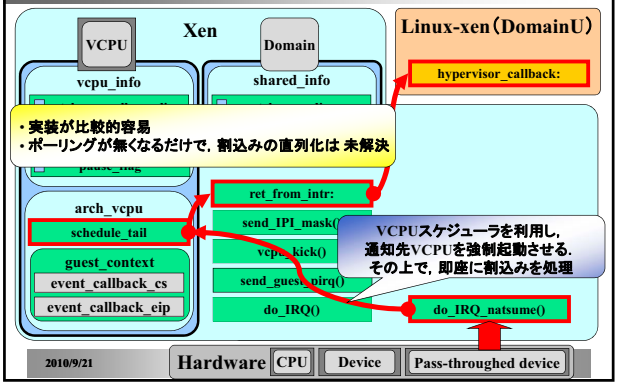
提案する割込みモデル その1
割込み処理の横取りと callback の実行



提案する割込みモデル その2
IPIを繰り返し，割込み通知の遅延を防ぐ手法



提案する割込みモデル その3
VCPUスケジューラを利用する手法



考察

- 最も”割込みの占有”に近い方式は，
  「割込み処理の横取り」を行う手法
  - 元々存在している割込み通知機構や，
    ドメインの動作状態を意識する必要がある
  - 例）元々の割込み通知機構が利用している資源や
    メモリ領域の排他制御など
- 実装が比較的容易な方式は，「IPIを繰り返す」
  「VCPUスケジューラを利用する」手法
  - 割込み通知そのものは，元々存在している機構を利用
  - 割込みに優先度をつけられるようになるが，
    割込み通知の直列化は解決できない

今後の予定

- 設計
  - 提案した割込み通知モデルの実現可能性・
    課題点を調査・考察し，詳細な設計を行う
    - 複数の手法を組み合わせた割込み通知モデルも
      視野に入れる
- 評価
  - プロトタイプ実装を行い，評価を行う
  - 対象：Xen 3.4.3 + Linux 2.6.18-xen
    - 後々，RT-OSも対象に

おわりに

- RT-OSとRich-OSの共存を目的とした，仮想計算機モニタ”Natsume”の研究・開発
  - Natsumeは各種資源をRT-OSに占有させ，信頼性を確保
  - Xenの割込み通知機構は，全デバイスで割込み経路を共有
  - I/Oを占有しても，他のドメインやデバイスの影響を受ける
- VMMで占有したデバイスを特定・区別し，割込みの占有を実現する割込み通知モデルの提案
  - MSIにより，割込み番号を占有させ，デバイスを特定
  - 占有した割込み番号からの割込みを，専用のハンドラで処理
  - 共有デバイスと異なる割込み通知機構による割込み通知
- 占有デバイス専用の割込み通知機構を提案
  1. 割込み処理の横取りとcallbackを行う手法
  2. ポーリングによる割込み通知の遅延を防ぐ手法
  3. VCPUスケジューラを利用する手法
- 今後，詳細な設計とプロトタイプ実装を行う予定

協調型仮想計算機システムにおけるCPUスケジューリング方式

小野 利直 † 毛利 公一 ††
† 立命館大学情報理工学部 †† 立命館大学大学院理工学研究科

1 はじめに

近年, 仮想化技術を搭載したマルチコアプロセッサが登場し, 普及している. このようなマルチコアプロセッサと仮想化技術を組み合わせることにより, 複数の仮想計算機 (以下, VM と記す) に物理的なコアをそれぞれ割り当てることが可能となる. また, VM ごとに割り当てるコア数を指定することで, それぞれの処理能力を自由に変更することが可能である. これにより, 従来のシステムでは, 計算機に依存していた OS の処理能力を仮想化レイヤで自由に変更し, OS の特性を考慮したプロセッサの割り当てを行うことで, 計算機資源を有効に利用できるようになった.

従来の仮想計算機モニタ (以下, VMM と記す) の CPU スケジューリングは, ゲスト OS 上で動作するプロセスやスレッドのことを考慮しておらず, 実行状態であるか, アイドル状態であるかの 2 種類の状況だけを判断している. また, CPU スケジューラが動作しているコア資源の数を動的に変更するということはない. 複数の OS が 1 台の計算機上で動作する VM 環境上において, VMM が, プロセッサコアを OS の負荷状況に応じて配分することによって, より効率的な資源利用が可能となる. また, 仮想 CPU 数を動的に変更することにより, 必要のないコアを停止させ消費電力を抑えることや, 停止中のコアを処理能力を必要としている OS に割り当てることで, 各 OS の要求を満たすことも可能である.

我々は, 上記のように VMM と OS が協調して動作を行うシステムである協調型仮想計算機システムを提案する [1]. 現在, OS のターゲットとして Linux を, VMM のターゲットとして Xen を用いて開発を行っている.

本稿では, はじめに協調型仮想計算機システムについて述べ, 次にプロトタイプ実装について述べ, 最後に今後の予定について述べる.

2 協調型仮想計算機システム

協調型仮想計算機システムは, VMM とゲスト OS が協調して動作し, 互いに情報交換を行いながら, 効率的に資源を利用するシステムである. 従来, VMM とゲスト OS が各々独立してスケジューリングを行っていたが, 本システムでは, VMM が, ゲスト OS 上の物理資源の利用率やプロセスの動作状況などの情報を考慮したスケジューリングを行う. これにより, 従来のシステムでは検知出来なかった, 使用率が低い時のプロセッサの使用を検知でき, より資源の必要なゲスト OS に多くの資源を割り当てる事が可能となる.

また, 本システムでは, 実 CPU を動的に変動させるだけでなく, 仮想 CPU を動的に増減させることで, より効率的な資源の利用が可能となる.

3 処理のパターン分類

どのような時に実 CPU や仮想 CPU を割り当てると処理効率が向上するのかを確認するために, 予備調査として性能評価を行った. その結果, 処理を 4 つのパターンに分類することが可能であることが分かった.

3.1 多数のプロセスが頻繁に切り替わり CPU を使用する処理

このパターンは, 実 CPU が増加すると処理性能が向上し, 仮想 CPU が増加すると処理性能が低下していくという特徴がある. これは, 実 CPU と仮想 CPU を割り付ける CPU スケジューラのオーバヘッドよりも, 仮想 CPU とプロセスを割り付けるプロセススケジューラのオーバヘッドの方が小さいということに起因していると考えられる. このことから, このような処理を行う時には, 可能な限り実 CPU を割り当て, 仮想 CPU を減少させると良い.

また, このような処理はロードアベレージが高く, コンテキストスイッチ回数が多いという特徴がある.

3.2 少数のプロセスが連続して CPU を占有する処理

このパターンは, 実 CPU が増加すると処理性能が向上する一方で, 仮想 CPU が増加しても処理性能に変化はないという特徴がある. これは, CPU やプロセスの遷移があまり起こらないためであると考えられる. そのため, このような処理を行う時には, 仮想 CPU を減少させるときに起こるオーバヘッドを考慮し, 実 CPU を増加, 仮想 CPU を維持することにより処理性能を向上させることが可能である.

また, このような処理はロードアベレージが低く, コンテキストスイッチ回数は少ないという特徴がある.

3.3 少数のプロセスが頻繁に切り替わり CPU を使用する処理

このパターンは, 実 CPU が増加すると処理性能が向上し, 仮想 CPU 数がプロセスの数を越えるまで一定の性能, 仮想 CPU 数がプロセスの数を越えると処理性能

A CPU scheduling algolithm of Co-Virtual Machine Monitor
Toshinao Ono† and Koichi Mouri††
†College of Information Science and Engineering, Ritsumeikan University
††Graduate School of Science and Engineering, Ritsumeikan University

が低下していくという特徴がある. つまり処理がプロセス数に依存している状態にあると言える. このため, 仮想 CPU 数とプロセス数から割り当てる実 CPU 数を決定する必要がある.

また, このような処理はロードアベレージが低く, コンテキストスイッチ回数は多いという特徴がある.

3.4 CPU 以外にボトルネックが存在する処理

このパターンは, 実 CPU が増加しても性能は変化せず, 仮想 CPU が増加すると性能が低下していくという特徴がある. これは CPU 以外の部分でボトルネックが存在するためである. この場合には, 実 CPU, 仮想 CPU ともに減少させ, 余った実 CPU を他のゲスト OS に割り当てることで, システム全体の処理効率が向上する.

また, このような処理には IO 待ちがあるという特徴がある.

4 プロトタイプ

今回, 本実装の前に, アルゴリズムのテストを行うためのプロトタイプとして, ホスト OS が CPU の増減を判断し, xm コマンドで CPU の割り付けを行うというプログラムの作成を行った.

4.1 アルゴリズム

ハイパーバイザは, ゲスト OS 上でどのパターンの処理を行っているかを検出し, それに適応した CPU 割当てを行うことにより処理効率の向上を図る. 図 1 にどのパターンか検出するアルゴリズムを示す. 多数のプロセスが頻繁に切り替わり CPU を使用する処理をパターン 1, 少数のプロセスが連続して CPU を占有する処理をパターン 2, 少数のプロセスが頻繁に切り替わり CPU を使用する処理をパターン 3, CPU 以外にボトルネックが存在する処理をパターン 4 とする.

4.2 実装

プロトタイプ実装では, ゲスト OS はハイパーバイザではなく, ホスト OS にデータを送信し, ホスト OS がそのデータを元に, CPU の割当てを行う. 以下にゲスト OS 側の処理を示す.

1. 必要なデータの取得
2. アルゴリズムから資源の不足を判断
3. ホスト OS へデータを送信

以下にホスト OS 側の処理を示す.

1. ゲスト OS からデータを受信
2. CPU 割当ての判断
3. xm コマンドを用いて CPU の再割当て

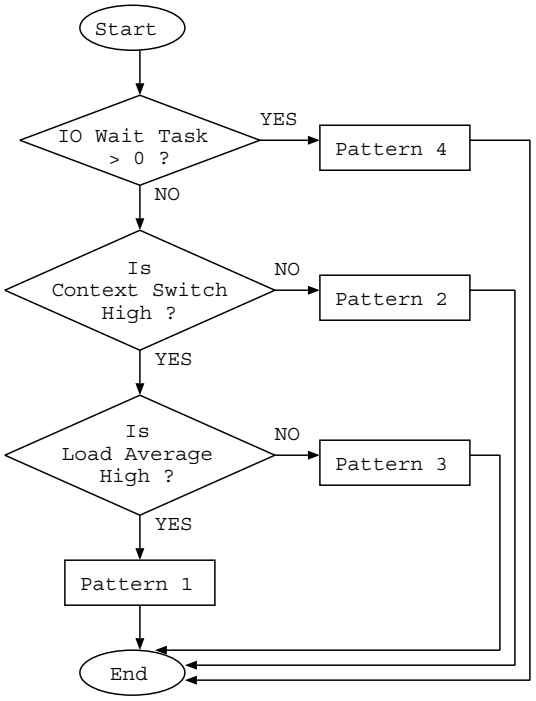


図 1 パターン検出アルゴリズム

表 1 計算機構成

環境	詳細
CPU	Core i7 920
RAM	6GB
Xen	3.4.1
Dom-U	Debian lenny i386 RAM1024MB 固定割当て

4.3 性能評価

作成したプロトタイプで, ベンチマークソフトを用いて性能評価を行った. 計測は, 表 1 に示す計算機構成で, Unixbench5.2.1 を使用し, プロセス数を 3 個に固定して行った. また, プロトタイプとの比較対象として, 実 CPU, 仮想 CPU がともに 3 個のとき, 2 個のとき, 1 個のときの性能評価も行った. 図 2 に評価結果のグラフを示す. またグラフは, CPU の個数が 3 個のときの性能を 1.0 とした比率で表記している.

パターン 1 に関しては, CPU を 3 個割り当てたときに対して, 96%の性能を計測することができた. これは, プロトタイプにおいても, ほぼ常に 3 個割り当てているということであり, 正確に処理を検出できていることが分かる.

パターン 2 に関しては, 3 個のときの 54%で, 1 個のときと 2 個のときの中間程度の性能しか計測することができなかった. 原因は調査中であるが, パターン 2 の処理

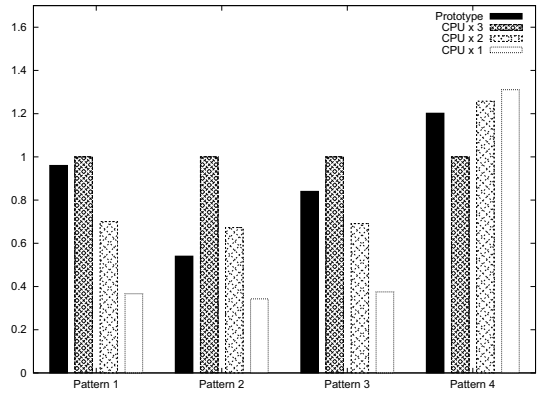


図 2　プロトタイプ性能評価

を行っているときには, 頻繁に CPU を増減させるという現象を見ることができた.

パターン 3 に関しては, CPU が 3 個のときの 84%の性能で, 2 個のときと 3 個のときの中間程度の性能を計測できた.

パターン 4 に関しては, CPU が少ないときの方が処理性能は向上するため, CPU を減らす処理を行っている. このため, CPU が 3 個のときの 120%の性能, CPU が 1 個のときの 92%の性能を計測することができた.

4.4　考察

以上の性能評価により, 常に最大数の CPU を割り当てていなくとも, その時々の処理に合わせて最適な数を割り当てることで, 最良の結果に近いパフォーマンスを出せるということが分かった.

5　おわりに

本稿では, 協調型仮想計算機システムの概要を述べ, 処理によって資源の割当て方を変えることについて述べた. また, プロトタイプ実装を行い, その性能評価について述べた. 今後は, アルゴリズムの精査, 本実装を行っていく.

参考文献

[1] 荒木 裕靖, 毛利 公一: “協調型仮想計算機システムにおける協調機構,” 第 71 回全国大会講演論文集, vol. 1, pp. 63-64, 情報処理学会 (2008).

[2] David Chisnall 著, 渡邉 了介訳: “仮想化技術 Xen 概念と内部構造,” 毎日コミュニケーションズ, pp. 301-324 (2008).

www.asl.cs.ritsumei.ac.jp

協調型仮想計算機システムにおける CPUスケジューリング方式

立命館大学大学院
毛利研究室
小野利直

発表内容

- はじめに
- 協調型仮想計算機システム
- 資源の増加を行う処理の流れ
- CPU数の変動による性能の変化
- プロトタイプ実装
  - アルゴリズム
  - 性能評価
- おわりに

はじめに(1/2)

- 近年, マルチコアプロセッサと仮想化技術が普及
  - 家庭用計算機だけでなく, サーバや組込み機器など, いろいろな場面で利用
- 従来の仮想計算機モニタ
  - 仮想計算機構築時に割り当てた仮想CPU数が不変
  - ゲストOSの状況を考慮せずにCPUを割当て
    - 本当にCPUが必要かは分からない
    - 状況によりシステム全体の処理性能が低下

効率的な資源割当てとは言えない

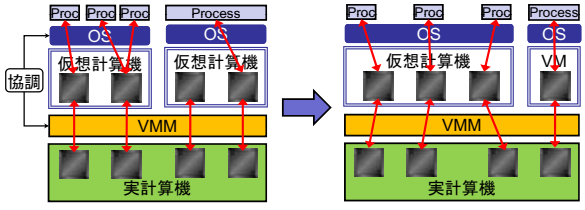
はじめに(2/2)

- 効率的な資源割り当て
  - ゲストOS上のプロセス情報を考慮
  - 実CPUの数と仮想CPUの数を動的に変更

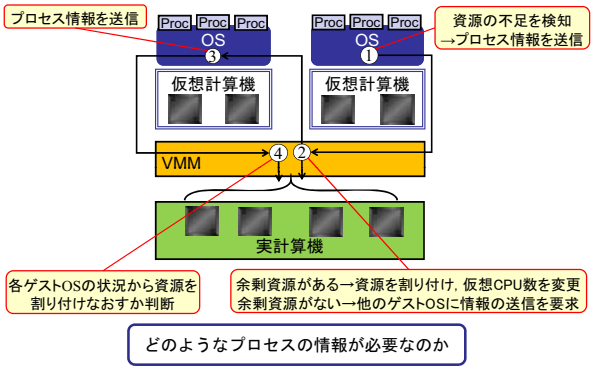
ハイパーバイザとゲストOSが協調し
情報のやりとりを行うことで実現

協調型仮想計算機システム

- VMMはゲストOS上の情報を元に資源を割当て
  - ゲストOSは自身のプロセスの情報をVMMに通知
  - VMMは情報を元に実CPU, 仮想CPUを割当て
- Xenを基に開発



資源の増加を行う処理の流れ



どのようなプロセスの情報が必要なのか

CPU数の変動による性能変化(1/2)

- 目的
  - プロセスの処理特性と実CPU, 仮想CPUの関係の調査
- 計測方法
  - UnixBench 5.2.1を使用
  - プロセス数：4個,
    実CPU数：1-3個, 仮想CPU数：1-8個
  - コンテキストスイッチ回数, ロードアベレージ
- 計測環境

CPU	Core i7 920
RAM	6GB
Xen	3.4.1
Domain-U	Debian lenny x86 RAM 1024MB 固定割り当て

CPU数の変動による性能変化(2/2)

- 実CPU, 仮想CPU1個の時の性能を1.00とする
- 各回, 各パラメータごとに性能の比率を算出
  - 例えば・・・

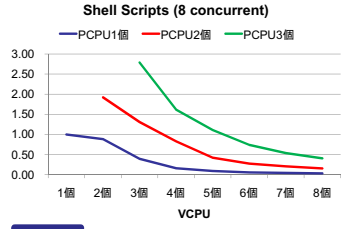
実CPU数 / 仮想CPU数	1個 / 1個 (性能比率)	1個 / 2個 (性能比率)
Dhrystone 2 using register variables	14471437 (1.00)	14335706 (0.99)
⋮	⋮	⋮
Double-Precision Whetstone	10988.2 (1.00)	5568.9 (0.51)
各回の性能比率の平均	1.00	0.87

性能比率：1.00とした時 ⇒ 性能比率：$\frac{14335706}{14471437} = 0.99$

各項目のパターン分類

- 各項目の性能変化は4パターンに分類可能
  - 実CPUの増加とともに性能が向上
    - ① 仮想CPUの増加とともに性能が低下
    - ② 仮想CPUの数にあまり影響を受けない
    - ③ 仮想CPU数がプロセス数を超えるまで一定の性能
      プロセス数を超えると性能が低下
  - 実CPUが増加しても性能が向上しない
    - ④ 性能が仮想CPU1個, 実CPU1個の時を超えない

①仮想CPUの増加とともに性能が低下(1/2)

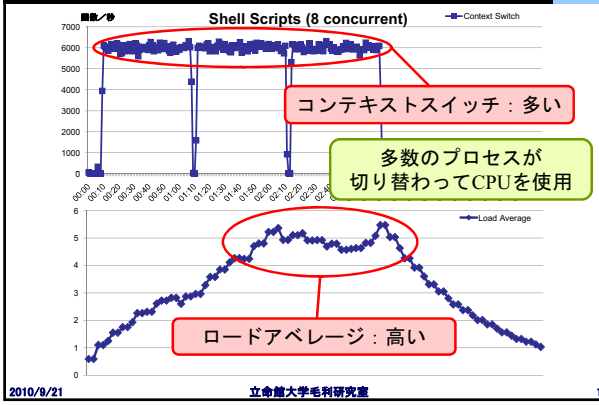


①の処理
- Process Creation
- Shell Scripts (1 concurrent)
- Shell Scripts (8 concurrent)

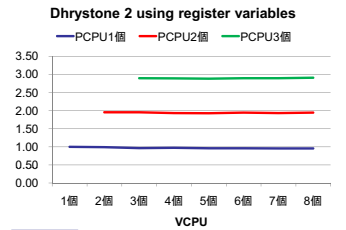
原因：CPUスケジューラのオーバヘッド ＞プロセススケジューラのオーバヘッド

対応：実CPUを増加 仮想CPUを減少

①仮想CPUの増加とともに性能が低下(2/2)



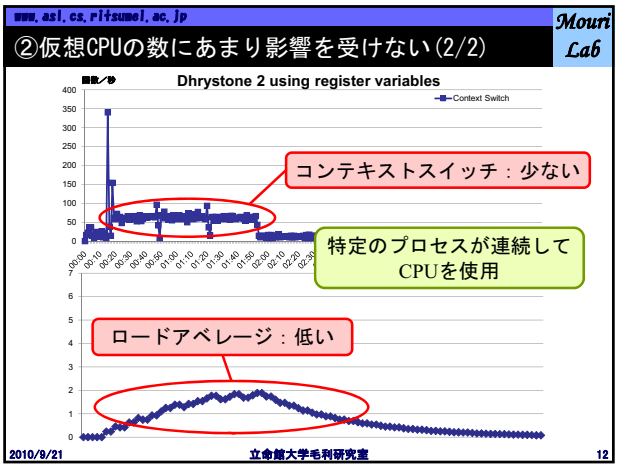
②仮想CPUの数にあまり影響を受けない(1/2)

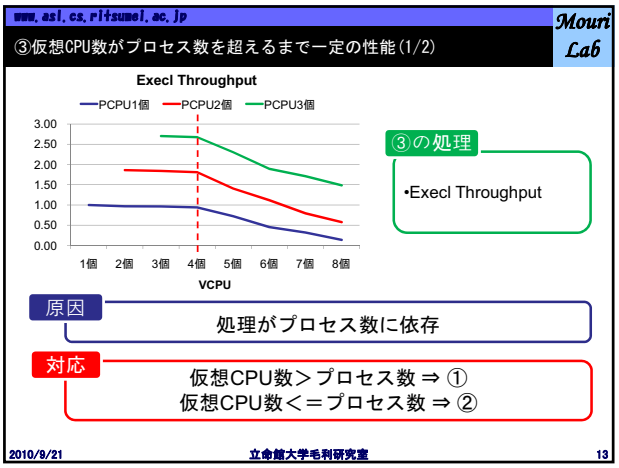


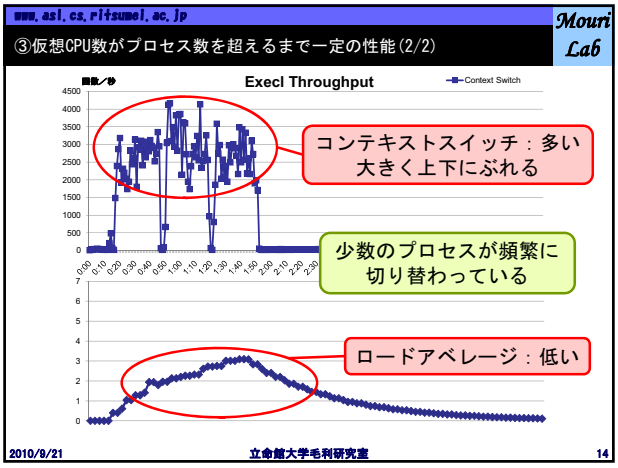
②の処理
- Dhrystone 2 using register variables
- Pipe Throughput
- Pipe-based Context Switching
- System Call Overhead

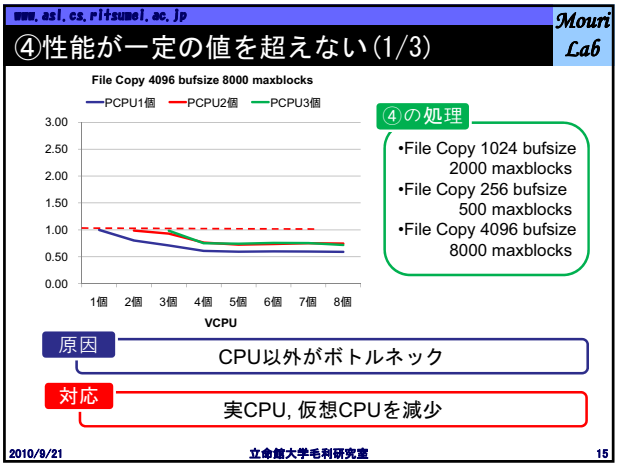
原因：CPUやプロセスの遷移が余り起こらない

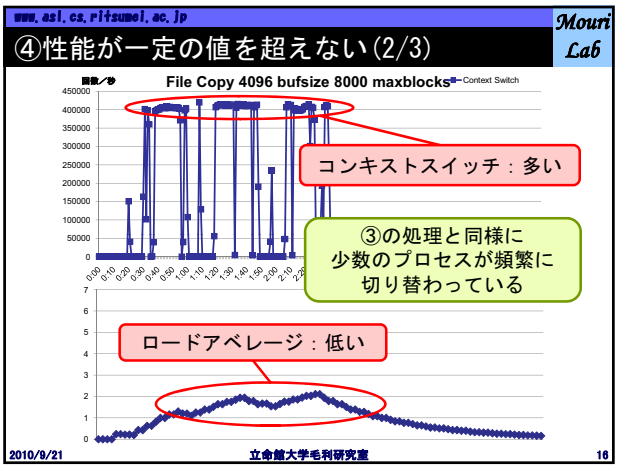
対応：仮想CPUを減少させる事によるオーバヘッドを考慮し, 実CPUを増加, 仮想CPUを維持

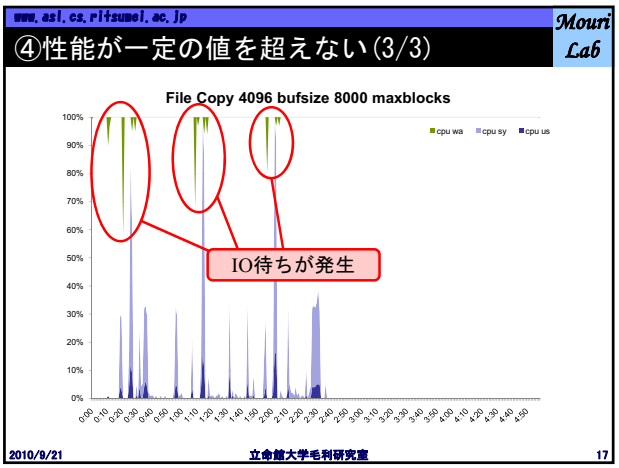

www.asl.cs.ritsumei.ac.jp
Mouri Lab
②仮想CPUの数にあまり影響を受けない(2/2)
Dhrystone 2 using register variables
Context Switch
コンテキストスイッチ：少ない
特定のプロセスが連続して
CPUを使用
ロードアベレージ：低い
2010/9/21
立命館大学毛利研究室
12


www.asl.cs.ritsumei.ac.jp
Mouri Lab
③仮想CPU数がプロセス数を超えるまで一定の性能(1/2)
Execl Throughput
PCPU1個
PCPU2個
PCPU3個
VCPU
③の処理
•Execl Throughput
原因
処理がプロセス数に依存
対応
仮想CPU数＞プロセス数 ⇒ ①
仮想CPU数＜＝プロセス数 ⇒ ②
2010/9/21
立命館大学毛利研究室
13


www.asl.cs.ritsumei.ac.jp
Mouri Lab
③仮想CPU数がプロセス数を超えるまで一定の性能(2/2)
Execl Throughput
Context Switch
コンテキストスイッチ：多い
大きく上下にぶれる
少数のプロセスが頻繁に
切り替わっている
ロードアベレージ：低い
2010/9/21
立命館大学毛利研究室
14


www.asl.cs.ritsumei.ac.jp
Mouri Lab
④性能が一定の値を超えない(1/3)
File Copy 4096 bufsize 8000 maxblocks
PCPU1個
PCPU2個
PCPU3個
VCPU
④の処理
•File Copy 1024 bufsize
2000 maxblocks
•File Copy 256 bufsize
500 maxblocks
•File Copy 4096 bufsize
8000 maxblocks
原因
CPU以外がボトルネック
対応
実CPU, 仮想CPUを減少
2010/9/21
立命館大学毛利研究室
15


www.asl.cs.ritsumei.ac.jp
Mouri Lab
④性能が一定の値を超えない(2/3)
File Copy 4096 bufsize 8000 maxblocks
Context Switch
コンキストスイッチ：多い
③の処理と同様に
少数のプロセスが頻繁に
切り替わっている
ロードアベレージ：低い
2010/9/21
立命館大学毛利研究室
16


www.asl.cs.ritsumei.ac.jp
Mouri Lab
④性能が一定の値を超えない(3/3)
File Copy 4096 bufsize 8000 maxblocks
cpu wa
cpu sy
cpu us
IO待ちが発生
2010/9/21
立命館大学毛利研究室
17

各パターンの特徴

①の処理
多数のプロセスが頻繁に
切り替わってCPUを利用
ex) Shell Scripts (8 concurrent)

③の処理
少数のプロセスが頻繁に
切り替わってCPUを利用
ex) Execl Throughput

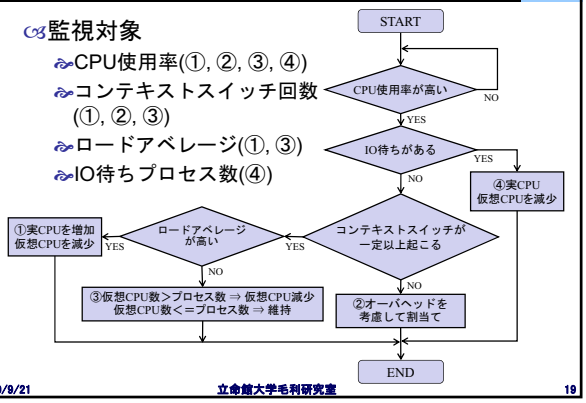
②の処理
特定のプロセスがCPUを占有
ex) Dhrystone 2 using
register variables

④の処理
CPU以外にボトルネックがある
ex) File Copy 4096 bufsize
8000 maxblocks

ハイパーバイザはどのパターンか検知して処理

アルゴリズム

- 監視対象
  - CPU使用率(①, ②, ③, ④)
  - コンテキストスイッチ回数 (①, ②, ③)
  - ロードアベレージ(①, ③)
  - IO待ちプロセス数(④)



プロトタイプ実装

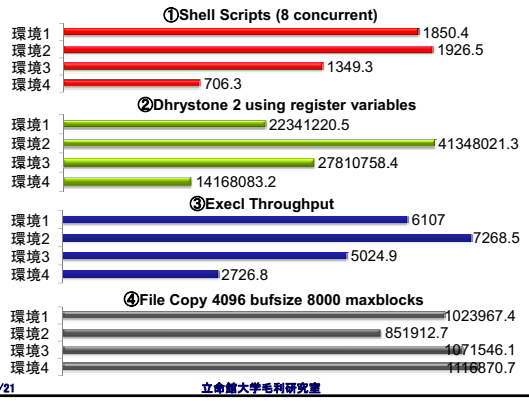
- ホストOSとゲストOSが通信, 増減を判断, xmコマンドで割当てを実行
- ゲストOS側の機構
  - ① 必要なデータの取得(CPU利用率, ロードアベレージ, コンテキストスイッチ回数, IO待ち時間)
  - ② アルゴリズムから資源の不足を判断
  - ③ ホストOSへデータを送信
- ホストOS側の機構
  - ④ ゲストOSからデータを受信
  - ⑤ CPU割当ての判断
  - ⑥ xmコマンドでCPUの再割当て

プロトタイプの性能評価(1/2)

- 目的
  - CPU数を変動させることにより性能を向上させる
- 計測方法
  - UnixBench 5.2.1を使用

- 環境1 : 実CPU数 : 変動(1-3個)
  仮想CPU数 : 変動(1-3個)
- 環境2 : 実CPU数 : 3個, 仮想CPU数 : 3個
- 環境3 : 実CPU数 : 2個, 仮想CPU数 : 2個
- 環境4 : 実CPU数 : 1個, 仮想CPU数 : 1個

プロトタイプの性能評価(2/2)



考察

- 最大数を割り当てなくとも, その時々の処理に合わせて最適な数を割り当てる事で最良の結果に近いパフォーマンスを出せる

処理名	最大値に対するプロトタイプの性能	平均割付コア数
①Shell Scripts (8 concurrent)	96%	2.9
②Dhrystone 2 using register variables	54%	1.6
③Execl Throughput	84%	2.6
④File Copy 4096 bufsize 8000 maxblocks	92%	1.52

- 問題点(②)
  - 実CPUの増減を繰り返すという動作をし, 性能が低下

おわりに

- 協調型仮想計算機システム
  - VMMとOSが協調→効率的な資源の利用が可能
  - 仮想CPU数を動的に変動→全体の処理性能の向上
- 処理を4つに分類し，その時の処理に適応した資源を割りつけ
- プロトタイプ実装・性能評価
  - 一部の状況を除き，最良の結果に近いパフォーマンスを測定できた
- 今後の予定
  - 資源変動アルゴリズムの精査
  - 複数のゲストOSに対応
  - 他のベンチマークで評価

Plan9 を用いた分散組込みシステムの オブジェクト指向プログラミングシステムの提案

盛合 智紀† 並木 美太郎‡
東京農工大学大学院工学府 †/東京農工大学大学院共生科学技術研究院 ‡

1 はじめに

近年の組み込み機器の高性能化，低価格化を背景にセンサーネットワークを始めとした組込み機器を対象とした分散システムが注目を浴びている．大規模システム向けの分散システムを小型の組込み機器にそのまま利用するのは困難であり，新しいプログラミングモデルが求められている．

本研究ではシステム中に大量に設置される組込み機器であるノードの利用者と開発者の双方にとって透過性の高いプログラミングモデルの提供を目的とする．ノードの利用者にはノードの計算機資源を利用するためのファイル I/O ベースのインタフェースを提供し，開発者にはシステム全体を一つのファイルツリーとみなして通信プロトコルやハードウェアの差異を隠蔽したプログラミングモデルを提供する．

上記目的を達成するため，ノードにネットワークファイルプロトコルの 9P プロトコル [1] をサポートした軽量な VM(Virtual Machine) を用いる．VM で動作するバイトコードの生成には ECMAScript をベースに独自拡張を施した言語を利用することで開発者に対しての目的を達成する．

2 分散システムのユーザービュー

本研究では分散システムの通信プロトコルに TCP/IP を利用した 9P プロトコルを用いる．9P プロトコルはアプリケーション層のプロトコルであり，TCP/IP 以外の通信プロトコルにも対応できるメリットがある．ノード上の計算機資源 (周辺装置やバイトコードに記述されたプログラム) を VM が 9P プロトコルを通してファイルに仮想化する．このときノードは小さなファイルツリーを生成することとなり，ノードの生成したファイルツリーをクライアントマシン (Linux や Plan9 など 9P をサポートした計算機) のファイルツリーの一部にリモートマウントすることで利用する．

以下にクライアントマシンからのノードの見え方を表した分散システムのイメージを示す．

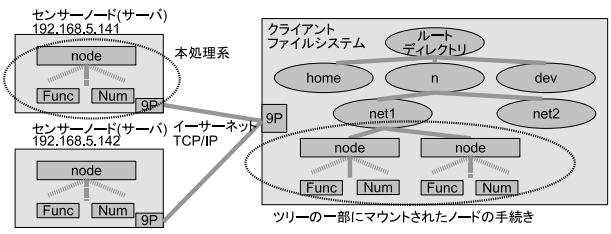


図 1: 利用者から見えるシステムのイメージ

図 1 に示したように，クライアントのファイルツリーの部分木としてノードの計算機資源が取り込まれる．クライアントのファイルツリーの一部に取り込まれたノードの計算機資源は cat コマンド等ファイル I/O を利用し，実行結果を文字列として取得する．利用者は通信プロトコルやノードの所在地を意識することなく，位置透過性やアクセス透過性の高い分散資源管理をクライアント PC の OS のサポートによって実現する．

3 分散システムのプログラミング手法

ノードの開発者は分散システム全体を一つのファイルツリーとみなしたプログラミングモデルを提供する．ファイルツリーへの仮想化に必要なファイル名等の情報はコンパイラから取得し，バイトコードに付加することで解決する．開発言語として ECMAScript をベースにしたものを用い，通信プロトコルのサポートや通信の待ちうけは VM が行い開発者に意識させない．ECMAScript はプロトタイプベースオブジェクト指向言語なので，オブジェクトの動的探索を利用したプロトタイプチェーンで親オブジェクトのプロパティを暗黙的に参照できる特徴が有る．ECMAScript の持つオブジェクト指向的な特徴とプロトタイプチェーンをファイルツリーにマッピングさせることで，ノードを超えた計算機資源を暗黙的に利用することを可能とする．オブジェクト指向言語とファイルツリーの対応関係をまとめる．

表 1: オブジェクト指向言語とファイルツリーの対応

オブジェクト	ファイルツリー
名前空間	ディレクトリ
クラス	ディレクトリ
メソッド	ファイル
プロパティ	ファイル
インスタンス化	ファイルへの仮想化
継承関係	階層関係
クラスとインスタンス	階層関係
プロトタイプチェーン	階層を遡る

表 1 の対応を元に開発者のマシンからシステム全体を一つのファイルツリーとして扱う際のノードの見え方は図 2 のようになる．

図 2 に示したように，開発者も利用者同様ノードをリモートマウントして管理するが，マウントするノードは親ノード一つである．親ノードの下には子ノードがぶら下がる形になり，親ノードのルートディレクトリを頂点としてシステム全体で一つのファイルツリーを形成しているのが見て取れる．親ノードを管理することで，親ノードの VM が一つのバイトコードから子ノードに持たせるバイトコードを自動的に判別する．

子ノードのルートディレクトリは利用者がリモートマウントする際のマウントポイントに対応するディレクトリである．これは利用者がノードの階層を意識せ

Object-oriented Programming System for Distributed Embedded System using Plan9

† Tomoki Moriai
Graduate school of Engineering, Tokyo University of Agriculture and Technology
‡ Mitaro Namiki
Graduate school of Engineering, Tokyo University of Agriculture and Technology

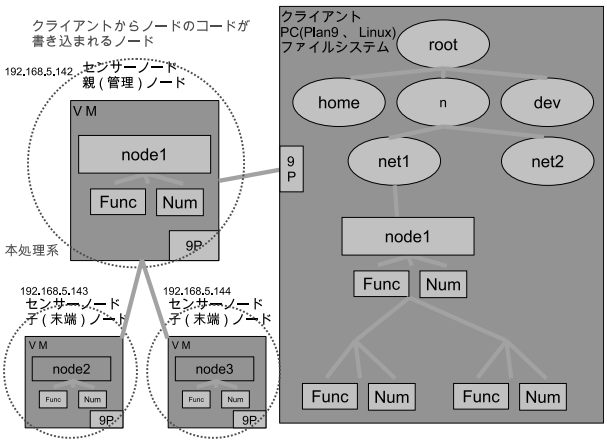


図 2: 開発者から見えるシステムのイメージ

ずに個々のノードをクライアント PC のファイルツリーにリモートマウントする際には見えるが，開発者の意識するファイルツリー上には登場しない．ルートディレクトの下にはバイトコードから読み取った結果，ファイルツリーに仮想化されるオブジェクトがディレクトリとして存在する．各オブジェクトディレクトリの下にはファイルに仮想化される指定を受けた変数がファイルとして存在する．この変数が仮想化されたファイルに I/O を行うことで計算結果を取得したり変数を与えたりする．オブジェクトディレクトリの下にディレクトリが存在する場合，オブジェクトの継承関係を意味しており ECMAScript の場合プロトタイプチェーンの対象となる．よって，個々のノードが提供するファイルツリーは図 3 のようになる．

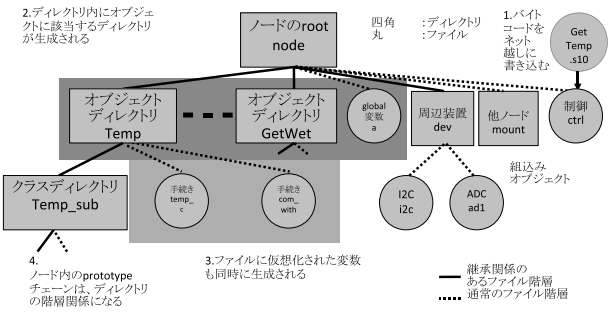


図 3: ノードの提供するファイルツリー

利用者は図 3 で示すツリーを必要個数分ローカルファイルツリーの一部にマウントすることとなる．開発者は親ノードのルートディレクトリを頂点としたツリーを一つマウントして利用する．図 3 で示したオブジェクトディレクトリは継承関係が発生している場合ディレクトリの階層となる．ノードを超えた継承関係も有りえ，親オブジェクトのプロパティやメソッドへのアクセスもノードを超えて可能とする．ノードを超えた継承が本システムの特徴であり，プログラム中でもノードの所在地を意識する必要が無くなり透過性の高いプログラミングモデルを提供する．プロトタイプチェーンを利用したノード間の協調を行うコード例を示す．

図 4 で示したコード中にはオブジェクトをどのノードに配置するかのコードは含んでいない．TempSuper は親ノードに設置され，Temp は複数の子ノードに設置さ

```
/*ファイルに仮想化されない親ノードのオブジェクト*/
function TempSuper() {}
TempSuper.prototype.temps = new Array[num_of_nodes];
TempSuper.prototype.average = function() {
      var temps_sum = 0;
      for (i=0; i<num_of_nodes; i++) {
            temps_sum += temps[i];
      }
      return temps_sum / num_of_nodes;
}

/*ファイルに仮想化される手続きを含む子ノードのオブジェクト*/
function Temp() {}
Temp.prototype = new TempSuper();
/* ファイルに仮想化する変数を指定するvisible */
Temp.visible.temp_c = function () {
      return dev.ad1 * (変換式);
}
/* プロトタイプチェーンの利用例 */
Temp.visible.comp_with_average = function (my_number) {
      temps[my_number] = this.temp_c;
      if ( teis.temp_c > average ) {
            beep();
      }
}

/*ノードとオブジェクトの設置場所の対応もECMAScriptで記述*/
```

図 4: プロトタイプチェーンを利用した例

れることを想定したコードである．Temp では visible を用いることでファイルに仮想化する変数を明示している．図 4 では AD コンバータのデータを利用して温度を取得することを想定している．Temp の comp_with_average で親ノードの Temp にプロトタイプチェーンでアクセスし，子ノードで取得した温度データを記録している．また，全子ノードの平均温度の計算は親ノード内で average として計算させ，計算結果だけを子ノードは利用している．ノードに相手ノードの所在地を意識せずに，あたかも自分のノード内でプロトタプチェーンを利用しているかのようにアクセス透過性と位置透過性の高い方法で協調動作をコーディングできる例である．

4 おわりに

本研究では先に行った試作システムによりファイル I/O ベースのインタフェースの提供によりノードの利用者にとって透過性が高まることを確認した．今回は ECMAScript の利用により，開発者にとって透過性の高いプログラミングモデルによる分散システムの構築法を提案した．

今後の課題としては，ECMAScript の処理系を組込み機器に移植することと，独自拡張を言語処理系に付加することである．また，9P プロトコルの一部など試作で作成したものも移植し，システムを実装・評価する．

参考文献

[1] Bell-labs: Plan 9 File Protocol, 9P http://plan9.bell-labs.com/sys/man/5/INDEX.html (2008)

[2] Arduino Software: Arduino http://www.arduino.cc/ (2005)

[3] Yoshimasa Niwa, Satoru Tokuhisa, Masa Inakage: Talktic: a development environment for pervasive computing applications ACM International Conference Proceeding Series; Vol. 352 pp. 34-41 (2008)

Plan9 を用いた分散組込みシステムのオブジェクト指向プログラミングシステムの提案

Object-oriented Programming System for Distributed Embedded System using Plan9

東京農工大学 大学院 工学府
情報工学専攻 並木研究室

盛合 智紀

1

背景

- 小型組込み機器の低価格・高性能化
  - 演算能力の向上 / 記憶領域の増加
  - 周辺装置の充実
  - ネットワークへの参加
- 情報インフラの拡充
  - 高速な通信回線が何処でも

- 組込み分散システムの構築方法が求められる

2

先行研究

- Arduino (Arduino Software, 2005)
  - Processing/Wiring言語 (C,C++風)
  - 対応アーキテクチャが限られている
  - イーサーネットの通信に関する記述が必要
- Talktic (稲蔭 正彦, 2004)
  - ECMAScript
  - P2Pのアドホックネットワークによる通信
  - PCに接続されたホストノードが必要

3

既存のシステムの問題点

- 分散システムの資源管理
  - 通信プロトコル
    - ソケットを用いた低水準なネットワークプログラミング
  - 分散資源の利用にライブラリが必要
  - その他大規模システムで用いられる分散システムは豊富な計算機資源の利用が前提
    - リッチすぎる機能をサポート
- 組込み機器の開発
  - アーキテクチャ頃に異なる開発環境
  - 周辺装置の利用

4

本研究の目標

- ファイル入出力のプログラミングモデルを提供
  - 通信プロトコルに関する記述をしない
  - 組込み機器上の資源をファイルに仮想化する
- 複数ノードのハードウェアを管理
  - 通信や周辺機器の利用はライブラリに隠蔽する
  - 一つのコードで複数ノードを制御する
- 小型の組込み機器で動作
  - CPU:数10MHz ROM:128KB程度 RAM:数10KB

5

システム構成方法

- Plan9の9Pプロトコルによるファイルへの仮想化
  - プログラムや周辺装置の実行にファイルI/Oを利用
  - ファイルI/Oを用いたネットワークプログラミング
    - クライアント(PC)側プログラムはOSが分散資源管理
    - サーバ(ノード)側プログラムはVMが分散資源管理
- VM(Virtual Machine)を利用
  - 言語処理系のVMによるノードのプログラム実行
  - 9Pプロトコルのサポート
  - ハードウェアの差異を吸収

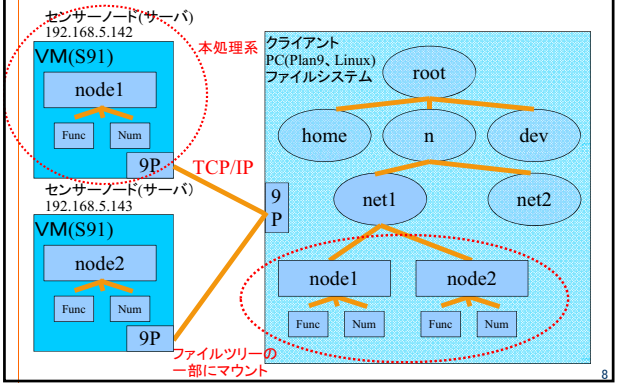
6

ノードの提供する機能

- ノード上で動作するコードをクライアントから取得
- コードのロードとファイルへの仮想化
- ノードを管理するためのインタフェースをファイルに仮想化
- 仮想化したファイルに対するアクセスを解釈し、コードを実行した結果を返す

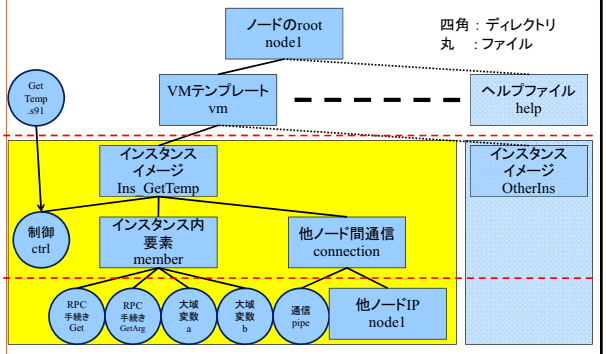
7

システム構築図



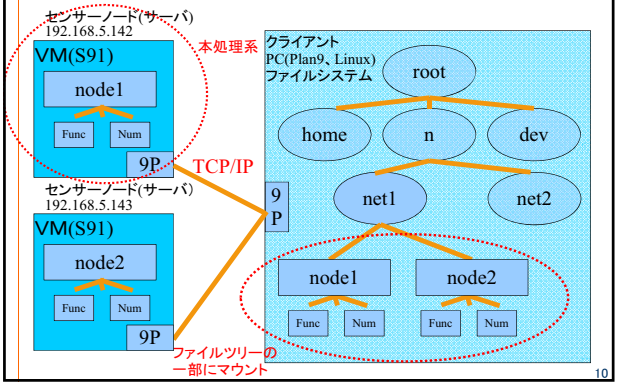
8

S91VMが提供するファイルツリー



9

システム構築図



10

ノードの処理内容の記述方法

```
int a;                          /* グローバル変数 */
int final b = 1;                /* 書き換え不可能 */

/* RPC対象関数 */
setrpc int Get() {
        int temp;

        /* sence_adの呼び出し */
        temp = sence_ad();

        /* RPCの結果として返す */
        return temp;
}
```

```
int sence_ad() {            /* RPC対象外関数 */
        return adc(0);
}

/* 引数付きRPC対象関数 */
setrpc int GetArg(int num) {
        ・・・
}

void main() {
        /* RPC実行時以外の処理 */
        while(1){・・・}
}
```

11

特徴

- 位置透過性の高い資源管理
  - クライアントのファイルツリーの中にノードのファイルが入り込む
- アクセス透過性の高い資源管理
  - ファイルに対するI/O操作がノードの制御と等価
  - 通信プロトコルに関する記述が不要
- ハードウェア非依存な資源管理
  - ハードウェア依存部分は組込み関数等で隠蔽

12

ノードのアクセス例

管理者

```
modprobe 9p
mount -t 9p -o proto=tcp 192.168.5.142 /n/node/ ～ノードのマウント～
mkdir /n/node/node1/vm/Ins_GetTemp ～VMテンプレートの継承～
echo new > /n/node/node1/vm/Ins_GetTemp/ctrl ～インスタンス化～
cat GetTemp.s91 > /n/node/node1/vm/Ins_GetTemp/ctrl
echo start > /n/node/node1/vm/Ins_GetTemp/ctrl ～インスタンスの初期化～
```

利用者

```
cat /n/node/node1/vm/Ins_GetTemp/member/Get ～RPCの実行～
echo 2 > /n/node/node1/vm/Ins_GetTemp/member/GetArg ～引数を与える～
cat /n/node/node1/vm/Ins_GetTemp/member/GetArg ～RPCの実行2～
```

```
～インスタンスの消去、資源の解放～
echo kill > /n/node/node1/vm/Ins_GetTemp/ctrl
```

13

ノードの処理系

- S91VM
  - ノード上で動作するスタックマシン型VM
  - 少ないRAMで動作
  - 9Pプロトコルをサポート
- C--'コンパイラ
  - シンタックスはC言語のサブセット
  - 独自拡張の予約語setrpcで仮想化する手続きを判別
  - 通信や周辺装置の利用をサポートする組込み関数 (9Pプロトコルの記述は不要)

14

C--' とファイルツリーの対応

C--'	ファイルツリー
名前空間 (ソースで一つ)	ディレクトリ (vmテンプレートで一つ)
関数	ファイル
変数	ファイル
関数の宣言	ファイルへの仮想化

15

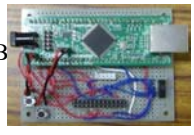
9Pプロトコルの拡張解釈

- 例：vmディレクトリ下への操作
  - open
    - パーミッションの確認ファイルI/Oの準備を行う
  - create
    - ディレクトリまたはファイルを生成する
    - ディレクトリの生成はVMのテンプレートの継承
    - 通常の9Pと異なり、本処理系の独自拡張解釈
  - read
    - ディレクトリ内のファイル情報を列挙する
    - ファイル内のデータを読み出す

16

実装

- ColdFire MCF52233 基盤を用いて試作
  - CQ出版社 Interface誌 2008/9
  - 60MHz ROM：256KB RAM：32KB



- Plan9ランタイムの施策
  - 9Pプロトコルスタックの一部を作成(VMでlibixpを利用)
  - Linuxコマンドのmount、ls、cd、echo、mkdirに対応
  - catコマンドでコードの実行
- S91の移植
  - 動作検証と実行速度の比較

17

試作のまとめ

- ノードの資源をPlan9のノードに仮想化しファイル入出力によるプログラミングモデルを提供

↓

- 通信用のライブラリ・ホストノードが不要
- クライアント/サーバの双方で通信プロトコルに関する記述が不要
- 透過的に分散資源を利用出来る

↓

- 分散資源の利用の敷居が下がる

18

試作の課題と改善方針

- C言語ベースのノードのプログラミング
  - 名前空間がソースコード毎に切れている
    - システム全体を一つのファイルツリーとして管理したい
      - 個々のノードにコードを転送する必要がなくなる
    - 動的探索を用いてより柔軟な資源管理したい
  - ディレクトリの対応関係などファイル階層が不自然
    - オブジェクト指向言語の名前空間やクラスの利用
- ECMAScriptによるノードのプログラミング

19

特徴

- 複数ノードを管理するノードへの操作でシステム全体を管理可能
  - コード転送も一度で良い
- プロトタイプチェーンの利用で他ノードの場所を明示せずに他ノードの計算機資源を活用できる
- VMの利用によりノードのアーキテクチャ依存が無いので管理ノードをより高性能なものにも

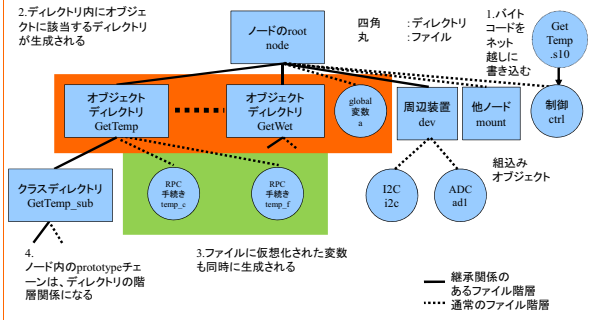
20

OOPとファイルツリーの対応

オブジェクト	ファイルツリー
名前空間	ディレクトリ
クラス	ディレクトリ
メソッド	ファイル
プロパティ	ファイル
インスタンス化	ファイルへの仮想化
継承関係	階層関係
クラスとインスタンス関係	階層関係

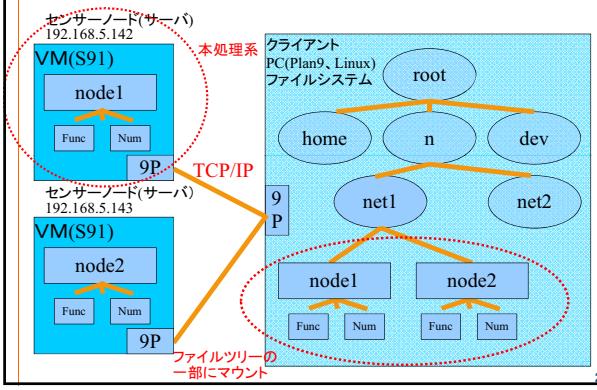
21

VMが提供するファイルツリー



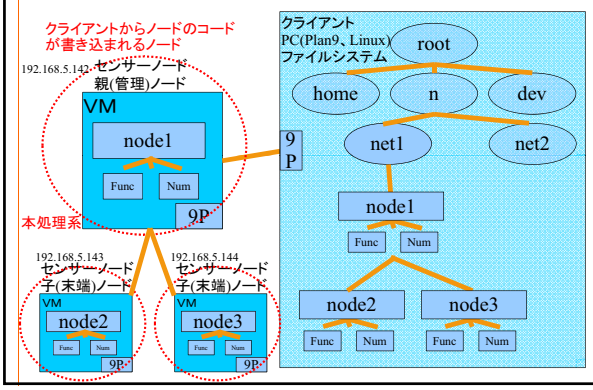
22

利用者から見えるシステム構築図



23

管理者から見えるシステム構成図



24

ノードの処理内容の記述方法

```
var a;                  /*グローバル変数 → ファイルに仮想化*/
/*ファイルに仮想化されないオブジェクト*/
function GetAd() {
        this.get = function () { return ad.get };
}

/*ファイルに仮想化されるオブジェクトのコンストラクタ*/
function GetTemp(var v4ip) {
        this.ip = v4ip;                 /*ファイルに仮想化されるオブジェクトの存在するIP*/
        this.name = "GetTemp";          /*ファイルツリーにおけるディレクトリの名前*/

        this.adc = new GetAd();
        /*ファイルに仮想化される変数に対応する手続き*/
        this.visible.temp_c = adc.get * (変換式);
        this.visible.temp_f = adc.get * (変換式);

}

/*コードの載っているノード上にファイルとして仮想化*/
var my_node_object = new GetTemp(localhost);
/*IPで指定されたノード上にファイルとして仮想化*/
var node2_object = new GetTemp(192.168.5.143);
var node3_object = new GetTemp(192.168.5.144);
```

25

ノードのアクセス例

```
modprobe 9p                                                          管理者

mount -t 9p -o proto=tcp 192.168.5.142 /n/node/ ～ノードのマウント～

echo new > /n/node/node/ctrl ～ノードの初期化～
cat GetTemp.s10 > /n/node/node/ctrl     ～ユーザーコードの転送～

cat /n/node/node/GetTemp/temp_c ～RPCの実行～                        利用者

echo 引数 > /n/node/node/GetTemp/引数に代入される変数 ～引数を与える～

cat /n/node/node/GetTemp/引数が必要なメソッド名 ～RPCの実行2～

～インスタンスの消去、資源の解放～
echo kill > /n/node/node/ctrl
```

26

まとめ

- システム全体の計算機資源を一つのファイルツリーに取り込む
- 自オブジェクト内にないプロパティは親オブジェクト内を検索することで実行する
- 親オブジェクトが他ノードに有る場合も辿ってプロパティを実行する
  - Ex) 親ノードの周辺装置等の資源を利用するメソッドだった場合は取得結果を子ノードが利用できる

27

今後の課題

- 実装
  - VMの性能評価
  - 9Pプロトコルスタックの移植
  - コンパイラの拡張
- 設計
  - 周辺装置や割込みのクラス
  - ノード間のコード転送

28

広域環境での使用に対応した分散ファイルシステム Gfarm の評価

竹川　知孝†

東京農工大学　大学院　工学府　電子情報工学専攻†

1. はじめに

近年，新たなコンピューティングの形態としてクラウドコンピューティングが普及し始めている．広域環境での使用を前提とした使用方法によって，ローカルの計算機で処理をするのではなく，ネットワークの先にあるサーバで処理をする環境が整ってきた．しかし，広域環境でコンピュータを使用する事が容易になった一方で，ネットワーク上に分散したファイルの管理や，増加するデータサイズへの対応など新たな課題が生まれている．

2. 目的

本研究では広域環境での使用に対応した分散ファイルシステム Gfarm の評価を行う．Gfarm は筑波大学の建部氏を中心にオープンソースで開発されているネットワークファイルシステムである．広域環境でスケーラブルなアクセス性能を実現するために複製作成や RTT を元にした最短経路でのファイルの取得といった機能が備わっている．この Gfarm をローカル環境で動作させ NFS と比較する事で Gfarm の利点を示す．また広域環境での使用を想定し，ネットワークエミュレータを使って遅延を発生させた環境で動作させる事で Gfarm を広域環境で使用する利点を示す．

3. 実験

Gfarm は大きく分けて，メタデータサーバ，ファイルシステムノード，クライアントの３つのパートで構成される．実験ではメタデータサーバ，ファイルシステムノードを BUFFALO 社の TeraStation で動作させ，PC をクライアントとしてサーバに接続する事で Gfarm を評価する．

Gfarm の性能を評価するために以下の二つの実験を行った．

実験１　ローカル環境において TeraStation を 1 台，PC を 1 台配置し，これらを Gigabit Ether で接続する．TeraStation をメタデータサーバとファイルシステムノードとし，PC をクライアントとして Gfarm を動作させた場合と NFS(Ver.3)を動作させた場合の転送速度を計測する．転送するファイルは 1MB，10MB，100MB，1GB，10GB のファイルであり，これら以下のコマンドで転送し，転送速度を比較する．

- time dd if=/dev/zero of=/mnt/gfarm/zero.dat bs=1M count=1

図 1　実験 1 の環境

実験２　広域環境を再現するためのネットワークエミュレータと，2 台の TeraStation，1 台の PC を Gigabit Ether で接続する．それぞれの TeraStation をファイルシステムノードとし，一方の TeraStation はメタデータサーバも動作させる．ネットワークエミュレータには往復で 20msec の遅延を設定し，二つの TeraStation 間に置く．このような環境で 1MB，10MB，100MB，1GB，10GB のファイルを以下のコマンドで転送し，書き込み，読み込みの転送速度を算出する．

- time dd if=/dev/zero of=/mnt/gfarm/zero.dat bs=1M count=1
- time dd if=/mnt/gfarm /zero.dat of=/home/takekawa/

図 2　実験 2 の環境

Evaluation of the Gfarm Filesystem Corresponded to Use in the Wide Area Network

† Tomotaka TAKEKAWA

† Graduated School of Computer and Information Sciences, Tokyo Univ. of Agri. and Tech.

4. 結果

表 1　実験 1　Gfarm

	1MB	10MB	100MB	1GB	10GB
平均転送時間(sec)	0.0918	0.4330	4.562	50.01	510.0
平均転送速度(MB/s)	10.9	23.1	21.9	20.0	19.6

表 2　実験 1　NFS(Ver.3)

	1MB	10MB	100MB	1GB	10GB
平均転送時間(sec)	0.1734	0.5140	4.657	47.31	504.8
平均転送速度(MB/s)	5.77	19.5	21.5	21.1	19.8

ローカル環境での Gfarm の転送速度は NFS と比較して大きな差がない事が表 1と表 2から確認できる．1MB のファイル転送時に Gfarm と NFS で差があるのは転送のための準備にかかった時間の差だと考えられる．

表 3　実験 2　遅延 20msec を設定し，ファイルシステムノード B のファイルを読み込む

	1MB	10MB	100MB	1GB	10GB
転送時間(sec)	0.3956	2.441	24.11	259.3	2672
転送速度(MB/s)	2.52	4.10	4.15	3.86	3.74

ネットワークエミュレータを介してファイルを読み込むと転送速度が低下する．10MB 以上のファイルを読み込んだ場合，値が約 4MB/s に収束している事が確認できる．

表 4　実験 2　ファイルシステムノード B からファイルシステムノード A へ複製を作成

	1MB	10MB	100MB	1GB	10GB
転送時間(sec)	1.564	3.428	17.21	171.5	1793
転送速度(MB/s)	0.64	2.92	5.81	5.83	5.58

ネットワークエミュレータを介してファイルシステムノード B から A へ複製を作成する．転送経路は表 3が示すネットワークエミュレータを介したファイル転送と変わらないが，転送速度は向上している．

表 5　実験 2　ファイルシステムノード A から複製を読み込む

	1MB	10MB	100MB	1GB	10GB
転送時間(sec)	0.09534	0.5839	5.464	55.52	562.1
転送速度(MB/s)	10.5	17.1	18.3	18.0	17.8

クライアントはファイルシステムノードへの最短経路を RTT によって計測し，複製が存在するファイルを読み込む時は，もっとも近いファイルシステムノードに接続し，最短経路でファイルを取得する機能が備わっている．このため複製作成後のクライアントはファイルシステムノード A からファイルを取得し，転送速度は最大で約 4.75 倍向上している．

表 6　実験 2　ネットワークエミュレータを介したファイルシステムノード B への書き込み

	1MB	10MB	100MB	1GB	10GB
転送時間(sec)	0.3451	2.139	21.51	222.3	2042
転送速度(MB/s)	3.0	4.9	4.9	4.7	5.1

書き込みにおいてもネットワークエミュレータを介する事で転送速度が低下する．また実験 1 における Gfarm の転送速度の結果はファイルシステムノード A への書き込みと捉える事ができる．表 1と表 6を比較するとネットワークエミュレータを介する事で書き込み速度が約 25%に低下している事が確認できる．

5. 考察とまとめ

ローカル環境での動作では既存のネットワークファイルシステムとして NFS を比較対象にした転送速度を計測した．結果，転送速度に関して Gfarm と NFS で大きな差がない事がわかった．しかし Gfarm の複製作成を用いる事でローカル環境でも障害に強いネットワークファイルシステムを構築できる．

ネットワークエミュレータを用いて広域環境を再現して転送速度を計測した．クライアントがファイルシステムノードとの RTT を考慮して接続する事で，読み込みにおいては最大で約 4.75 倍，書き込みにおいても約 4 倍の転送速度の向上が確認できた．

6. おわりに

本研究では広域環境での使用に対応した分散ファイルシステム Gfarm の評価を行った．ローカル環境と，広域環境をネットワークエミュレータで再現した環境でのそれぞれの評価から，耐故障性，広域環境での位置透過性をもったファイルシステムとして Gfarm が有効である事がわかった．今後はファイルシステムノードの追加や，実際に広域環境での Gfarm の動作を実現したい．

広域環境での使用に対応した分散ファイルシステムGfarmの評価

JSASS2010
東京農工大学　大学院　工学府　電子情報工学専攻　竹川知孝
2010/8/30

目次

- 背景
- 目的
- Gfarm
- TeraStation
- TeraStationでのGfarmの評価
  - NFSとの比較
  - ネットワークエミュレータを用いた評価
- 考察とまとめ
- 今後の予定

背景

- クラウドコンピューティングなど、広域環境で使用可能な共有ファイルシステムの必要性
- データサイズの増加
- ネットワーク上に分散したファイルの管理

目的

広域環境での使用に対応した
ファイルシステムGfarmの有効性を示す

- 広域環境をエミュレーションしてGfarmを動作
- ローカルでGfarmを動作させた場合のNFSとの比較

Gfarm

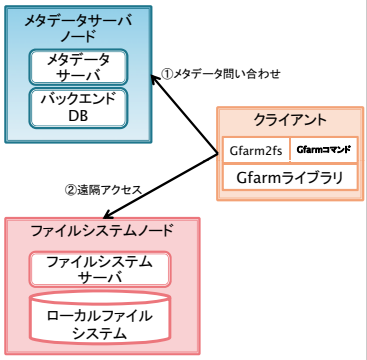
- 筑波大学の建部氏を中心に開発が進められている分散ファイルシステム
- 広域環境でファイル共有が可能
- 大容量，大規模データ処理の要求に応えるスケーラブルなアクセス性能を実現
  - 運用中にサーバの追加が可能
  - ファイルの複製を作成
    - 障害に強い
    - 最短経路でのファイル取得

広域でも性能がスケールアウトするファイルシステム

Gfarm

主に3つのパートで構成される

- メタデータサーバ
  - メタデータサーバはopen，close時だけアクセス
- ファイルシステムノード
  - データアクセスは直接ファイルシステムノードへ
- クライアント



クライアントでの使用

認証

- 共通鍵方式
- GSI方式（Grid Security Infrastructure）

利用方法

- Gfarm コマンドや，Gfarm 並列入出力 API
- gfarm2fs
  - FUSE (Filesystem in Userspace)の機能を利用して，Linux クライ アントから，Gfarm ファイルシステムをマウント

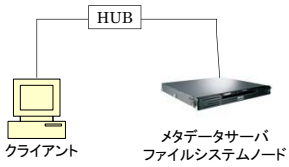
7

TeraStation

- Buffalo社のNAS
- スペック
  - CPU：Marvell MV78100 A0 800MHz
  - メモリ：512MB (Hynix HY5PS12821CFP-S5 DDR2 800 64Mx8 x 8)
  - NAND：512MB (ST Micro NAND04GW3B2DN6 4Gbit)
- 据え置き型とラックマウント型

8

TeraStationでのGfarmの性能評価



- 一台のメータデータサーバ兼ファイルシステムノード
- Gigabit Etherで接続
- クライアントとしてVMware上にFedora13を構築
  - FUSEを使ったgfarm2fsでマウント
- 1MB,10MB,100MB,1GB,10GBのファイルを転送
  - time dd if=/dev/zero of=/mnt/gfarm/zero.dat bs=1M count=1

9

TeraStationでのGfarmの性能評価

	1MB	10MB	100MB	1GB	10GB
1回目 転送時間（sec）	0.08533	0.4352	4.438	49.08	511.0
転送速度（MB/s）	11.7	23.0	22.5	20.4	19.57
2回目	0.1027	0.4290	4.750	49.89	507.1
	9.73	23.3	21.1	20.0	19.7
3回目	0.08739	0.4347	4.499	51.06	511.8
	11.4	23.0	22.2	19.58	19.5
平均	0.0918	0.4330	4.562	50.01	510.0
	10.9	23.1	21.9	20.0	19.6

10

TeraStationでのNFSの性能評価

	1MB	10MB	100MB	1GB	10GB
1回目 転送時間（sec）	0.1436	0.5431	4.766	46.86	501.0
転送速度（MB/s）	6.96	18.4	21.0	21.3	20.0
2回目	0.1700	0.4990	4.716	48.05	502.5
	5.88	20.0	21.2	20.8	19.9
3回目	0.2065	0.4998	4.488	47.02	510.8
	4.84	20.0	22.3	21.3	19.6
平均	0.1734	0.5140	4.657	47.31	504.8
	5.77	19.5	21.5	21.1	19.8

11

GfarmとNFSの比較

- Gfarm

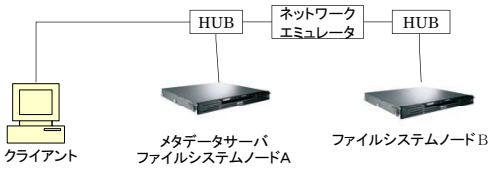
	1MB	10MB	100MB	1GB	10GB
平均転送時間(sec)	0.0918	0.4330	4.562	50.01	510.0
平均転送速度(MB/s)	10.9	23.1	21.9	20.0	19.6

- NFS

	1MB	10MB	100MB	1GB	10GB
平均転送時間(sec)	0.1734	0.5140	4.657	47.31	504.8
平均転送速度(MB/s)	5.77	19.5	21.5	21.1	19.8

12

ファイルシステムノードの追加



- TeraStationを2台用意
- ネットワークエミュレータは上りと下りで10ms、合計20msの遅延を作る

13

読み込み時の評価

- 広域環境での使用を想定し、NEを介したファイルシステムノードBのファイルを取得する

	1MB	10MB	100MB	1GB	10GB
転送時間(sec)	0.3956	2.441	24.11	259.3	2672
転送速度(MB/s)	2.52	4.10	4.15	3.86	3.74

- 複製をファイルシステムノードAに作成

	1MB	10MB	100MB	1GB	10GB
転送時間(sec)	1.564	3.428	17.21	171.5	1793
転送速度(MB/s)	0.64	2.92	5.81	5.83	5.58

同じNEを通過するにも関わらず、Gfarmの方が高速

14

読み込み時の評価

- 複製を作成後、もう一度ファイルを取得する

	1MB	10MB	100MB	1GB	10GB
転送時間(sec)	0.09534	0.5839	5.464	55.52	562.1
転送速度(MB/s)	10.5	17.1	18.3	18.0	17.8

- RTTが短いファイルシステムノードに接続する事で転送速度が向上する
  - 1MB　4.2倍　(2.52MB/s → 10.5MB/s)
  - 10MB　4.17倍　(4.1MB/s → 17.1MB/s)
  - 100MB　4.4倍　(4.15MB/s → 18.3MB/s)
  - 1GB　4.67倍　(3.86MB/s → 18.0MB/s)
  - 10GB　4.75倍　(3.74MB/s → 17.8MB/s)

15

書き込み時の評価

- NEを介したファイルシステムノードBへの書き込み

	1MB	10MB	100MB	1GB	10GB
転送時間(sec)	0.3451	2.139	21.51	222.3	2042
転送速度(MB/s)	3.0	4.9	4.9	4.7	5.1

- ファイルシステムノードAへの書き込み
  - ファイルシステムノードが一つの場合と同じ環境になる

	1MB	10MB	100MB	1GB	10GB
転送時間(sec)	0.0918	0.4330	4.562	50.01	510.0
転送速度(MB/s)	10.9	23.1	21.9	20.0	19.6

16

考察とまとめ

- ローカル環境での使用
  - 転送速度において、GfarmとNFSで大きな差はない
  - 複製機能を用いる事で、障害に強いネットワークファイルシステムを構築できる
- 広域環境での使用
  - クライアントとファイルシステムノードのRTTを考慮して接続
  - 10GBのファイルを転送した時、近くのファイルシステムノードに接続する事で転送速度が4.8倍向上
- Gfarmの特性を生かすには、大規模ネットワーク、広域環境での使用が適している

17

今後の予定

- ファイルシステムノードの追加
- 大規模環境でのGfarmの動作
- 組込機器を対象としたクラウドコンピューティングの実現

18

分散共有メモリを用いたLinuxプロセスの共有

芝 公仁　　小鍛治 翔太
龍谷大学

1　はじめに

プロセッサの性能向上は，これまで，コアの処理速度を向上させることで実現されてきたが，近年，コアの数を増加させることで実現されるようになってきている．これに伴い，複数のコアを使用し効率化を図るアプリケーションが多く開発されるようになった．我々は，このような複数のコアを活用できるアプリケーションを，単一の計算機内のプロセッサだけでなく，複数の計算機のプロセッサを同時に使用できるようにするためのシステムを構築している [1]．本システムでは，複数の計算機からプロセスを共有することを可能とし，共有されるプロセスが任意の計算機を使用して位置透過に処理を行えるようにするプラットフォームを実現する．

本稿では，複数の計算機からプロセスを共有する際に重要になるアドレス空間の共有について述べる．アドレス空間の共有が可能になると，プロセスは共有するどの計算機からでもコードやデータにアクセスすることが可能になり，複数のスレッドを持っていた場合でも，それらを異なる計算機上で同時に動作させることができる．

本システムは，アドレス空間の共有を実現するために，分散共有メモリの技術を使用する．通常の分散共有メモリでは，データのみの共有を行い，プロセッサがコードとしてアクセスするメモリは共有の対象としないことが多い．これに対し，本システムでは，データの共有だけでなくコードの共有も可能としている．これによって，プロセスを実行するスレッドが任意の計算機上で動作することを可能する．また，分散共有メモリの実現はソフトウェアのみで行っており，専用のハードウェアを必要としない．そのため，通常のPC等で使用することができ，既存のPCをそのまま本システムで管理することによって，システム全体の処理性能を向上させることができる．同様に，ソフトウェアも既存のものを使用することができる．本システムによって，アドレス空間の共有を行うと，プロセスが持つ各スレッドは，どの計算機で動作していても，同様のメモリ内容を読み出すことができる．このとき，計算機間のメモリ内容の一貫性制御は本システムが自動的に行うため，各スレッド自体は一貫性制御を行う必要がなく，また，一貫性制御が行われていることを意識する必要もない．そのため，既存のプログラムをそのまま複数の計算機にまたがって動作させることが可能である．

本システムの構築において，アドレス空間全体を共有するためのシステムコールと分散共有メモリの一貫性を維持するための機構を作成した．本システムは，現在，Linuxカーネル2.6.33上に構築されている． 以下，本稿では，本システムの構成，および，アドレス空間の共有手法について述べる．

2　システム構成

本研究の目的は，単一のプロセスを複数の計算機上にまたがって動作できるようにし，そのプロセスが持つスレッドがそれぞれの計算機の資源を効率的に活用できるようにすることである．計算機間でプロセスを共有するために，本システムでは，ソフトウェア分散共有メモリの技術を使用し，ネットワーク上の複数の計算機間でのアドレス空間の共有を実現する．本システムの構成を，図1に示す．

ターゲットプロセスは共有されるプロセスであり，ターゲットプロセスの持つスレッドは，アドレス空間を共有する任意の計算機上で動作可能である． アドレス空間の共有を行い一貫性を維持する制御プロセスは，カーネル内のメモリ操作機構を使用し， ターゲットプロセスのアドレス空間を操作する．また，他の計算機上の制御プロセスと通信を行い，ターゲットプロセスが必要とするデータを他の計算機の制御プロセスから取得する．

制御プロセスは，ターゲットプロセスを監視し，必要に応じて他の計算機上の制御プロセスと通信を行い，ターゲットプロセスが複数の計算機上で実行されているのを意識することなく動作できるような環境を構築する．特に，計算機間でのアドレス空間の共有を実現するために，メモリの一貫性制御を行う．これによって，ターゲットプロセスはどの計算機上でも同一のメモリ内容を読み出すことができ，また，ある計算機上でメモリ内容を変更した場合，それはどの計算機上でも読み出すことができる．

Linuxは，アドレス空間内のメモリをページを単位として管理する．プロセスが利用可能な領域は，プログラムのコードやデータ，また，プログラムの実行に使用されるスタック，ヒープ，スレッド固有のデータ，共有ライブラリが配置される．これらは，単純にプロセスによって管理されるのではなく，どの領域にどのファイルがマップされているか，どの領域がどのように使用されるかなど，Linuxカーネルによって管理される．複数の計算機でアドレス空間を共有するためには，単純にメモリ内容の一貫性制御を行うだけでは十分でなく，アドレス空間の情報も各計算機で同一のものとなるよう制御する必要がある．

制御プロセスは，メモリ操作機構が提供する次の機能を使用して，ターゲットプロセスのアドレス空間を制御する．

- p_mmap(pid, addr, len)

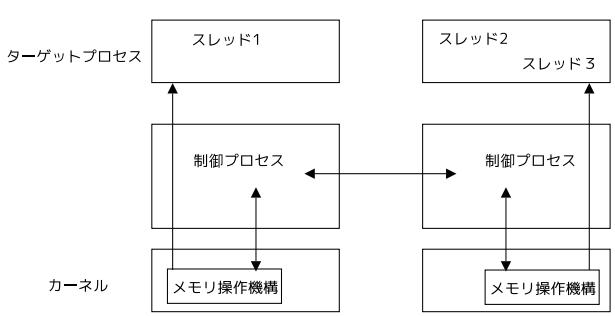


図1　システム構成

- p_munmap(pid, addr, len)
- p_mprotect(pid, addr, len, prot)

p_mmap はプロセス識別子 pid で指定したプロセスが持つアドレス空間のアドレス addr から len バイトの領域を使用可能にする．このとき，当該領域のメモリページには書き込み権も読み出し権も与えられない．p_munmap はプロセス識別子 pid で指定したプロセスが持つアドレス空間のアドレス addr から len バイトの領域を使用不可にする． p_mprotect はプロセス識別子 pid で指定したプロセスが持つアドレス空間のアドレス addr から len バイトの領域のメモリの保護属性を属性 prot に変更する．なお，addr と len はページサイズの整数倍でなければならない．ページの保護に違反するメモリアクセスを行おうとするとページフォールトが発生し，カーネルはこれをメモリアクセス違反のシグナルとしてプロセスに通知する．制御プロセスは，ターゲットプロセスへのシグナルを横取りし，ターゲットプロセスのメモリへの読み書きを検出する．

3 アドレス空間の共有

3.1 一貫性制御

制御プロセスはページの状態を管理し，他の計算機の制御プロセスと協調し，各ターゲットプロセスのアドレス空間に順序一貫性 [2] のメモリモデルを実現する．順序一貫性が実現されるため，ターゲットプロセスはメモリの一貫性制御を意識することなく動作可能である．制御プロセスは，アドレス空間内のメモリをページ単位で管理し，どの計算機でもメモリの読み書きができるよう，ページの複製をそれを必要とする計算機に作成する．このとき，制御プロセスは，メモリの一貫性が保たれるよう，各ページの保護属性を次のいずれかになるように保つ．

- ある計算機でページの状態が読み出し可能であれば，他の計算機で当該ページは読み出し可能か読み書き不可である．
- ある計算機でページの状態が読み書き可能であれば，他のすべての計算機で当該ページは読み書き不可である．

すなわち，メモリの読み出しにおいては，同時に複数の計算機が複製を持つことを可能とし，効率的にアクセスを行えるようにする．また，メモリの書き込みにおいては，ひとつの計算機でのみ行えるようにし，一貫性が保たれるようにする．このように各計算機が持つページの複製を管理することによって，順序一貫性のメモリモデルが実現され，ある計算機でメモリに書き込まれた値は，ただちにどの計算機からでも参照可能になる．これは，同一の計算機内で，複数のスレッドがアドレス空間を共有する時のメモリモデルと同様のものである．

3.2 ページ取得処理

制御プロセスは，他の計算機にページの複製を作成するために，ターゲットプロセスのメモリの内容を読み出したり，他の計算機から取得したメモリ内容をもとにページの複製を作成するために，ターゲットプロセスのメモリに書き込んだりする必要がある．Linux では，ptrace システムコールを用いることによって，プロセスは他のプロセスのメモリを操作することができる．しかし，x86 アーキテクチャの環境において，ptrace システムコールでは，一度に 4 バイトしか読み書きすることができない．1 ページは 4096 バイトであるため，1 ページの読み書きを行うためには，1024 回のシステムコールの発行が必要になる．システムコール呼び出しのオーバヘッドを削減するため，メモリ操作機構は，他のプロセスにメモリをページ単位で読み書きする次の機能を提供する．

- pokepage(src, pid, dst)
- peekpage(pid, src, dst)

pokepage はプロセス識別子 pid で指定したプロセスが持つアドレス空間のアドレス dst に，アドレス src から 1 ページ分のデータを書き込む．peekpage はプロセス識別子 pid で指定したプロセスが持つアドレス空間のアドレス src から 1 ページ分のデータを，アドレス dst の領域に書き込みを行う．これらのシステムコールにより，他のプロセスのメモリを 1 ページ分アクセスする際，システムコールの呼び出し回数が 1 回になる．

pokepage，peekpage による効率化を確認するために，メモリの一貫性制御で行われる，ページの複製の作成に要する時間を測定した．具体的には，1000Mbps のイーサネットで接続された 2 台の計算機 A，B があり，一方の計算機 A が複製を持たないページに対して読み出しを行ったとき，計算機 B からページの内容を取得し，ターゲットプロセス内にそれを書き込みページの複製を作成するのに要した時間を測定した．このとき，計算機 A ではターゲットプロセスのメモリへの書き込み，計算機 B ではターゲットプロセスのメモリからの読み出しが行われる．ptrace を用いて 4 バイトずつ読み書きを行った場合，処理時間は 1.428ms であった．また，pokepage，peekpage で一度にページの読み書きを行った場合，処理時間は 0.258ms であった．このように，ページ単位での読み書きを行えるようにすることによって，大幅に効率化が行えることが確認できた．

4 おわりに

本稿では，複数の計算機からプロセスを共有することを可能とし，共有されるプロセスが任意の計算機の資源を使用して位置透過に動作できる環境を構築するシステムについて述べた．本システムによって，複数の計算機でアドレス空間を共有することができ，プロセスはどの計算機からでも自身のコードやデータにアクセスすることが可能となる．そのため，共有されるプロセスが複数のスレッドを持っていた場合，それらは異なる計算機上で同時に動作することができる．

参考文献

[1] 小鍛治翔太, 芝公仁, 岡田至弘: プロセスを共有するためのソフトウェア分散共有メモリの実現, 第 72 回情報処理学会全国大会論文集, Vol. 72, No. 1, pp. 83–84 (2010).

[2] Lamport, L.: How to Make a Multiprocessor That Correctly Executes Multiprocess Programs, *IEEE Trans. on Computers*, Vol. C-28, No. 9, pp. 690–691 (1979).

分散共有メモリを用いたLinuxプロセスの共有

龍谷大学
芝 公仁, 小鍛治 翔太

内容

- 目的
  - メモリモデル
- システム構成
- 処理手順
  - 初期化，一貫性制御
- Linuxでの実装
- 評価
- まとめ

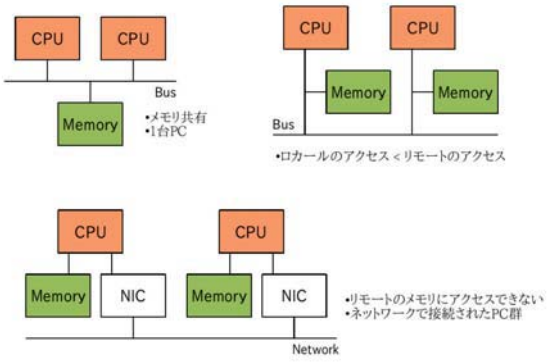
目的

- プロセッサの処理性能の向上がコア数の増加に
- 複数のプロセッサを活用できるアプリケーションの増加
  - データベースサーバ，Webサーバ
  - 動画のエンコード
  - Webブラウザ，ゲーム

目的

- 他の計算機のプロセッサを使用可能に
- 複数の計算機にまたがってプロセスを動作させる
- 既存のアプリケーションを変更せずに

プロセッサとメモリ



目的

- プロセスにとって，複数の計算機のCPUからメモリを共有できるようにする

- プロセスの共有
  - アドレス空間
    - データだけでなくコードも
  - スレッド
  - ファイルなどの資源
  - セマフォ

分散共有メモリ

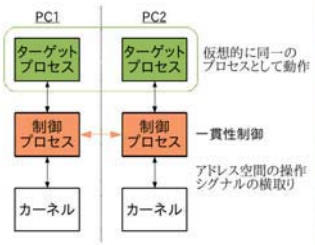
- ソフトウェア分散共有メモリ
  - 複数の計算機にメモリページの複製を作成
  - 計算機間で一貫性制御を行う
- 高速化，効率化
  - 一貫性制御プロトコル，クラスタ
- プロセスマイグレーション

構成

ターゲットプロセス
- 複数の計算機から共有されるプロセス

制御プロセス
- 一貫性制御を行い，共有を実現

カーネル
- アドレス空間を操作する機能を提供



共有するための処理は，すべてターゲットプロセス以外の部分で行う．

制御プロセス

- 各計算機に共有するプロセスの複製を作成
  - アドレス空間などプロセスの資源の共有を実現
- メモリ内容の一貫性制御
  - 順序一貫性
    - ある計算機での書き込みが完了すると，他の計算機からは書き込まれた値が読み出される
    - 同一計算機上でのスレッド間のメモリ共有と同等
    - プロセスは一貫性制御から独立して動作
  - ページに分割して管理
    - 複数の計算機で読み出しのみ可能
    - いずれか1つの計算機でのみ読み書き可能

カーネル

- Linux（x86アーキテクチャ）を使用
  - プロセスが他のプロセスを操作できるよう，機能を一部拡張している
- ptraceシステムコール
  - ターゲットプロセスのメモリを読み書き
  - シグナルの横取り
- アドレス空間を操作するためのシステムコール
  - p_mmap(pid, addr, len)
  - p_munmap(pid, addr, len)
  - p_mprotect(pid, addr, prot)

プロセスの共有

- アドレス空間
  - データだけでなくコードも
- CPUレジスタ
- カーネル内のデータ
  - プロセスの情報，使用している資源の情報
- スレッド
  - セマフォなど同期機構

処理手順

- ターゲットプロセスのメモリマップ
  - 制御プロセスが子プロセスを作成
  - 子プロセスにマップされているファイルをアンマップ
  - 共有プロセスと同様のメモリマップを作成
  - 初期化直後はすべてのページが読み書き不可
- スレッド局所記憶
  - プロセス生成毎にアドレスがランダムに設定される
  - 子プロセスを共有プロセスに合わせる

メモリの一貫性制御

- 読み出し
  - 複製を持たないページを読み書き不可に
  - 読み出しアクセスをシグナルで検出
  - 他の計算機からページの内容を取得
  - 当該ページを読み出し可能にして，スレッドを再開させる
- 書き込み
  - 他の計算機が複製を持つページを書き込み不可に
  - 書き込みアクセスをシグナルで検出
  - 他の計算機の複製を無効化
  - 当該ページを読み書き可能にして，スレッドを再開させる

最後に書き込まれた値が読み出されることが保証される．

プロセス

- テキスト，データ，スタック
- Heap，ライブラリ，VDSO(Virtual Dynamic Shared Object)
- ファイルをマップする形での管理

```
% cat /proc/self/maps
08048000-0804f000 r-xp 00000000 08:01 2596887    /bin/cat
0804f000-08050000 rw-p 00006000 08:01 2596887    /bin/cat
09686000-096a7000 rw-p 00000000 00:00 0          [heap]
b755e000-b7748000 r--p 00000000 08:01 9519717    /usr/lib/locale/locale-archive
b7748000-b7749000 rw-p 00000000 00:00 0
b7749000-b789e000 r-xp 00000000 08:01 4612416    /lib/i686/cmov/libc-2.7.so
b789e000-b789f000 r--p 00155000 08:01 4612416    /lib/i686/cmov/libc-2.7.so
b789f000-b78a1000 rw-p 00156000 08:01 4612416    /lib/i686/cmov/libc-2.7.so
b78a1000-b78a4000 rw-p 00000000 00:00 0
b78b6000-b78b8000 rw-p 00000000 00:00 0
b78b8000-b78b9000 r-xp 00000000 00:00 0          [vdso]
b78b9000-b78d3000 r-xp 00000000 08:01 4612351    /lib/ld-2.7.so
b78d3000-b78d5000 rw-p 0001a000 08:01 4612351    /lib/ld-2.7.so
bff01000-bff16000 rw-p 00000000 00:00 0          [stack]
```

アドレス空間の共有

- スレッド局所記憶(Thread Local Storage; TLS)
  - スレッド固有のデータを保持するメモリ領域
- gsレジスタ
  - セグメントセレクタ
- set_thread_area，get_thread_area
  - Linuxのシステムコール
  - libcはこれを呼び出すラッパー関数を提供していない
  - libc自体はこれを使用する

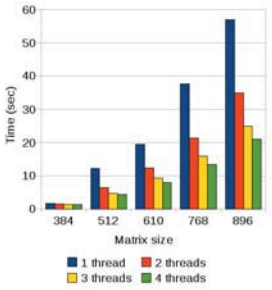
アドレス空間の様々な属性の一貫性制御を行うことで，計算期間での共有を実現

性能評価

- 1つのプロセス内のスレッドを複数の計算機で動作させ，処理時間を測定する
  - n×nの正方行列Aの2乗を求める
  - スレッド数，行列のサイズを変化させる
- 環境
  - Pentium4 2.8 GHz
  - 1000 Mbps Ethernet

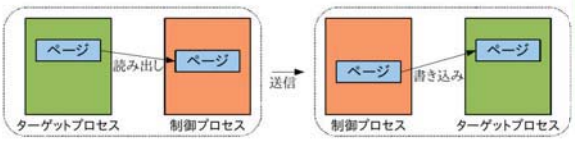
処理時間

- 行列のサイズが増加すると処理時間が長くなる
- スレッド数が増加すると処理時間が短くなる



メモリアクセスの効率化

- 制御プロセスによるターゲットプロセスのメモリの操作
  - 他の計算機に複製を作成するために，メモリページの内容を取得する
  - 取得したメモリページの内容をターゲットプロセスのアドレス空間に書き込む
- ptraceでは4バイト単位
  - 1ページに1024回のシステムコール呼び出しが必要

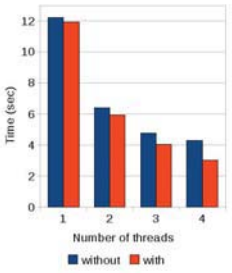


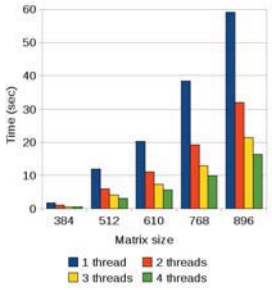
メモリアクセスの効率化

- 制御プロセスによるターゲットプロセスのメモリの操作
  - 他の計算機に複製を作成するために，メモリページの内容を取得する
  - 取得したメモリページの内容をターゲットプロセスのアドレス空間に書き込む
- ptraceでは4バイト単位
  - 1ページに1024回のシステムコール呼び出しが必要
- ページ単位に読み書きできる機能を追加
  - ページ取得時間: 1.43 ms -> 0.26 ms

処理時間

- 処理時間が短くなっている
  - スレッド数が多いほど効果がある





行列サイズ: 512 × 512

まとめ

- 複数の計算機でプロセスを共有する手法
  - 分散共有メモリの技術を使用
  - コードも共有するなどアドレス空間自体を共有
  - Linux(x86)での実装
  - 高速化のためなど，カーネルの機能を拡張

MANETにおけるDHTを用いたデータベースの横断検索

西原 雄太†† 横田 裕介† 大久保 英嗣†
† 立命館大学情報理工学部 †† 立命館大学大学院理工学研究科

1 はじめに

本研究は，MANETを構成する各ノードがデータベースを持っている環境を想定し，それらを横断的に検索する手法を提案する．従来，このような検索は，フラッディング方式を用いた検索によって実現される．フラッディング方式では，柔軟なクエリを処理することができ，各ノードのスキーマが異なる場合にも対応できる．しかし，発生するトラフィックが膨大になるため大規模なネットワークを構築できないことや検索結果入手の不確実性が解決できない．このため，本研究では，DHTを用いることで，スケーラビリティを実現すると同時に検索結果入手の確実性を向上させる検索方法を提案する．

本稿では，2章でインターネット上でノードが保持するデータベースを検索する研究であるPIER(Peer-to-peer Information Exchange and Retrieval) について説明する．3章で提案するMANETにおけるDHTを用いたデータベースの横断検索について述べる．4章で評価について述べる．

2 既存研究

PIER[1] は，データベースをDHTを用いて検索するシステムであり，各ノードが保持しているデータベースが，共通のグローバルなスキーマで定義されている環境を想定している．PIERの検索は，各ノードが保持するデータベースをDHTに登録することによって実現される．データベースのDHTへの登録は，各ノードが保持するデータベースの全タプルについて，put関数を実行することで行う．put関数は，テーブル名，属性名，属性値をまとめてハッシュ化したものをキーとし，所持ノードのIPアドレス，テーブル名，タプル番号といったタプルの所在情報を値としてDHTに登録する関数である．一方，登録した値の取得には，get関数を用いる．get関数は，テーブル名，属性名，属性値をまとめてハッシュ化したキーをDHTに対して問合わせ，キーに登録されているタプルの所在情報をすべて取得する関数である．データベースの検索は，このget関数で取得したタプルの所在情報から各ノードとユニキャストで通信することで行う．

3 提案手法

本章では，2章で述べたPIERをMANETに応用することでデータベースを検索する手法について述べる．これにより，スケーラブルな検索を実現する．しかし，PIERをMANETへそのまま適用した場合，いくつかの問題が生じる．具体的には，インターネットを想定したDHT上に実装されている点と，データベースをDHTに登録する際に発生するトラフィックが多い点である．どちらも，高速で安定した通信が可能なインターネット環境では問題にならないが，低速で不安定な無線によって通信するMANET環境ではネットワークへの負荷が高い．そのため，提案手法では，PIERの仕組みにMANET環境を考慮した改良を加えることで，ネットワークへの負荷を低減する．まず，インターネットを想定したDHT上に実装されている問題は，MANETを想定したDHTであるMADPastry[2]を用いることで解決する．MADPastryは，ネットワークをクラスタに分割してクラスタ内のノードに論理的に近いノードIDを割り振ることにより，ノード間の近接性を考慮したオーバレイネットワークの構築を実現している．次に，データベースをDHTに登録する際に発生するトラフィックが多い問題は，PIERのput関数の変更と，事前に属性値の特徴や発行されるクエリを考慮してデータベースの登録方法を変更し，put/get関数の実行回数の削減，1回のput/get関数で発生するトラフィックを削減して効率的にデータベースを登録することで解決する．

3.1 put関数の変更

PIERのデータベースの登録は，全タプルをDHTに登録している．すなわち，1つのテーブルをDHTに登録する場合，(タプルの総数 × 属性数) 回のput関数を実行する必要がある．そのため，提案手法のput関数では，タプルの所在情報でなく，テーブルの所在情報を登録する．これにより，タプルの挿入が行われた際，テーブル内に同じ属性値を持ったタプルが存在し，そのタプルがすでにput関数によってDHTに登録されている場合は，put関数を実行する必要がなくなる．

3.2 効率的なデータベースの登録方法

提案手法では，データベースに格納される属性値の特徴，およびアプリケーションから発行されるクエリが事前にわかっているものとして，効率的にデータベースをDHTに登録する．まず，発行されるクエリを調べ，検索で利用しない属性はDHTに登録しない．つぎに，提案手法では属性ごとに，属性値の特徴と発行されるクエリによりDHTへの登録方法を選択する．これにより，DHTへ効率的にテーブルの所在情報を登録する．以下に，データベースのDHTへの登録方法を提案する．

複数データを統合して登録 複数のデータを同じハッシュ値として扱い，DHTへテーブルの所在情報を登録する方法である．この方法は，例えば，温度データをDHTに登録する場合，$hash(\text{"テーブル名, 温度, 0"}) = \cdots = hash(\text{"テーブル名, 温度, 9"})$ のように0度から9度までを同じハッシュ値として扱う．これにより，0度から9度までの温度データを1回のput関数の実行によって登録することができる．同様に，0度から9度までの温度データの所在情報を1回のget関数の実行によって取得することができ，レンジクエリを効率よく処理することができる．

クラスタ内にデータを登録 MADPastryによってクラスタに分割されたネットワークを利用してテーブルの所在情報を，テーブルを保持しているノードと同じクラ

Database query processing in MANET using DHT
Yuta Nishihara††, Yusuke Yokota† and Eiji Okubo†
†College of Information Science and Engineering, Ritsumeikan Univ.
††Graduate School of Science and Engineering, Ritsumeikan Univ.

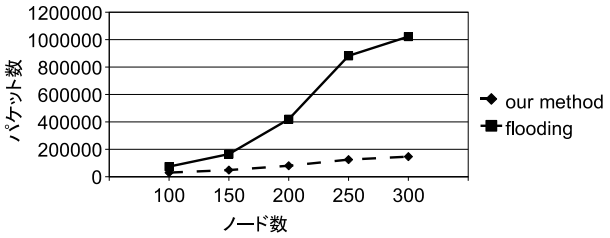


図 1　提案手法とフラッディング方式のパケット数の比較

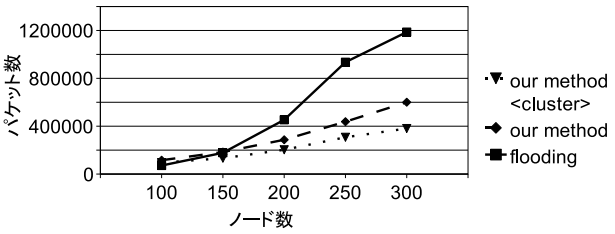


図 2　クラスタ内にデータを登録する手法とフラッディング方式のパケット数の比較

スタ内のノードが管理する方法である．この方法では，例えば，$hash$(”テーブル名, 温度, 0”) を管理するノードがネットワークに複数 (クラスタの個数分) 存在することになる．0 度の温度データを保持しているノードは，テーブルの所在情報を自身が所属しているクラスタ内のノードと通信することで登録する．これにより，1 回の put 関数の実行によって発生するトラフィックを軽減できる．また，データベースを検索してデータを回収する処理もクラスタごとに実行可能となり，トラフィックを軽減できる．

4　評価

本章では，提案手法の評価について述べる．Network Simulator 2[3] にフラッディング方式のデータベースの検索システムと，提案手法を用いたデータベースの検索システムを実装して評価を行った．シミュレーションは，シミュレーション時間を 1000 秒，ポーズ時間を 20 秒とした．また，ノードの移動モデルは Random Waypoint Model を用い，ノードの移動速度は 0 m/s から 1 m/s とした．MANET のルーティングプロトコルは AODV を用いた．

シナリオは，フラッディング方式のデータベースの検索システムと提案手法の双方ともに，事前にノード ID の割り振りやアプリケーションレベルの経路構築は終えているものとする．また，提案手法に関しては，事前にネットワークに存在するすべてのデータベースを DHT に登録し終えているものとする．このような環境において，100 秒間隔でランダムに選ばれたノードがクエリを生成する．クエリの内容によって，該当するデータが格納されているデータベースを保持しているノードが，結果をクエリを生成したノードに返す．

4.1　データーベースの検索

上述した環境で，全体の 5%程度のノードが保持しているデータを要求するクエリを生成させ，提案手法とフラッディング方式のデータベースの検索システムでシミュレーションを行った結果を図 1 に示す．縦軸は，ネットワーク全体で発生した送信パケットの合計を示している．横軸は，ノード数を示している．結果から，フラッディング方式では，ノード数の増加に対して爆発的にパケット数が増えている．一方で，提案手法は，ノード数の増加に対してパケット数の増加を比較的低く抑えることができ，スケーラブルであることがわかる．

4.2　クラスタ内にデータを登録する効果

本節では，クラスタ内にデータを登録する手法を評価する．4.1 節とは違い，全体の 50%程度のノードが保持しているデータを要求するクエリを生成させ，フラッディング方式，提案手法，クラスタ内にデータを登録する提案手法でシミュレーションを行った．結果を図 2 に示す．結果から，フラッディング方式は図 1 と同様に，ノード数の増加に対して爆発的にパケット数が増えている．また，提案手法は，図 1 よりもノード数の増加に対してパケット数の伸びが大きくなっている．これは，クエリを生成したノードがテーブルの所在情報を取得後，ユニキャストでデータを保持している各ノードにクエリを転送する際，物理的に遠いノードと通信することが多いためである．しかし，クラスタ内にデータを登録する提案手法は，ノードの増加に対してパケット数の増加を低く抑えることができている．これは，クエリを生成したノードの代わりに，クラスタごとに存在するテーブルの所在情報を管理しているノードがデータを回収しているため，物理的に遠いノードと通信することが少ないためである．

5　おわりに

本稿では，MANET を構成する各ノードが保持しているデータベースを検索する手法について述べた．本手法は，インターネット上で構築されている P2P ネットワークの各ノードが保持するデータベースを DHT を用いて検索するシステムである PIER の仕組みを利用して，スケーラブルな検索を実現する．評価として，シミュレータを用いて，フラッディング方式のデータベースの検索システムと本手法を用いたデータベースの検索システムを比較した．結果から，提案手法がフラッディング方式よりもスケーラブルであることがわかった．

今後は，データの保持率とクラスタ内にデータを登録する提案手法の効果の関係を明らかにするために，シミュレーションによる評価を引き続き行う．その後，属性値の特徴と発行されるクエリによって DHT への登録方法を選択する指針を示す．

参考文献

[1] R. Huebsch, J.M. Hellerstein, N. Lanham, B.T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pp. 321–332. VLDB Endowment, 2003.

[2] T. Zahn and J. Schiller. MADPastry: A DHT substrate for practicably sized MANETs. In *Proc. of ASWN*, 2005.

[3] The Network Simulator - ns2. `http://www.isi.edu/nsnam/ns/`.

MANET環境におけるDHTを用いたデータベースの問合せ処理

立命館大学院 理工学研究科
大久保・横田研究室
西原 雄太

1

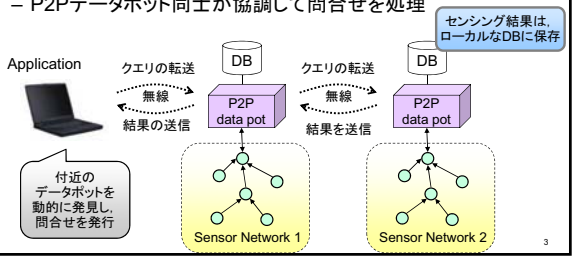
目次

- はじめに
  - MANETを構成する各ノードが保持するデータベースを横断的に検索する問合せ
- PIER
  - インターネット上のP2Pネットワークのノードが持つデータベースをDHTを用いて検索するシステム
- 提案手法
  - PIERをMANETに応用して，データベースを検索
    - MANETを想定しているDHTを利用
    - 効率的なデータベースの登録方法
- 評価
- おわりに

2

はじめに

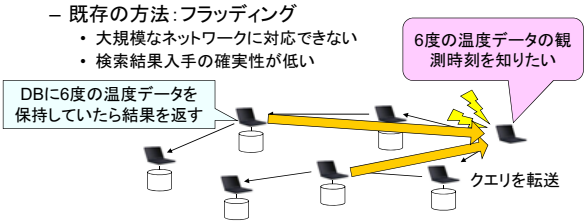
- MANET (Mobile Ad-hoc Network)
  - モバイル端末同士が動的にネットワークを構築
  - 既存のインフラに依存せず容易・安価に構築可能
- P2Pデータポット：環境観測システム
  - P2Pデータポット同士が協調して問合せを処理



3

はじめに（2）

- 各ノードが保持するデータベースを検索する問合せ
  - 既存の方法：フラッディング
    - 大規模なネットワークに対応できない
    - 検索結果入手の確実性が低い



→分散ハッシュテーブル（DHT）を用いることで解決
  - 特定のサーバを用いないような環境において，高速な検索を可能とする技術
  - ✓PIER [Huebsch03]の仕組みをMANETに応用

4

PIER [Huebsch03]

- インターネット上のP2Pネットワークのノードが持つデータベースをDHTを用いて検索
  - ノードが保持するデータベースをDHTに登録 (put)することで効率的な検索を実現
- put関数
  - テーブル名，属性名，属性値をまとめたハッシュ値にタプルの所在情報を登録する関数

put (hash(“ テーブル名;属性名; 属性値”) , “所在情報”)

- get関数
  - DHTに登録してあるタプルの所在情報を取得

get(hash(“テーブル名;属性名;属性値”)) → 該当するタプルの所在情報

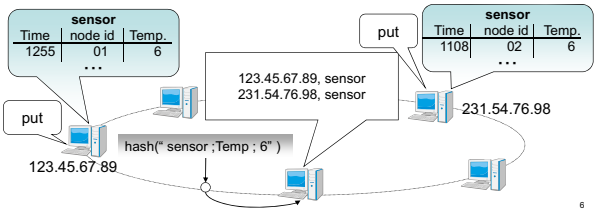
5

PIERの仕組み(1)

- put関数でデータベースをDHTに登録

put (hash(“ sensor;Temp; 6”) , “123.45.67.89,sensor”)
put (hash(“ sensor;node id; 01”) , “123.45.67.89,sensor”)
put (hash(“ sensor;Time; 1255”) , “123.45.67.89,sensor”)

put (hash(“ sensor;Temp; 6”) , “231.54.76.98,sensor”)
put (hash(“ sensor;node id; 02”) , “231.54.76.98,sensor”)
put (hash(“ sensor;Time; 1108”) , “231.54.76.98,sensor”)

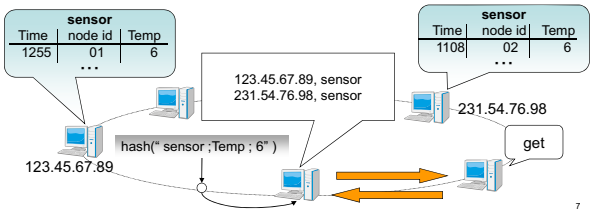


6

PIERの仕組み（2）

- get関数でタプルの所在情報を取得

get (hash(“ sensor;Temp; 6”)) →
“123.45.67.89, sensor”, ”231.54.76.98, sensor”



7

PIERをMANETに応用する際の問題

1. インターネットを想定したDHTを利用
   - 物理ネットワークのトポロジを考慮しない
     - 冗長な経路を経由
2. データベースの登録・検索によって発生する
   トラフィックが多い
   - (タプルの総数×属性数)回のput
   - タプルの挿入のたびに(属性数)回のput
- MANETは，インターネットと比べて低速で不安定

→ トラフィックの削減が必要

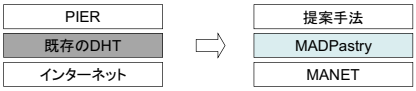
8

PIERの問題点の解決方法（1）

1. インターネットを想定しているDHTを利用

→ MANETを想定したDHT（MADPastry）を利用

- MADPastry
  - 物理ネットワークを考慮してトポロジを構築するDHT
    - 無駄なホップを削減
  - 物理的に近いノードに論理的に近いノードIDを割り振る
    - ネットワークをクラスタに分割
    - クラスタ内のノードに論理的に近いノードIDを割り振る
      （クラスタ内のノードのノードIDに共通のプレフィックスを与える）



9

PIERの問題点の解決方法（2）

2. データベースの登録・検索によって発生する
   トラフィックが多い

→ 効率的にデータベースを登録

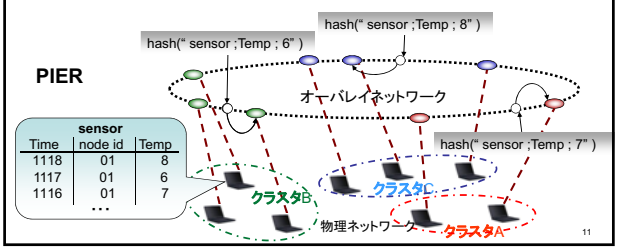
- 事前にデータベースや発行されるクエリを調査
  - 登録する必要のない属性は登録しない
    - 検索しない属性は登録しない
  - 属性値 / クエリの特徴によって属性ごとに登録方法を選択
    A) 複数データを統合して登録
    B) クラスタ内にデータを登録

10

A) 複数データを統合して登録

- 複数のデータを同じハッシュ値として扱う
  - put / get回数を削減
    - 1回のputで複数のデータを登録
    - 1回のgetで複数のデータを取得

→センシングデータを効率的に登録
→レンジクエリを効率的に処理

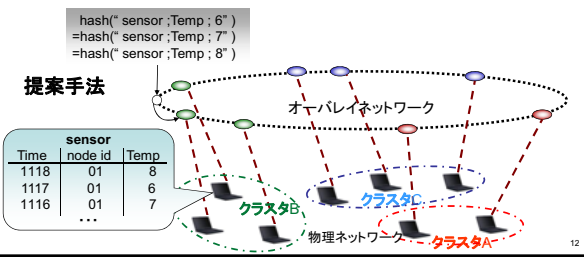


11

A) 複数データを統合して登録

- 複数のデータを同じハッシュ値として扱う
  - put / get回数を削減
    - 1回のputで複数のデータを登録
    - 1回のgetで複数のデータを取得

→センシングデータを効率的に登録
→レンジクエリを効率的に処理

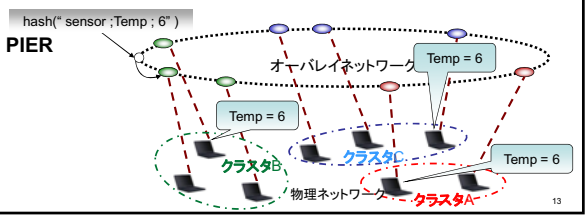


12

B) クラスタ内にデータを登録

- 自ノードが所属するクラスタ内のノードがデータを管理
  - 物理的に近いノードにデータの所在情報を登録
    - 1回のputで発生するトラフィックを削減

  →同じ属性値を多くのノードが保持している場合に有効

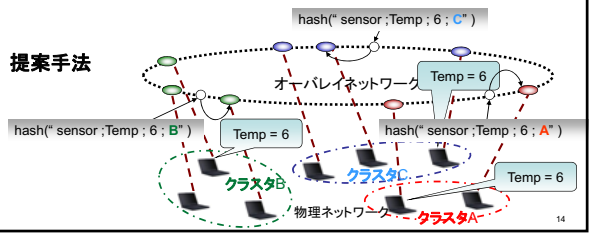


13

B) クラスタ内にデータを登録

- 自ノードが所属するクラスタ内のノードがデータを管理
  - 物理的に近いノードにデータの所在情報を登録
    - 1回のputで発生するトラフィックを削減

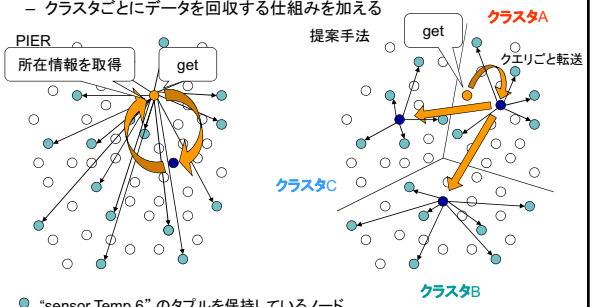
  →同じ属性値を多くのノードが保持している場合に有効



14

B) クラスタ内のデータの回収

- データ回収の効率化
  - クラスタごとにデータを回収する仕組みを加える



- “sensor,Temp,6” のタプルを保持しているノード
- hash(“ sensor; Temp ; 6”)を管理するノード
- クエリを生成したノード

15

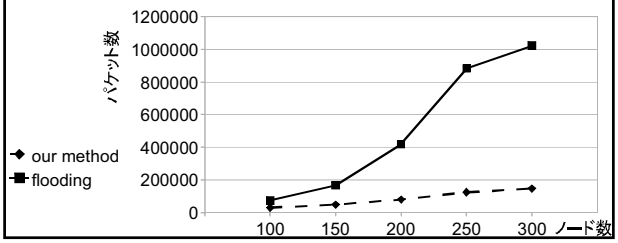
評価

- Network Simulator 2を利用
  - フラッディング，提案手法を用いた検索システムを実装
- 評価項目
  - ネットワーク全体で発生するパケット数の合計
- シミュレーション環境
  - 移動モデル：Random Waypoint Model
  - 移動速度：0 m/s ～1 m/s
  - ルーティングプロトコル：AODV
  - 事前にノードIDの割り振り，経路構築，データベースのDHTへの登録を完了
  - クエリを100秒間隔で9回生成

16

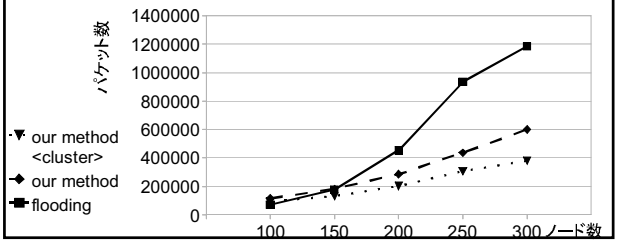
データベースの検索

- 全体の5%程度のノードが保持しているデータを要求
  - フラッディング
    - 全ノード (N個) と通信
  - 提案手法 ：
    - { 平均logN個のノード(DHTの検索) + 0.05×N個のノード }と通信



クラスタ内にデータを登録する効果

- 全体の50%程度のノードが保持しているデータを要求
  - 提案手法：
    - { 平均logN個のノード(DHTの検索) + 0.5×N個のノード }と通信
  - 提案手法 (cluster)：
    - {平均logN個のノード(DHTの検索)×16 + 0.5×N個のノード}と通信
    - 物理的に遠いノードと通信することが少ない



おわりに

- 発表内容
  - PIERをMANETに応用してデータベースの検索
    - MANETを想定したDHTであるMADPastryの利用
    - 事前にデータの特徴や発行されるクエリを考慮してデータベースの登録
  - 評価
- 今後の予定
  - 引き続き，シミュレーションによる評価を行う
    - 複数データを統合して登録する手法の評価
    - PIERと比較
  - 属性値の特徴と発行されるクエリによってDHTへの登録方法を選択する指針を示す

19

CPU と GPU を考慮した行列積計算の負荷分散手法

田邉 克哉 †　　桑原 寛明 ‡　　國枝 義敏 ‡
† 立命館大学大学院理工学研究科 ‡ 立命館大学情報理工学部

1　はじめに

高性能計算の分野では Graphic Processing Unit(GPU) を汎用計算に用いる General-purpose computing on GPU(GPGPU)[1] が注目されており，1 台の筐体に複数の GPU を積んだ計算機が盛んに利用されている．このような計算機の性能を引き出すには問題を適切な大きさに分割し，各プロセッサに負荷を分散する必要がある．しかし，GPU の利用時にオーバーヘッドが発生するため最適な割合で問題を分割することは困難である．

そこで，我々は CPU と複数の GPU へ最適な割合で問題を分割し割り当てることで両者の性能を引き出す Basic Linear Algebra Subprograms(BLAS)[2] を開発している．BLAS は，ベクトルと行列の基本演算を行う代表的な数値計算ライブラリであり，各プロセッサに最適化された実装が配布されている．多くの科学技術計算用アプリケーションは高速化のために内部で BLAS を利用している．そのため，BLAS の実行が高速化できれば，さまざまなアプリケーションの性能向上が期待できる．

本稿では，BLAS 中の一般行列積計算における負荷分散手法について述べる．提案手法では与えられた行列サイズから実行時間を予測するモデルを構築し，それを基に CPU と GPU へ最適な割合で負荷を分散する．これにより，任意の問題サイズで行列積計算を高速に実行することができる．

2　GPGPU の性能特性

GPU は多くの演算器と高速なメモリを搭載し，高い性能を持っている．しかし，GPU で演算を行うには PCI-e のバスを介してメインメモリから GPU 側のメモリへデータを転送する必要がある．これは，GPU を利用するためのオーバーヘッドとなる．転送速度は GPU の性能と比べ極めて遅い．また，複数の GPU に対して同時に通信するとさらに転送速度が低下する．

このオーバーヘッドが存在するために，転送量に比べ演算量が少ないような問題で GPU よりも CPU の方が高速になることがある．そのため，行列積のような単純に問題を分割できるような問題においても，プロセッサの理論性能比で問題を割り当てることが最適ではない．

3　行列積計算の負荷分散手法

本研究では行列積計算の実行時間を予測するモデルをもとに最適な割合で CPU と GPU に負荷を分散する手法を提案する．

3.1　CPU と GPU を併用する行列積計算

本研究で行う行列積計算は BLAS の実装に則り，$C = \alpha AB + \beta C$(A, B, C は行列, α, β はスカラ) とする．CPU と GPU を併用して行列積を解くには問題を分割する必要がある．本研究では行列 B および C を列方向に分割し，CPU ならびに各 GPU に分散し，部分行列を計算する方法を採った．

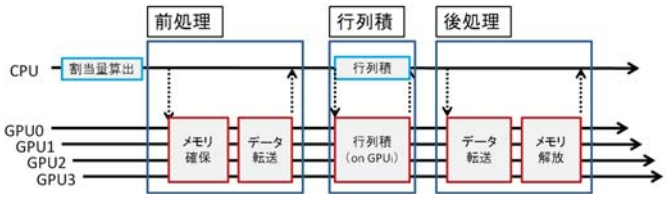


図 1　CPU と GPU を併用した行列積計算の処理手順

図 1 に計算の処理手順を示す．GPU で処理を行うには実行の準備を行う前処理と実行後の後始末を行う後処理が必要である．現在の実装では前処理と後処理の実行中に他の処理を行うことができない．そのため，CPU 側の行列積を前処理や後処理とオーバーラップさせずに処理を進める．

図に示すとおり，実行時には初めに 3.2 節で説明する割当量の決定方法をもとに CPU と GPU への割当量を算出する．次に，GPU 側メモリの確保，データの転送をまとめた前処理を行う．前処理終了後，決定した割当量をもとに CPU と GPU で負荷を分散し行列積計算を行う．計算終了後，GPU 側のメモリからメインメモリへデータを書き戻しメモリを解放する．CPU と GPU で行われる行列積計算は既存の各プロセッサ向け BLAS を用いる．

3.2　最適な割当量の決定方法

本手法では行列 B を 64 列ごとに分割し，探索する必要があるすべての組み合わせの割当量で実行時間を予測する．予測した実行時間が最小となる組み合わせを最適な割当量とする．大きな行列の場合は最適な割当率が理論性能比付近に存在する．この性質を利用し，探索範囲を限定することで割当率の決定時間を高速化する．

ここで，64 列ごとに扱う理由は，GPU 側行列積計算において行列のサイズが 64 の倍数で高い性能を発揮するためである．

3.3　実行時間予測モデル

本研究で構築した実行時間予測モデルは行列サイズと各 GPU への割当量から行列積計算全体の実行時間を予測する．モデルでは行列積の実行時間を前処理と後処理，行列積に分割し，それぞれの実行時間を計算量の一次式に近似している．一次式の係数，切片は実行環境に依存する定数である．事前に複数の大きさの正方行列を用いた行列積計算の実行時間を基に最小二乗法で求めている．モデル中で使用する変数を表 1 に，定数を表 2 に示す．

ここで，変数 n と ng_i，nc の間には等式 1 の関係が成り立つ必要がある．

$$\sum_{i=1}^{GPU\ 数} ng_i + nc = n \tag{1}$$

A Load Balancing Method of Matrix Multiplication among CPU and GPUs

Katsuya Tanabe†, Hiroaki Kuwabara‡ and Yoshitoshi Kunieda‡
†Graduate School of Science and Engineering, Ritsumeikan Univ.
‡College of Information Science and Engineering, Ritsumeikan Univ.

表 1　モデル中の変数

m	行列 A と C の行数
n	行列 B と C の列数
k	行列 A の列，行列 B の行数
ng_i	i 番目の GPU へ割り当てた列数
nc	CPU へ割り当てた列数

表 2　モデル中の定数

pre_i	i 番目の GPU の前処理の係数
$pre\_startup_i$	i 番目の GPU の前処理の切片
$post_i$	i 番目の GPU の後処理の係数
$post\_startup_i$	i 番目の GPU の後処理の切片
$gemm\_cpu$	CPU 側行列積の係数
$gemm\_cpu\_startup$	CPU 側行列積の切片
$gemm\_gpu_i$	i 番目の GPU の行列積の係数
$gemm\_gpu\_startup_i$	i 番目の GPU の行列積の切片

全体の実行時間は式 (2) となる．

$$T_{total} = T_{pre\_total} + T_{gemm\_max} + T_{post\_total} \quad (2)$$

前処理にかかる時間は各 GPU の前処理にかかる時間の和となり，式 (3) で表せる．これは今回の実装で複数の GPU へ並行して通信を行うことができないためである．また，各 GPU での前処理にかかる時間は行列の要素数に関する一次式であり，式 (4) と書ける．

$$T_{pre\_total} = \sum_{i=1}^{GPU\ 数} T_{pre}(m \cdot k + k \cdot ng_i + m \cdot ng_i, i) \quad (3)$$

$$T_{pre}(x, i) = x \cdot pre_i + pre\_startup_i \quad (4)$$

後処理も前処理と同様に各 GPU で並行して実行できない．そのため，後処理全体の実行時間は式 (5) となる．また，各 GPU での後処理の実行時間は式 (6) と書ける．

$$T_{post\_total} = \sum_{i=1}^{GPU\ 数} T_{post}(m \cdot k + k \cdot ng_i + m \cdot ng_i, i) \quad (5)$$

$$T_{post}(x, i) = x \cdot post_i + post\_startup_i \quad (6)$$

式 (7) で求める行列積の実行時間 T_{gemm} は CPU と各 GPU での行列積の実行時間の最大値となる．これは，行列積が各プロセッサで並行して実行できるためである．また，CPU と GPU 単体での行列積の実行時間は式 (8), (9) となる．ここで式 (8)，(9) は行列積計算の核となる 3 重ループの総繰り返し回数に関する一次式である．

$$T_{gemm\_max} = max\left(T_{gemm\_cpu}(m \cdot k \cdot nc), max_i\left(T_{gemm\_gpu}(m \cdot k \cdot ng_i, i)\right)\right) \quad (7)$$

$$T_{gemm\_cpu}(x) = x \cdot gemm\_cpu + gemm\_cpu\_startup \quad (8)$$

$$T_{gemm\_gpu}(x, i) = x \cdot gemm\_gpu_i + gemm\_gpu\_startup_i \quad (9)$$

4　評価

提案した行列積の負荷分散手法に対して実行時間の予測精度と行列積の実行性能を評価する．また，評価では倍精度で計算を行った．使用した環境を表 3 に示す．

4.1　実行時間の予測精度

1 辺が 2048 要素の正方行列積を CPU と GPU1 基 (GTX480) で計算し，GPU への割当量を変化させたときの予測実行時間と実測値を図 2 に示す．予測実行時間と実測値の差は平均で実測値の 1.6%，最大で 4% と小さな誤差で実行時間を予測できた．

表 3　評価環境

CPU	Xeon W3680
GPU	GTX480 GTX285 GTX460 GTX260
メモリ	6GB
OS	Cent OS 5.4(2.6.18)
BLAS(CPU)	GotoBLAS2 1.13
BLAS(GPU)	CUBLAS 3.1

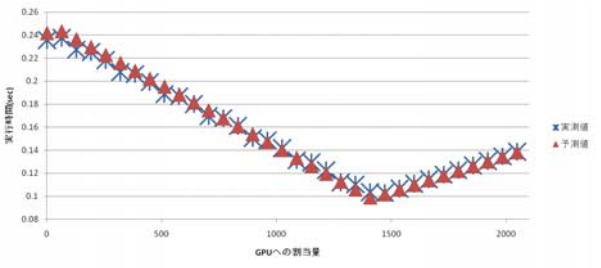


図 2　実行時間の予測精度

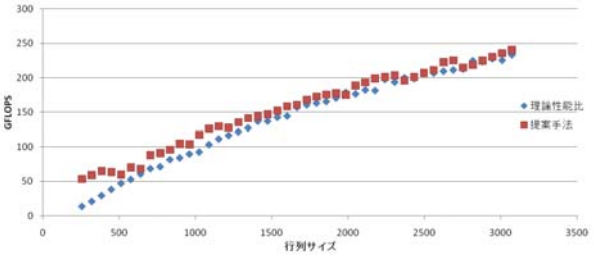


図 3　行列積の処理速度

4.2　行列積の実行性能

評価環境に搭載されている GPU4 基を用い，単純にプロセッサの理論性能比で割り当てた場合と提案手法で割当量を決定し計算した場合の処理速度を図 3 に示す．理論性能比で割り当てた場合と比べ提案手法は全体的に高い性能を記録した．特に問題サイズが小さい場合に効果が大きい．これは，GPU を利用するオーバーヘッドと比べ計算量が少ないため提案手法では GPU を利用しない，あるいは利用台数を少なくするという判断がされ，その効果が効いているためであると考えられる．

しかし，行列サイズが多きくなるにつれて差が小さくなっている．これは，CPU 側の行列積計算が GPU との通信時間とオーバーラップできない実装であるため，理論性能比が最適な割当量に近いためである．

5　おわりに

本稿では，CPU と GPU を考慮した行列積計算の負荷分散手法を提案した．利用する GPU 数を問題サイズにより変化させたことにより小さい問題サイズの場合に，プロセッサの理論性能比で問題を分割した場合と比べて高い性能を出すことができた．

今後は GPU への通信と CPU 側の行列積計算をオーバーラップさせる実装を行い，評価を行う．さらに他の BLAS 演算の実装，最適分割率の予測時間の削減などを行っていく．

参考文献

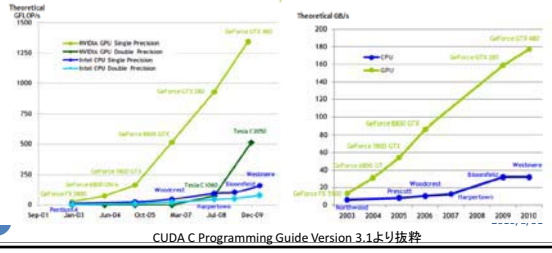
[1] General-Purpose Computation on Graphics Hardware: http://www.gpgpu.org/.
[2] BLAS (Basic Linear Algebra Subprograms)：http://www.netlib.org/blas/.

CPUとGPUを考慮した行列積計算の負荷分散手法

立命館大学大学院 理工学研究科
高性能計算機ソフトウェアシステム研究室
田邉 克哉　桑原 寛明　國枝 義敏

研究背景

- General-purpose computing on GPU（GPGPU）
  - 画像処理用以外の汎用計算にも利用
  - 高い並列性と広いメモリバンド幅
  - CPUと並列に利用可能



CUDA C Programming Guide Version 3.1より抜粋

研究背景

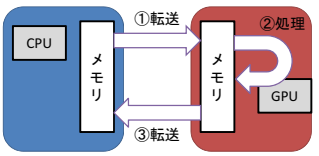
- HPC分野ではGPGPUが盛んに用いられている
  - 高い計算能力が必要
  - 複数のGPUを利用
- 既存のアプリケーション
  - 数値計算ライブラリを利用している
    - ライブラリの性能が実行時間に大きな影響
    - BLAS、FFT、LAPACK
    - CPUとGPUを効率的に利用するものはない

⇒ CPUと複数のGPUを効率よく使うライブラリが必要

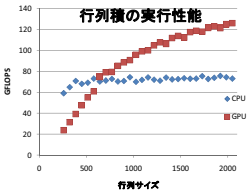
GPUの性能特性

- GPUを計算に用いるにはオーバーヘッドがかかる
  - GPUへのデータ転送が発生
  - 計算量の小さな問題⇒CPUの方が高速

⇒プロセッサの理論性能比で分割するだけではだめ



①③は極めて遅い（8GB/sec）
複数のGPUに同時に行うと性能低下



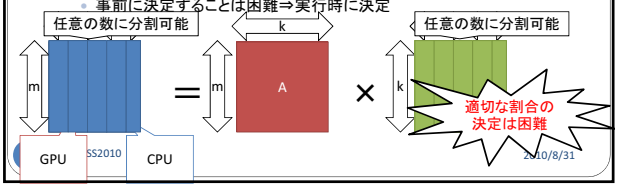
目的と提案

- 目的
  - GPU実行中のCPUを有効活用する
  - 既存のアプリをGPUを用いて高速化する
- 提案
  - CPUと複数のGPUを効率的に利用するBLAS
    - 実行時間を予測するモデルを構築
    - 構築したモデルをもとに実行時にCPUと各GPUの処理量の最適な割当量を求める
    - 一般行列積について評価、報告
      - C=αAB+βC

⇒BLASを利用するアプリの性能向上を図る

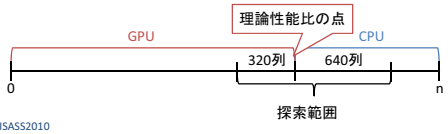
CPUとGPUを考慮したBLAS

- Basic Linear Algebra Subprograms
  - ベクトルと行列に関する基本線形代数操作を行う
  - 行列ベクトル積：y = αAx+βy
  - 行列積：C= αAB+βC
- CPUとGPUの性能を引き出すには
  - 問題を分割しCPUとGPUへ割り当てる
  - 最適な割当量を選択する必要（nを適切に分割）
    - 事前に決定することは困難⇒実行時に決定



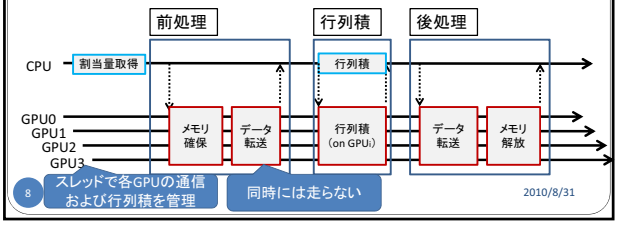
割当量の決定方法

- 問題サイズと割当量から実行時間を求めるモデルを構築
  - 割当量を変化させ実行時間を予測
    - 64列を単位として変化
  - 最適な割当量＝予測実行時間が最小となったもの
- 任意の行列サイズに対応
  - 高速化のために理論性能比周辺だけを探索
    - 経験則として理論性能比の点から下に320、上に640探索



実装

- CPU・GPU BLASのラッパとして実装
  - 実行時に最適な割当量を予測
  - 予測した割当量でBLASを呼び出す
    - CPU： GotoBLAS
    - GPU： CUBLAS



実装上の課題

- 通信のオーバーラップ
  - 問題
    - GPUへの通信時に他のスレッドが実行されなくなる
    - GPUへの通信時にすべてのスレッドが同一のCPUコアで実行される
    ⇒ CPU側の行列積とGPUへの通信をオーバーラップできない
  - 対策
    - プロセッサー・アフィニティの設定
    - GPUをプロセスで管理

実行時間予測モデル

- 予測実行時間 = 前処理 ＋ 行列積 + 後処理
  - 前処理 = $\sum_{i=1}^{\text{GPU数}} T_{pre}(m\cdot k + k\cdot ng_i + m\cdot ng_i, i)$
  - 行列積 = $\max(T_{gemm\_cpu}(m\cdot k\cdot nc), \max_i(T_{gemm\_gpu}(m\cdot k\cdot ng_i, i)))$
  - 後処理 = $\sum_{i=1}^{\text{GPU数}} T_{post}(m\cdot k + k\cdot ng_i + m\cdot ng_i, i)$

ここで $T_{pre}, T_{gemm\_cpu}, T_{gemm\_gpu}, T_{post}$ ：処理量の1次式

⇒係数と切片は最小二乗法で事前に決定

m	行列AとCの行数
n	行列BとCの列数
k	行列Aの列，行列Bの行数
ngi	i番目のGPUへ割り当てた列数
nc	CPUへ割り当てた列数

評価

- 評価項目
  - モデルの精度検証
  - 最適割当量の探索時間
  - 実行性能

倍精度で評価

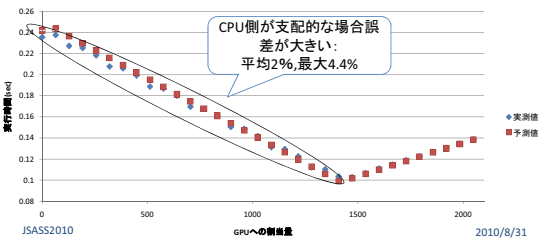
- 評価環境
  - CPU: Xeon W3680
  - RAM: 6GB
  - OS: Cent OS 5.4(2.6.18)
  - BLAS
    - CPU: GotoBLAS2 1.13
    - GPU: CUBLAS 3.1

◦ GPU

	GTX480	GTX285	GTX460	GTX260
Clock (GHz)	1.4	1.48	1.35	1.24
演算器数	480	240	224	216
RAM(GB)	1.5	1	1	0.87

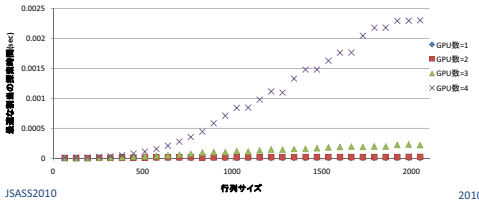
実行時間の予測精度

- 実行時間＝前処理＋行列積＋後処理
- GTX480とCPU
- 1辺が2048の正方行列積を割当量を変えて実行
  - 誤差の平均が1.6%



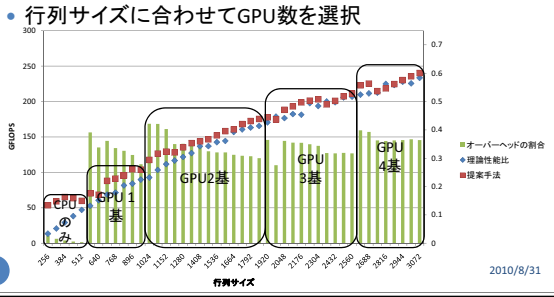
割当量の探索時間

- 正方行列の場合の探索時間
  - 3基までの探索時間は無視できる程度
  - GPUが多くなれば探索時間が飛躍的に増加
    - 一般的なマザーボードには多くて4基まで



実行性能

- GPU4基で正方行列積
  - 理論性能比で分割した場合と比較
- 行列サイズに合わせてGPU数を選択



まとめ

- 行列積の負荷分散手法を提案
  - 実行時間を予測するモデルを構築
    - 小さな誤差で予測可能
  - 理論性能比で割り当てた場合と比較
    - 問題サイズに合わせて最適なGPU数を選択
- 今後の課題
  - CPU側計算と通信のオーバーラップできる実装
  - 探索時間の削減
  - 他のBLAS関数への対応

スーパスカラ型ARMをベースアーキテクチャとする自動メモ化プロセッサの提案と実装

加藤拡† 津邑公暁† 松尾啓志†
†名古屋工業大学

1 はじめに

従来，プロセッサの高速化はプログラムの並列性に着目して研究されてきた．スーパースケーラプロセッサやマルチコアプロセッサなどがそれにあたる．その一方で，それとは着眼点を変えてプログラムの局所性に着目した高速化手法が提案されている．過去の計算結果を利用し計算を省略して高速化を図る手法である計算再利用がそれにあたる．計算再利用のハードウェア実装として自動メモ化プロセッサがある．[1] 自動メモ化プロセッサは従来は SPARC アーキテクチャをベースとして提案されてきた．そのため，他のアーキテクチャをベースとした際の実現可能性とパフォーマンスは未知である．本研究では，スーパースカラ型 ARM をベースとした自動メモ化プロセッサの実現可能性とそのパフォーマンスについて検証する．

2 研究背景

2.1 メモ化

メモ化（Memoization）とは，関数などの命令区間において，その入力と出力の組を記憶しておくことである．後に，再び同じ入力によりその命令区間が実行されようとした場合に，過去に記憶された出力を再利用することで，命令区間の実行自体を省略し，高速化を図ることができる．メモ化には，ハードウェアによるものやソフトウェアによるもの，またその双方を利用したものなど，様々なものが提案されている．

2.2 自動メモ化プロセッサ

自動メモ化プロセッサとは，ハードウェアによるメモ化を用いて計算の高速化を図る計算再利用の実装モデルである．自動メモ化プロセッサはまず，関数実行時にその入力と出力を表に記憶する．そして，再び同じ関数が実行されるときにその入力を過去の入力と比較し，一致した時には対応する出力をレジスタ，メモリに書き戻し，当該関数の実行を省略する．

自動メモ化プロセッサでは，関数の入出力を登録する MemoTbl と，その登録のための一次記憶領域である MemoBuf が通常のプロセッサ構造に追加されている．

関数の再利用は call 命令から return 命令までの命令区間を再利用対象として扱う．ある関数への call 命令が検出された際，その関数の先頭アドレスが MemoTbl 上に存在するか検索を行う．関数の先頭アドレスが存在した場合には，入力の一致比較を行う．すべての入力が一致すれば，登録済みの出力が MemoTbl からキャッシュやレジスタへと書き戻され，関数の実行は省略される．もし当該関数が見つからなかったり，すべての入力が一致せず再利用が行われなかった場合，関数を通常通りに実行し MemoBuf 上に関数の入出力 (引数格納レジスタ，及び局所変数以外のメモリ参照) を登録していく．当該関数の実行終了時，すなわち return 命令を検出した時，登録された内容を MemoTbl に登録する．

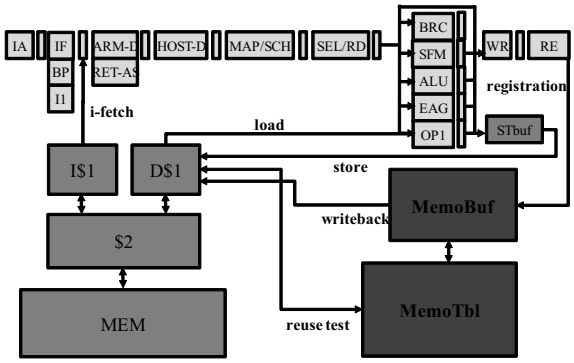


図 1: 提案モデルの HW 構成

3 提案モデル

提案モデルの HW 構成を図 1 に示す．再利用が可能かどうかを調べるために行う入力値の検索は，既存モ

Hiromu KATO†, Tomoaki TSUMURA†and Hiroshi MATSUO†
†Nagoya Institute of Technology

デル同様に関数呼び出し命令が検知された時に開始される．本提案モデルでは，リタイアステージにて関数呼び出し命令がコミットされた時点で入力値の検索は行われる．再利用が可能である場合はパイプライン上の後続命令はパイプラインフラッシュによって無効化され，関数復帰後の後続命令が改めてフェッチされ実行が再開される．

4 実装

スーパースカラ型 ARM をベースとした自動メモ化プロセッサを実現するための問題がいくつか存在する．

4.1 入力値同定

関数の入力は，入力値とその値の格納場所の組み合わせで表される．関数の入力を正しく認識するためには，関数の入力について引数と大域変数との区別をする必要がある．同じ関数が異なる関数コンテキストで複数回呼び出された場合，引数の絶対アドレスは関数コンテキストごとに異なる．よって，メモ化の際には引数のメモリアドレスはスタックポインタ相対アドレスで登録される必要がある．一方で大域変数の絶対アドレスは関数コンテキストに関係なく一定である．そのため，大域変数のメモリアドレスは絶対アドレスで登録される必要がある．しかし，これらの区別は困難である．そこで，これを解決するために，2 つのメモ化形式モデルを考案した．一つは，**コンテクスト固定メモ化形式**と呼ぶもので，関数が過去に呼び出されたときと同じコンテクストである状態でしか再利用が行えないモデルである．これは，関数の入力に関数実行時のスタックポインタアドレスを含めることによって実現する．もう一方は，**コンテクストフリーメモ化形式**と呼び，関数コンテクストに依存しない再利用が可能なモデルである．これはバイナリ生成前にソースプログラムからスタック領域情報を抽出することで実現しており，既存バイナリをそのまま実行することはできない．

4.2 関数呼び出し，復帰検知

ARM ABI では関数呼び出し及び復帰に特定の命令を使用すべきであるといった規定はない．ARM の関数呼び出し，復帰コードは多岐に渡るので，単純に特定の命令を監視するだけでは，メモ化対象区間を特定することができない．そこで，関数呼び出し，復帰を検知するために，デコードステージとリタイアステージに，関数呼び出しと復帰検知回路を追加した．

表 1: シミュレータ諸元

MemoTbl CAM	256 kBytes
Comparison (register and CAM)	1 cycles/64Bytes
Comparison (Cache and CAM)	2 cycles/64Bytes
Write back (MemoTbl to Reg./Cache)	1 cycle/64Bytes
D1 cache	32 KBytes
line size	64 Bytes
ways	4 ways
miss penalty	8 cycles
D2 cache	2 MBytes
line size	64 Bytes
ways	4 ways
miss penalty	40 cycles

4.3 再利用オーバヘッド

再利用を行う場合，MemoTbl を検索する時間がかかる．そのため，再利用が成功せずともオーバヘッドは発生する．また，MemoTbl からキャッシュへの出力の書き戻しにかかる時間も存在する．そのため，再利用によって得られる効果が小さいような短い命令区間については，削減されるサイクル数よりも，再利用オーバヘッドの方が大きいという場合が存在する．このような命令区間に対しては，計算再利用を適用しないようにするため，既存モデルでは再利用オーバヘッドを動的に評価し，それに基づいて入力値検索の実施を決定する機構が提案されている．提案モデルに実装する場合では更に，再利用成功時に発生するパイプラインのバブルを再利用オーバヘッドとして考慮する必要がある．

5 評価

SPEC CPU95 INT ベンチマークソフトによる評価結果を行った．表 1 に評価環境を，図 2 に結果を示す．横軸がプログラム名を示し，縦軸はメモ化無しの通常実行を 1 として正規化した実行時間を示している．各実行プログラム毎の 4 本のグラフは左から SPARC をベースアーキテクチャとしている既存モデル, コンテクスト固定メモ化モデル, コンテクストフリーメモ化, コンテクストフリーメモ化+オーバヘッドフィルタモデルを示す．

再利用率は既存モデルに近い結果を得ることができた．しかし，再利用オーバヘッドを含めたプログラム全体の実行 cycle 数は一部のプログラムで大幅に上昇

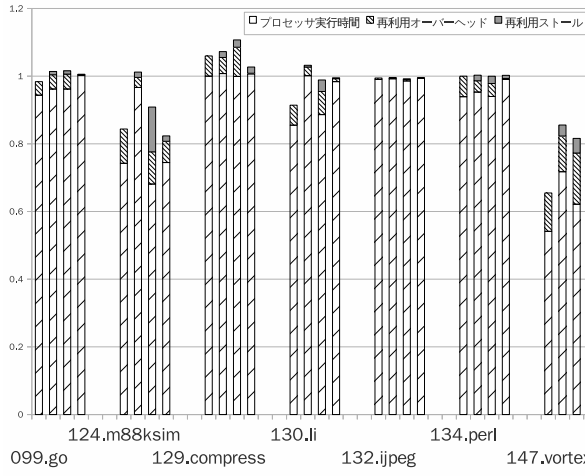


図 2: SPEC CPU95 INT 実行結果

してしまっている．提案モデル独自のオーバヘッドである再利用ストールを考慮しない場合，平均 5.8%，最大 22.0%　削減 と既存モデルの削減率 (平均 7.9%，最大 19.8%) により近い結果となる．そのため，再利用ストールを隠蔽することが今後の課題と考えられる．

また，オーバヘッドフィルタを実装したモデルでは，一部のプログラムでは高速化できているが，一方でいくつかのプログラムにおいては逆に低速化してしまっている．これは，オーバヘッドフィルタの精度が低いために，本来は再利用を適用すべきであったエントリについても再利用を適用しなかったためであると考えられる．

6　まとめ

本研究では，スーパースカラ型 ARM プロセッサをベースアーキテクチャとした自動メモ化プロセッサモデルを提案し，その実現可能性とパフォーマンスの検証を行った．結果，既存モデルに比較して解決すべき問題が増えたものの，既存モデルに近い高速化性能を得ることが出来ることがわかった．

今後の課題として，再利用ストールの隠蔽，オーバヘッドフィルタの精度向上，既存モデルに実装済みの高速化手法の実装などが挙げられる．

謝辞

本研究の一部は，(財）栢森情報科学振興財団研究助成金による．

参考文献

[1] Tsumura, T., Suzuki, I., Ikeuchi, Y., Matsuo, H., Nakashima, H. and Nakashima, Y.: Design and Evaluation of an Auto-Memoization Processor, *Proc. Parallel and Distributed Computing and Networks*, pp. 245–250 (2007).

スーパスカラ型ARMをベースアーキテクチャとする自動メモ化プロセッサの提案と実装

名古屋工業大学大学院
加藤　拓

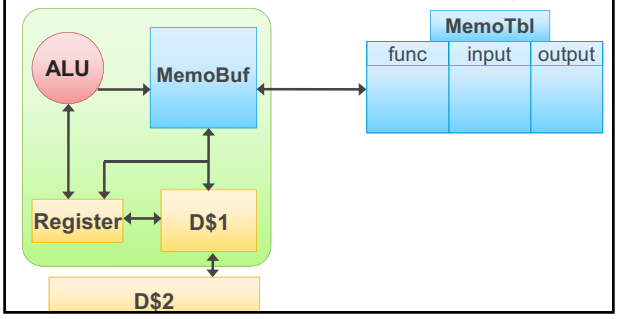
研究背景

- 様々なプロセッサ高速化手法
  - 複数単位処理の同時実行による高速化
  - プログラムの並列性を利用
    - 高いクロック周波数以外による高速化
      - SIMD・スーパスカラ
    - 電力効率と性能向上の両立
      - 複数コアを搭載したマルチコアプロセッサ
  - 処理の実行自体の省略による高速化
  - プログラムの局所性を利用
    - 計算再利用による高速化

自動メモ化プロセッサ

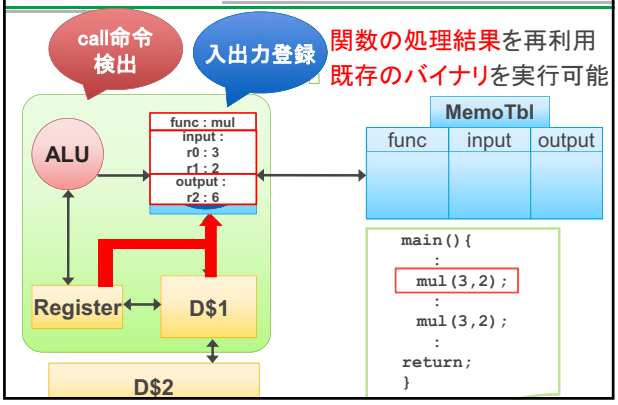
自動メモ化プロセッサ

- 関数の処理結果を再利用
- 既存のバイナリを実行可能



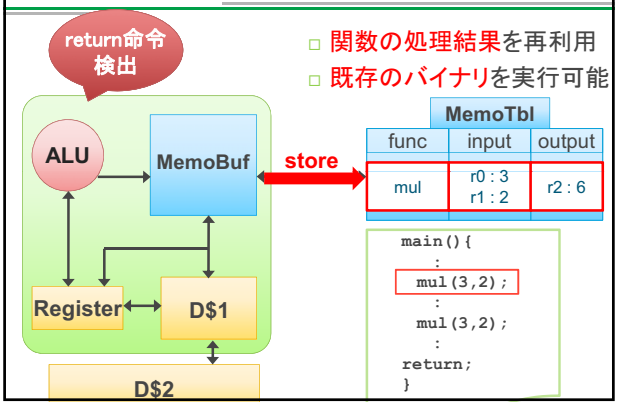
自動メモ化プロセッサ

- 関数の処理結果を再利用
- 既存のバイナリを実行可能



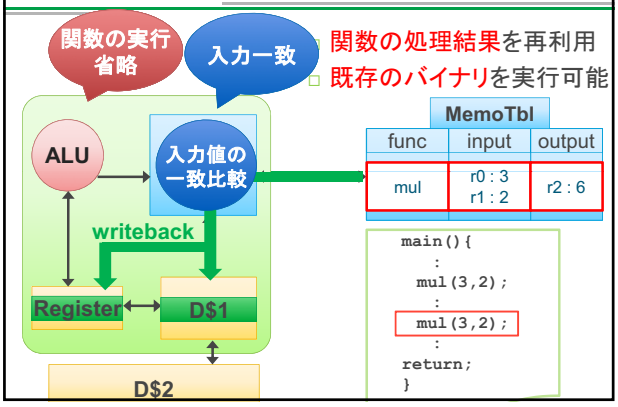
自動メモ化プロセッサ

- 関数の処理結果を再利用
- 既存のバイナリを実行可能



自動メモ化プロセッサ

- 関数の処理結果を再利用
- 既存のバイナリを実行可能

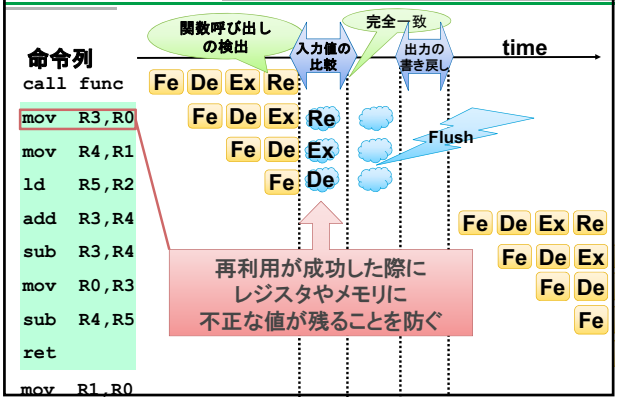


ベースアーキテクチャ

- 従来モデル
  - 単命令発行型のSPARCがベース
    - 再利用に必要な情報について仕様が明確であり、検出が容易
      - 関数call,return
      - 関数の入出力の格納場所
    - 他のアーキテクチャをベースとした際の実現可能性は未知
- 提案モデル
  - スーパースカラ型ARMがベース

パイプラインプロセッサをベースとした際の実現可能性・パフォーマンスを検証

動作モデル

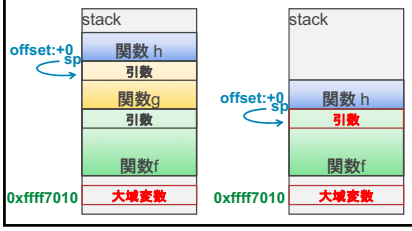


問題点

- 提案モデル実現のための問題点
  - 入力値同定の問題
    - 関数実行時のレジスタ，メモリ参照の正しい記憶
  - 関数の呼び出しと復帰検知の問題
    - 関数区間の正しい認識
- 提案モデル高速化のための問題点
  - 再利用時にかかるコストの問題
    - 再利用が適用されても高速化するとは限らない

入力値の同定問題

- 関数の入力値とは関数実行中に参照したレジスタ，メモリのアドレスと値の組
- 入力値アドレスの記憶形式は
  - 引数参照　：スタックポインタ(sp)相対アドレス
  - 大域変数参照：絶対アドレス



引数と大域変数の区別は困難

入力値同定問題の対策手法

- 既存モデル（SPARC）の対策手法
  - メモ化対象を**メモリを介して引数を受け渡ししない関数**に限定
    - 引数格納領域へのストア命令を監視
    - メモリ参照による入力は全て絶対アドレスで記憶
  - 提案モデル(ARM)では?
    - ARMでは引数格納領域は不定

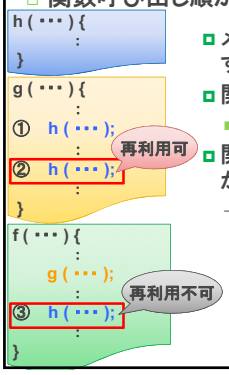
→既存モデルと同様の対策手法は不可能

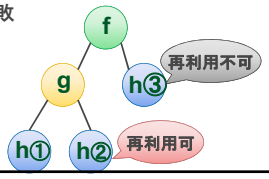
◆ 提案モデルの対策手法
- ◆ コンテクスト固定メモ化
- ◆ コンテクストフリーメモ化

コンテクスト固定メモ化

- 関数呼び出し順が過去と同じ再利用のみ可能
  - メモリ参照による入力の登録形式はすべて絶対アドレス
  - 関数コンテクスト情報を入力に含める
    - コンテクスト情報：関数開始時のSP
  - 関数コンテクストが過去と異なるとSPが不一致

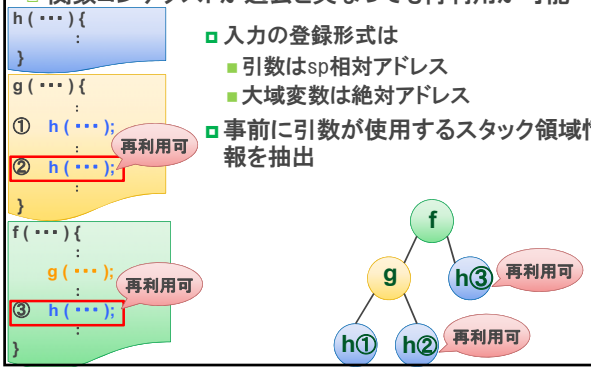
→再利用失敗





コンテクストフリーメモ化

- 関数コンテクストが過去と異なっても再利用が可能
  - 入力の登録形式は
    - 引数はsp相対アドレス
    - 大域変数は絶対アドレス
  - 事前に引数が使用するスタック領域情報を抽出

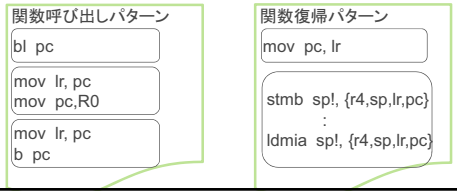


問題点

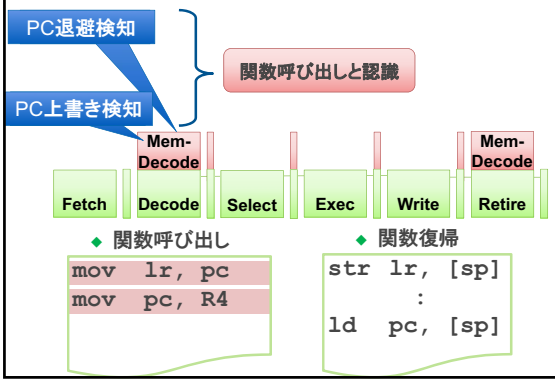
- 提案モデル実現のための問題点
  - 入力値同定の問題
    - 関数実行時のレジスタ，メモリ参照の正しい記憶
  - 関数の呼び出しと復帰検知の問題
    - 関数区間の正しい認識
- 提案モデル高速化のための問題点
  - 再利用表の検索コストの問題
    - 再利用が適用されても高速化するとは限らない

関数の呼び出し，復帰検知の問題

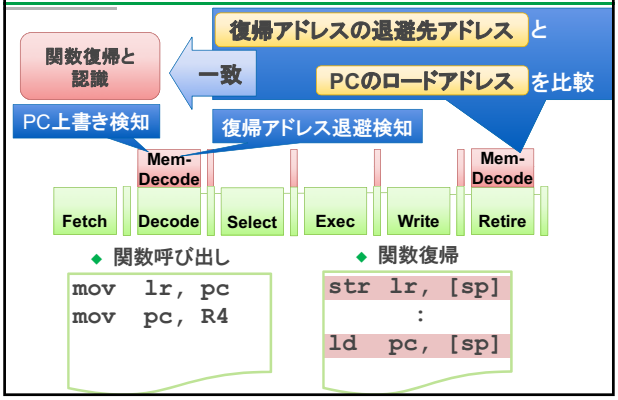
- 既存モデル(SPARC)
  - 監視対象
    - 関数呼び出し：call　　関数復帰:ret,retl
- 提案モデル(ARM)
  - 監視対象



関数の呼び出し，復帰検知デコーダ



関数の呼び出し，復帰検知デコーダ



問題点

- 提案モデル実現のための問題点
  - 入力値同定の問題
    - 関数実行時のレジスタ，メモリ参照の正しい記憶
  - 関数の呼び出しと復帰検知の問題
    - 関数区間の正しい認識
- 提案モデル高速化のための問題点
  - 再利用時にかかるコストの問題
    - 再利用が適用されても高速化するとは限らない

計算再利用の際にかかるコスト

- 再利用を試みる際にかかるオーバーヘッド
  - 検出した関数呼び出しの入力値とMemoTblの各エントリの比較

再利用が成功せずともコストは必ず発生する

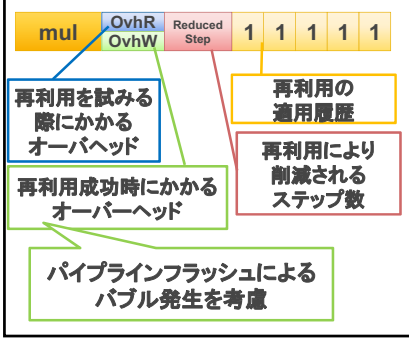
- 再利用成功時にかかるオーバーヘッド
  - 検索したエントリの出力値のレジスタやキャッシュへの書き戻し
  - 再利用成功時のパイプラインのフラッシュによるバブル発生

再利用が成功しても高速化するとは限らない

効果が見込めないエントリは
そもそも登録・検索すべきではない

オーバヘッドフィルタ

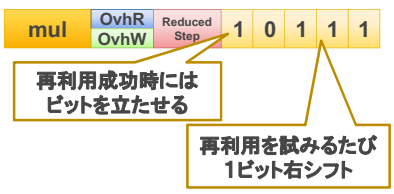
- 効果を見積もり、効果の出る関数のみ比較・登録を行う




```
main(){
   :
mul(3,2);

mul(3,2);

mul(3,2);

mul(4,2);
   :
return;
}
```

オーバヘッドフィルタ

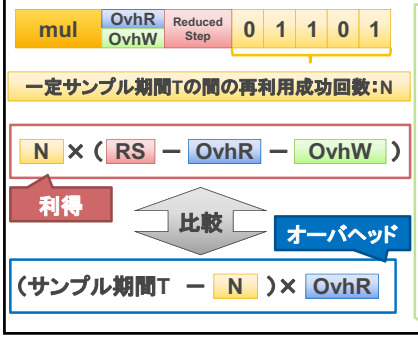
- 効果を見積もり、効果の出る関数のみ比較・登録を行う




```
main(){
   :
mul(3,2);

mul(3,2);

mul(3,2);

mul(4,2);
   :
return;
}
```

オーバヘッドフィルタ

- 効果を見積もり、効果の出る関数のみ比較・登録を行う




```
main(){
   :
mul(3,2);

mul(3,2);

mul(3,2);

mul(4,2);
   :
return;
}
```

評価環境

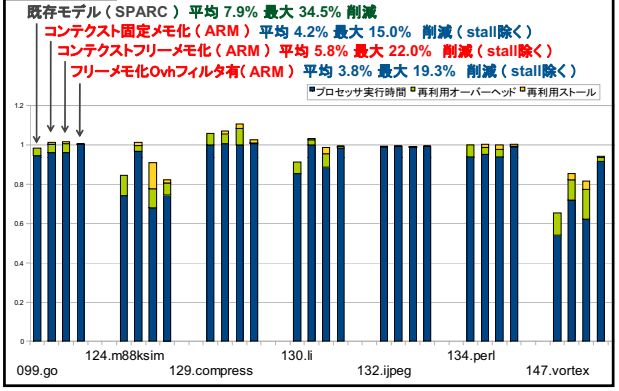
- 評価パラメータ

アーキテクチャ	ARM
パイプライン段数	9
分岐予測	g-share

Size L1			32	Kbytes
L2			2	Mbytes
Penalty L1 miss			8	cycles
L2 miss			40	cycles
再利用Ovh	入力値の比較	CAM ⇔ レジスタ	1	cycle
		CAM ⇔ L1	2	cycles
	出力書き戻し	MemoBuf ⇔ レジスタ	1	cycle
		MemoBuf ⇔ L1	2	cycles

- SPEC CPU95 INT

評価結果



まとめと今後の課題

- スーパスカラ型ARMをベースアーキテクチャとする自動メモ化プロセッサを提案し、性能を検証した.
  - シミュレーション評価により, 既存モデルに近い性能が得られることを確認した.

- 今後の課題
  - オーバーヘッドフィルタの精度向上
  - オーバーヘッドの削減・隠蔽
    - オーバーヘッドフィルタの精度向上
    - ストール隠蔽手法の提案
  - 既存の自動メモ化プロセッサに搭載済みの拡張機能の実装
    - 入力値テストの削減手法
    - Loop再利用の実装

自動メモ化プロセッサにおける並列事前実行コアによるプリフェッチ効率向上

池谷 友基 † 津邑 公暁 † 松尾 啓志 †
† 名古屋工業大学

1 はじめに

配線遅延の相対的な増大に伴い，高いクロック周波数だけでは高速化を実現しにくくなったことで，SIMDやスーパスカラなどの命令レベル並列性に基づく高速化手法が注目された．また，近年は電力効率と性能向上を両立させる観点から，複数コアを搭載したマルチコアプロセッサが主流となりつつあり，今後集積度の向上に伴なってコア数も増大していくと考えられている．

一方我々は，計算再利用技術に基づいた高速化手法である自動メモ化プロセッサを提案している．[1] 計算再利用とは，関数やループなどの命令区間に対してその入力と出力の組を実行時に記憶しておき，再び同じ入力によりその命令区間が実行されようとした場合に，過去に記憶された出力を利用することで命令区間の実行自体を省略し，高速化を図る手法である．また我々は，ループイタレーション等の命令区間のうち入力が単調変化するものに対し，入力を過去の履歴から予測し，その予測された値を用いて命令区間を別コアで予め実行しておくことで出力を生成・記憶する並列事前実行と呼ぶモデルを提案している．

本稿では，並列事前実行コアの動作区間を従来より増加させることで，キャッシュプリフェッチの効率を向上させる手法を提案する．

2 自動メモ化プロセッサと並列事前実行

2.1 自動メモ化プロセッサ

自動メモ化プロセッサは主に，メモ化制御機構，再利用表 **MemoTbl**，および MemoTbl への書込みバッファとして働く **MemoBuf** から構成される．命令区間実行開始時には MemoTbl を参照し，過去の入力との一致比較を行う．一致するエントリが存在した場合，対応する出力が書き戻され，命令区間の実行は省略される．一致するエントリが存在しなかった場合，入出力を MemoBuf に格納しつつ当該命令区間を通常実行し，実行終了時に MemoBuf の内容を MemoTbl に格納することで将来の再利用に備える．なお，入力には関数の引数はもちろんのこと，当該命令区間で発生した主記憶参照も全て含まれる．また出力には，関数の返り値および当該命令区間で発生した主記憶書込みが含まれる．

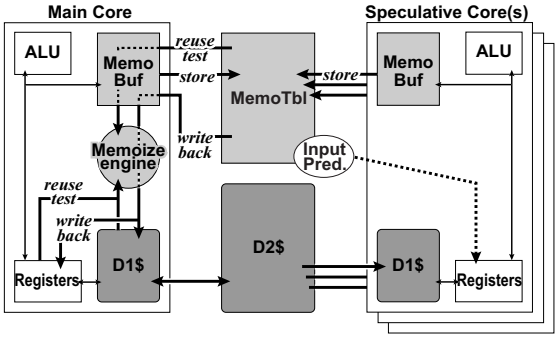


図 1: 自動メモ化プロセッサの構成

2.2 並列事前実行機構

自動メモ化プロセッサは計算再利用に基づく手法であり，当然ながらある命令区間を過去に完全に同一の入力セットで実行したことがある場合にのみ効果が得られる．よってイタレータ変数を入力のひとつとして扱うループイタレーションでは，全く効果が得られない．

そこで，計算再利用を行いながら実行を進めるメインコアとは別に，値予測に基づいて同一命令区間をメインコアに先がけて実行する投機実行コアを複数備えるシステムを考える．これを我々は**並列事前実行**と呼んでいる．以下この投機実行コアを**並列事前実行コア**と呼ぶこととする．プロセッサは複数の並列事前実行コアを用いて構成可能である．図 1 に，並列事前実行機構を含めた自動メモ化プロセッサの概要を示す．

2.3 オーバヘッドフィルタ機構

計算再利用のオーバヘッドが大きい場合には，メモ化適用により却って性能が悪化する場合もある．また，

Tomoki Ikegaya †Tomoaki Tsumura †Hiroshi Matsuo †
†Nagoya Institute of Technology

並列事前実行では投機実行の対象とする命令区間をいかに選択するかが重要である．そこで MemoTbl では，再利用におけるオーバヘッドやそれぞれの命令区間での区間実行サイクル数を記憶している．

これらの値から，再利用エントリの登録時および計算再利用適用時に判定を行い，再利用による効果が得られない，例えば実行サイクル数よりも再利用オーバヘッドが大きい場合には，再利用機構の動作を停止させている．

3 提案手法

オーバヘッドフィルタ機構の判定によって再利用機構の動作が停止すると，並列事前実行コアもそれに伴い動作を停止する．しかし，並列事前実行機構は当該区間をメインコアの直前に実行するため，キャッシュプリフェッチ機構として働く利点も兼ね備えている．そこで，従来のオーバヘッドフィルタ機構を改良し並列事前実行コアの動作区間を増加させることで，キャッシュプリフェッチ効率を向上させる．

再利用エントリ登録時ではオーバヘッドフィルタの判定により，不要なエントリは MemoTbl への登録は行わず破棄される．しかし，エントリが登録されなくなることで並列事前実行のための値予測もできなくなり，並列事前実行コアは動作を停止することになる．そこで，再利用エントリに含まれる情報の内，並列事前実行に必要なデータは無条件で MemoTbl に登録するようにした．

4 評価

上記の拡張を自動メモ化プロセッサに実装し，サイクルベースシミュレーションによる評価を行った．評価に用いたパラメータを表 1 に示す．

評価には汎用ベンチマークプログラムである SPEC CPU を用いて行い，メモ化を行わないモデル，従来のメモ化モデル，提案モデルについて評価をとった．また参考データとして，データの必要性を判断せず，無条件で MemoTbl への再利用エントリの登録を行うモデルについても評価を行った．評価結果を図 2 に示す．都合により FP のプログラムのみ載せている．

提案手法の適用により，並列事前実行コアの動作により共有の 2 次キャッシュにメインコアの実行に必要なデータが直前に乗ることで，2 次キャッシュミスサイクル数を大幅に削減することができた．

表 1: シミュレータ諸元

MemoTbl CAM	128 kBytes
Comparison (register and CAM)	9 cycles/32Bytes
Comparison (Cache and CAM)	10 cycles/32Bytes
Write back (MemoTbl to Reg./Cache)	1 cycle/32Bytes
D1 cache	32 KBytes
line size	32 Bytes
latency	2 cycles
miss penalty	10 cycles
D2 cache	2 MBytes
line size	32 Bytes
latency	10 cycles
miss penalty	100 cycles

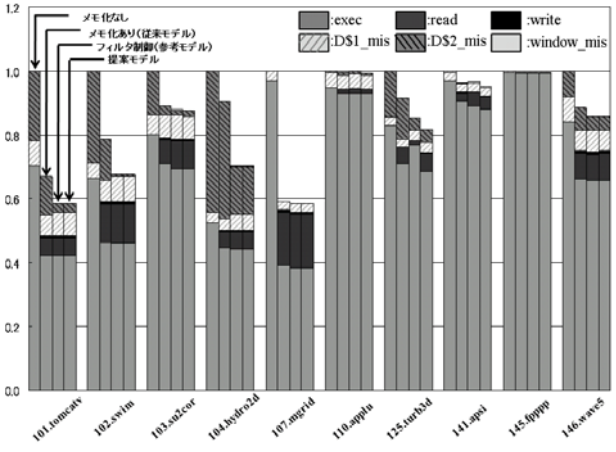


図 2: 実行サイクル数比

5 まとめ

自動メモ化プロセッサにおける並列事前実行コアの動作区間を増やし 2 次キャッシュミスを削減した．今後の課題は，キャッシュミスレートや再利用成功率を調査し，提案手法による影響を調べることが挙げられる．

謝辞

本研究の一部は，(財) 栢森情報科学振興財団研究助成金による．

参考文献

[1] Tsumura, T., Suzuki, I., Ikeuchi, Y., Matsuo, H., Nakashima, H. and Nakashima, Y.: Design and Evaluation of an Auto-Memoization Processor, *Proc. Parallel and Distributed Computing and Networks*, pp. 245–250 (2007).

自動メモ化プロセッサにおける並列事前実行コアによるプリフェッチ効率向上

○ 池谷友基†
津邑公暁†
松尾啓志†
中島康彦††

†名古屋工業大学
††奈良先端科学技術大学院大学

研究背景

□様々なプロセッサ高速化手法

複数の単位処理を同時実行により高速化

- 配線遅延の相対的な増大
- 高いクロック周波数だけでは高速化を実現しにくい
  - 命令間の並列性に基づく、SIMDやスーパスカラ
- 電力効率と性能向上の両立
  - 複数コアを搭載したマルチコアプロセッサ

□自動メモ化プロセッサ

処理の実行自体を省略し高速化

- 計算再利用技術に基づく高速化手法
- 関数およびループを再利用対象区間とみなす
- 過去の実行結果を利用することで実行自体を省略

計算再利用とメモ化

□計算再利用
- 命令区間に対して入力と出力の組を実行時に記憶
- 同一入力での実行を過去の出力を利用して省略

□メモ化
- 計算再利用を適用可能な形にすること

ラベルからRET命令 / 後方分岐命令とそのターゲットの間

関数

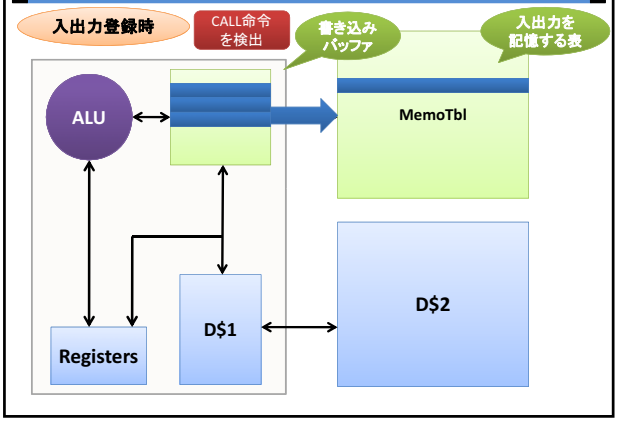
```
func:
      :
      :
   return %x
main:
      :
   call func
      :
```

ループ

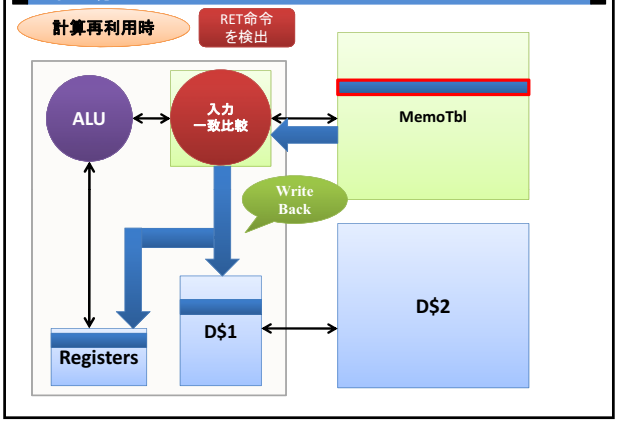
```
      :
      :
.LL3:
      :
      :
   ba .LL3
      :
      :
```

メモ化可能な命令区間

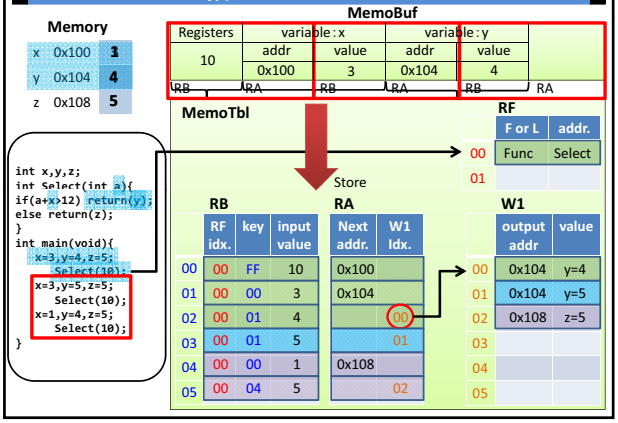
自動メモ化プロセッサ



自動メモ化プロセッサ



MemoTblの構造

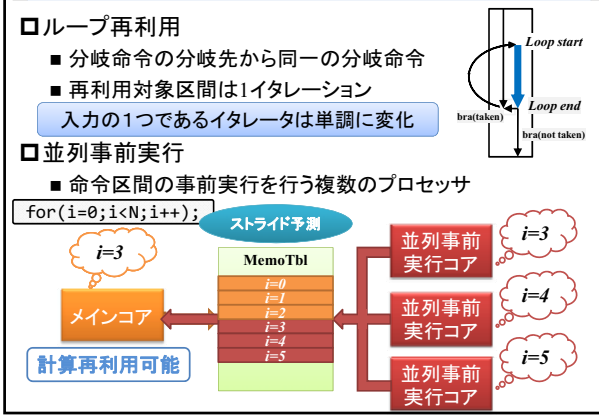


ループ再利用と並列事前実行

- ループ再利用
  - 分岐命令の分岐先から同一の分岐命令
  - 再利用対象区間は1イタレーション

入力の1つであるイタレータは単調に変化

- 並列事前実行
  - 命令区間の事前実行を行う複数のプロセッサ



オーバヘッドフィルタ機構

- 再利用オーバヘッド
  - 再利用表を検索するコスト
  - 出力をレジスタやキャッシュに書き戻すコスト

メモ化の適用によりサイクル数が増大する場合が存在

- オーバヘッドフィルタ機構
  - 効果が出ない区間に対して再利用適用をとめる
  - 判定：実行サイクル数、再利用オーバヘッドの見積り
    - 再利用テスト時
    - 再利用エントリの登録時

自動メモ化プロセッサの問題点

- フィルタが働く区間に対して再利用機構が動作しない
  - 並列事前実行コアが動作せず遊休の状態

- 並列事前実行の利点

メインコアが初めて実行する区間に対しても再利用が適用可能

直前に当該区間を実行することでプリフェッチ機構として働く

再利用効果が得られない区間に対しても
プリフェッチ機構としては動作させたい

提案：プリフェッチ効率向上

- 並列事前実行コアをフィルタの影響を受けず動作させる
  - 従来では動作しない区間に対しても動作するように改良

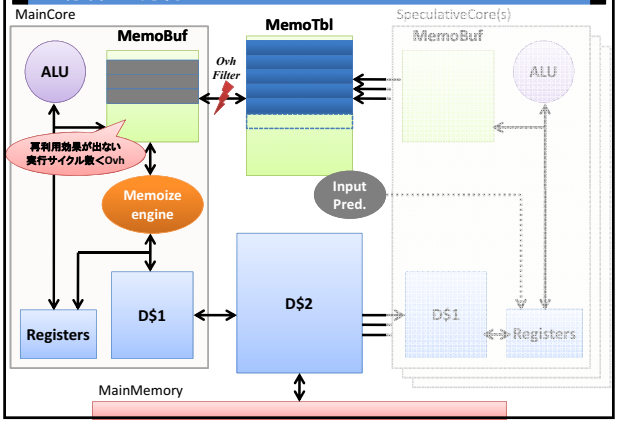
- フィルタによって並列事前実行コアが動作しない理由
  - 当該区間に対する再利用エントリが登録されない
  - ストライド予測に必要な情報が足りなくなる

- オーバヘッドフィルタ機構

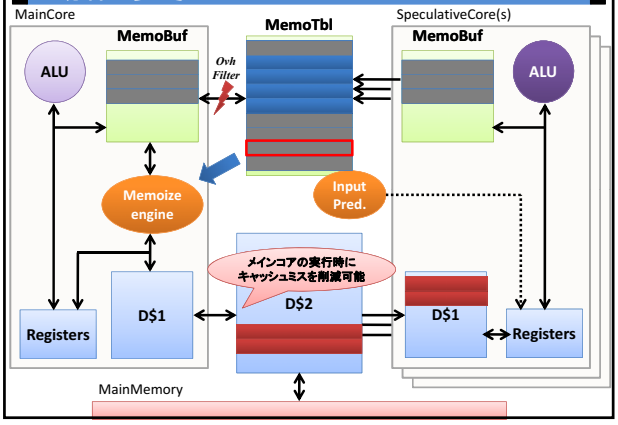
再利用テスト

再利用エントリの登録

動作：既存モデル



動作：参考モデル



予想される効果

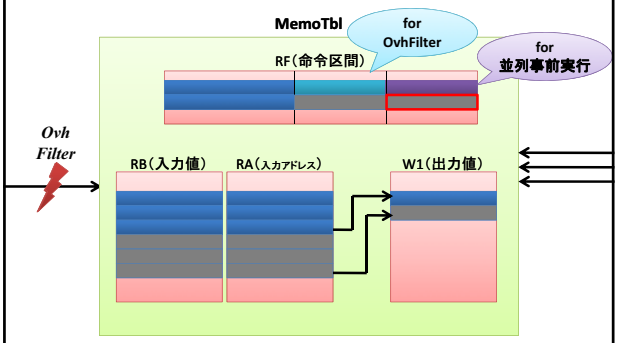
- OvhFilterによる登録時の制御を外す
  - MemoBufに蓄えたエントリは無条件でMemoTblに登録する

■ 予想される効果（メリット）
- 並列事前実行コアのプリフェッチ機構としての動作区間が増加
- メモリアクセスを減らし、2次キャッシュミスを削減

■ 予想される効果（デメリット）
- 効果の出ないエントリをMemoTblに登録
  - 有効なエントリを削除する可能性が高まり再利用率低減
- 効果の出ない区間に対してフィルタが誤判断
  - 再利用の適用により却ってサイクル数が増大

実装

- 並列事前実行に必要な情報のみ取得する



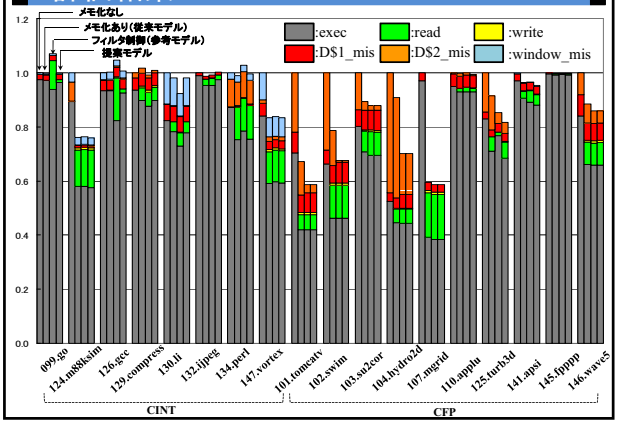
評価環境

- シミュレーション環境
  - 単命令発行のSPARC-V8シミュレータ
  - シミュレーションパラメータ

D$1Cache容量	32	Kbytes
D$1CacheMiss	10	cycle
D$2Cache容量	2	Mbytes
D$2CacheMiss	100	cycle
MemoTblサイズ	16	Kbytes
比較コスト（レジスタ⇔ MemoTbl）	9	cycle/32bytes
比較コスト（D$1⇔MemoTbl）	10	cycle/32bytes
書き戻しコスト（MemoTbl⇒レジスタ,D$1）	1	cycle/32bytes
並列事前実行コア	3	コア

  - ベンチマークプログラム
    - SPEC　CPU95

評価結果



まとめと今後の課題

- 並列事前実行コアによるプリフェッチの効率を向上
  - エントリ登録時のオーバヘッドフィルタの動作を改良
  - 並列事前実行コアのプリフェッチ機構としての動作区間を増加
  - メモリアクセスを減らし、2次キャッシュミスを削減

- 今後の課題
  - 再利用率やキャッシュミスレート等の詳細な調査
    - 更なるキャッシュミス削減手法考案への考察
  - ソフトウェアアシストによる性能向上

信号処理ライブラリ Framewave の Cell/B.E. 向け実装の提案

今井 満寿巳 † 津邑 公暁 † 松尾 啓志 †
† 名古屋工業大学

1 はじめに

今日のプロセッサではマルチコア化を進め，並列処理性能を向上させる事で，プロセッサ全体としての処理性能の向上を図っている．ヘテロジニアスマルチコアプロセッサの Cell/B.E.[1] はその一つで，PPE と呼ばれる 1 つの汎用コアと，SPE と呼ばれる複数の計算用コアを持つ．しかし，Cell/B.E. ではアーキテクチャ構成を意識したプログラミングをする必要があり，プログラマを育成する妨げとなっている．そこで，Cell/B.E. の性能を容易に引き出すことができる環境として，Cell/B.E. 向けの汎用ライブラリの開発を検討する．本研究では信号処理ライブラリ Framewave を Cell/B.E. 向けに実装することで，汎用ライブラリ開発への足掛かりを探る．

2 Cell/B.E.

Cell/B.E. は，SONY，東芝，IBM の 3 社により共同開発された，高い処理性能を目指したマルチコア SIMD プロセッサである．Cell/B.E. は，1 つの汎用プロセッサ PPE(PowerPC Processor Element) と複数の演算プロセッサ SPE(Synergistic Processor Element) を 1 チップ上に集約したヘテロジニアスマルチコアプロセッサである．

Cell/B.E. ではアーキテクチャ構成を意識したプログラミングをする必要がある．Cell/B.E. のプログラムは PPE,SPE のコア別に分かれており，それらはコアの特徴に合わせて記述しなければならない．また，SPE はメインメモリに直接アクセスすることができないため，自身の持つ LocalStore 呼ばれる領域に DMA 転送を用いてデータを転送する必要がある．さらに，SPE は SIMD 演算に特化したコアであり，その性能を引き出すには SIMD 演算を使用する必要がある．このような特徴は，プログラムの本質とは関係無く，これらを意識して記述することはプログラマにとって負担になると考えられる．

Masumi Imai †Tomoaki Tsumura †Hiroshi Matsuo †
†Nagoya Institute of Technology

3 Framewave

Framewave は，AMD により開発された信号処理ライブラリである．Framewave はオープンソースライブラリであり x86 プロセッサ向けに最適化されている．本研究ではこの Framewave を Cell/B.E. 向けに実装することで，Cell/B.E. 向け汎用ライブラリ開発への足掛かりを探る．Cell/B.E. はマルチメディア演算処理を得意としているため，信号処理ライブラリである Framewave を Cell/B.E. 向け実装するライブラリとして選択した．

4 実装

Cell/B.E. 向け汎用ライブラリの開発への足掛かりを探るため，Framewave を Cell/B.E. 向けに実装することを提案する．

Framewave のサンプルプログラム内の機能ブロックを Cell/B.E. に移植する．各機能ブロックでは Framewave の関数が使用されており，それらを Cell/B.E. 向けに実装する．実装する機能ブロックは次の 3 つの機能ブロックである．

- GREY
  画像をグレースケール化する機能ブロックである．処理の軽い関数を一つ使用している機能ブロックであり，DivC_8u 関数を使用している．
- GAMMA
  画像のガンマ値を変更する機能ブロックである．処理の軽い関数を複数使用している機能ブロックであり，Convert_32f8u 関数，DivC_32f 関数，MulC_32f 関数を使用している．
- MEDIAN
  画像にメディアンフィルタを適用する機能ブロックである．処理の重い関数を一つ使用している機能ブロックであり，FilterMedian_8u 関数を使用している．

4.1 SPE 自動使用

各ライブラリ関数の内部では SPE を自動的に使用するように実装した．ユーザプログラムがライブラリ関数を呼び出すと，ライブラリ関数は対応する処理を行う SPE プログラムを起動する．これにより，ユーザは関数を呼び出すだけで SPE を使用することができる．また，ライブラリ関数による SPE プログラムの起動は複数 SPE にも対応している．複数 SPE を使用する場合，自動的に関数内部で各 SPE に対して処理分割を行う．そのため，ユーザ使用する SPE の数を意識する必要はない．

4.2 処理分割

実装した各関数は，単一処理と近傍処理の 2 種類に分類される．単一処はある画素の処理にその画素のみが必要な処理であり，近傍処理はある画素の処理に近傍画素が必要な処理である．各 SPE への処理分割は単一処理と近傍処理で異なる．

単一処理の関数は，Convert_8u32f，DivC_8u，DivC_32f，MulC_32f である．Convert_8u32f は入力データの型を変換する関数，DivC_8u，DivC_32f は入力データを定数で除算する関数，MulC_32f は入力データを定数で乗算する関数である．単一処理は画素単位で処理を分割することができるが，Cell/B.E. の DMA 転送は 16bytes の倍数でしか転送できないこと，1 画素が 3bytes であることなどを考慮して，今回は 48bytes の倍数で処理を分割する．

近傍処理の関数は，FilterMedian_8u である．FilterMedian_8u は入力データに対してメディアンフィルタを適用する関数である．FilterMedian_8u では画像を 1 行毎処理すると効率が良いことから，今回は 1 行の倍数で処理を分割する．

5 評価

上記の実装を行い，評価をした．評価として既存の Framewave を Core2 Duo プロセッサで動作させたものと，Cell/B.E. 向けに実装した Framewave の実効速度を比較した．評価に用いた環境を表 1 に示す．Core2 Duo で動作させた Framewave では，x86 プロセッサ用の SIMD 命令である SSE を使用した場合と使用しない場合の 2 種類の速度を計測した．

GREY の結果を図 1 に示す．Cell/B.E. と Core2 Duo を比較すると，SSE を使用しない場合は Cell/B.E. の方が速いが，SSE を使用すると Core2 Duo の方が速い．また，SPE を 6 つ使用するよりも，SPE を 1 つ使用した方が速い．これは SPE の起動オーバヘッドが原因だと思われる．GREY では SPE の起動オーバヘッドに比べ計算時間が短いため，起動オーバヘッドの影響が顕著に現れてしまい，並列化による高速化よりもオーバヘッドの増加が勝ってしまった．起動オーバヘッドを除いた計算時間のみならば，SPE を 6 つ使用した場合は Core2 Duo の SSE 使用よりも高速であることが分かる．

表 1: 評価環境

CPU	Core2 Duo	Cell/B.E.
Frequency	2.66GHz	3.20GHz
OS	CentOS 5.4	Fedora 9
Memory	2GB	256MB
L2 cache	4MB	512KB
Compiler	gcc	gcc
		spu-gcc
Optimize Option	-O2	
Data size	512x512 pixel	

GAMMA の結果を図 2 に示す．Cell/B.E. と Core2 Duo を比較すると，Core2 Duo の方が速い．また，SPE を使用するよりも，PPE のみを使用する方が速い関数がある．さらに，GREY と同様に，SPE を 6 つ使用するよりも，SPE を 1 つ使用する方が速い．この原因も GREY の場合と同様に SPE の起動オーバヘッドであると考える．3 つの関数のうち，SPE を使用することで高速化した DivC_32f のみを，SPE で実行した場合の実行時間を参考として載せる．

MEDIAN の結果を図 3 に示す．MEDIAN では SIMD 演算が使用できないため，どちらも SIMD 演算は使用していない．Cell/B.E. と Core2 Duo を比較すると，他の 2 つとは異なり，Cell/B.E. の方が速い．また，SPE を 1 つ使用するよりも，6 つ使用した方が速い．これは，SPE の起動オーバヘッドに比べ MEDIAN の計算時間が長いことが原因だと考える．

6 考察

今回のように SPE を関数単位で使用すると次の 2 つの問題点があることが分かった．

- SPE の起動オーバヘッド
  関数毎に SPE を起動するので，毎回オーバヘッドがかかってしまう．そして，計算時間が短いとそのオーバヘッドがボトルネックとなってしまう．
- 関数外処理
  関数内部でしか SPE を使用しないので，関数で実

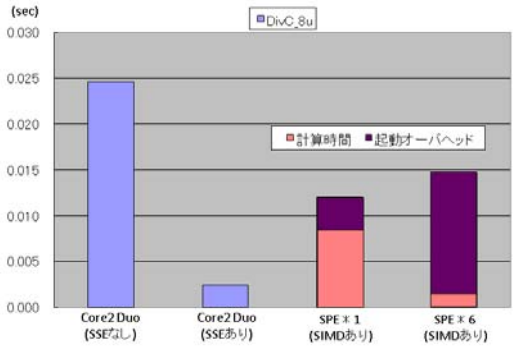


図 1: GREY

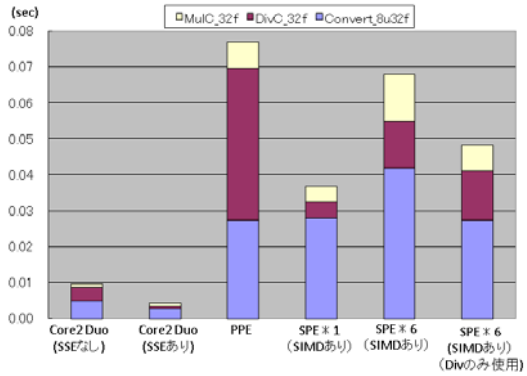


図 2: GAMMA

装されていない処理では SPE を使用できないため，高速な処理ができない．

汎用ライブラリの開発へ向けて，これらの問題点を解決することを考える．SPE の起動オーバヘッドを解決する方法としてオーバレイタスクを使用する．オーバレイタスクは Cell Tool Kit ライブラリがサポートしている機能で，SPE プログラムを通常の実行形式である SPE スレッドではなく，SPE タスクとして起動する．SPE タスクは SPE スレッドを分割したもので，SPE タスクは実行中に適宜置き換えられる．通常は SPE プログラムを起動する毎に SPE スレッドを生成する必要があるが，オーバレイタスクを使用すると SPE スレッドは 1 度起動するだけでよく，低コストで SPE プログラムを起動することができる．

関数外処理を解決する方法として，関数からの SPE プログラム自動生成を考える．関数で実装されていない処理は，ユーザに関数として記述してもらい，それの関数から SPE プログラムを生成することで，SPE で実行できるようにする．関数は入力データは引数として，出力データは戻り値として明記されているので，同様

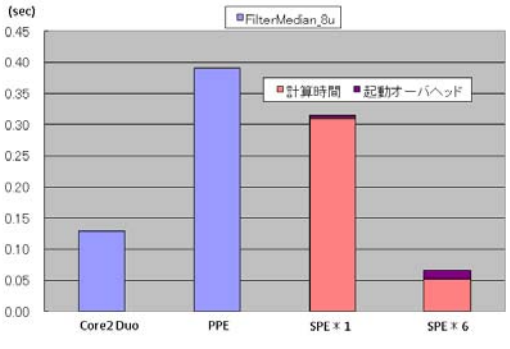


図 3: MEDIAN

の処理を SPE で実行する場合に DMA 転送が必要になるデータは明示化されている．そのため，DMA 転送文を自動挿入することは容易であると考える．また，繰り返し処理を記述するには，ユーザには単位データに対する処理のみを記述してもらい，それをもとに SPE プログラムを生成する．このとき，記述されているのは単位データに対する処理と分かっているので，SIMD 化することは容易であると考える．

7 まとめ

信号処理ライブラリ Framewave を Cell/B.E. 向けに実装し，実装を通して汎用ライブラリ開発への足掛かりを探った．．実装の結果，関数単位で SPE を使用すると，SPE の起動オーバヘッドと，関数で実装されていない処理が問題になることが分かった．そして，汎用ライブラリにおいてこの問題点を解決する方法を考えた．SPE の起動オーバヘッドを解決する方法として，オーバレイタスクの使用を，関数外処理を解決する方法として，関数からの SPE プログラム自動生成を考察した．今後は，今回考察した汎用ライブラリの開発を進めていく予定である．また，今回は対称を Cell/B.E. のみに限定しているが，他のヘテロジニアスマルチコアプロセッサにも広く対応できるような拡張が今後の課題として挙げられる．

謝辞

本研究の一部は，(財) 栢森情報科学振興財団研究助成金による．

参考文献

[1] Sony Computer Entertainment: *Cell Broadband Engine Architecture*, 1.01 edition (2006).

信号処理ライブラリFramewaveのCell/B.E.向け実装の検討

○今井満寿巳†
津邑公暁†
松尾啓志†

†名古屋工業大学

1

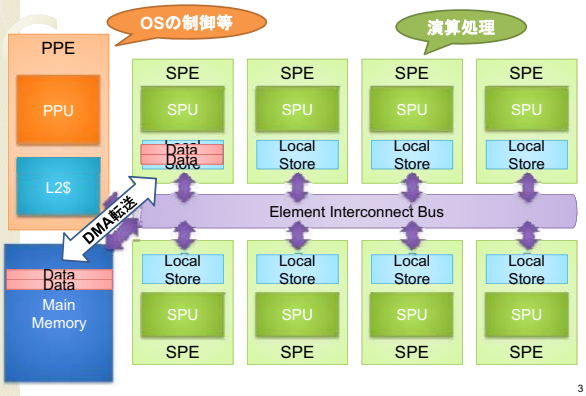
研究背景

- プロセッサの性能向上が求められている
  - スカラ処理性能の向上は困難
    - 消費電力や配線遅延の相対的な増大
- マルチコア化
  - 並列処理性能の向上
  - 複数種類のコアを使い分ける
    - ヘテロジニアスマルチコア

Cell/B.E.に着目

2

Cell/B.E.



3

Cell/B.E.の問題点

- プログラミングが煩雑
  - PPE用、SPE用プログラム
    - コアに合わせた書き分け
  - DMA転送
    - 1度の転送では最大16Kbytes
    - 転送データは16bytesの倍数
  - SIMD演算

Cell/B.E.の性能を
容易に引き出すことのできる環境が必要

↓

Cell/B.E.向け汎用ライブラリの開発を検討

4

提案

- 既存ライブラリのCell/B.E.への移植
  - 汎用ライブラリへの足掛かりを探る
    - ライブラリ構成
    - 問題点の抽出
  - Framewave
    - AMDが開発したオープンソースな信号処理ライブラリ
    - Cell/B.E.が得意とするマルチメディア処理
    - x86プロセッサ向けに最適化

5

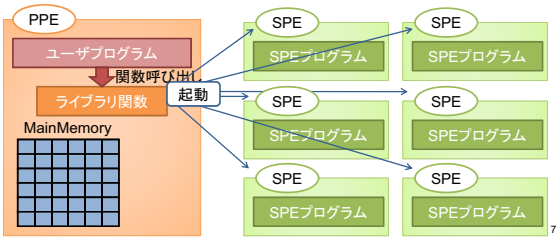
実装

- Framewaveのサンプルプログラムの機能ブロックを移植
- 機能ブロック内のライブラリ関数を実装
  - GREY（グレースケール化）
    - DivC_8u
  - GAMMA（ガンマ値の変更）
    - Convert_8u32f、DivC_32f、MulC_32f
  - MEDIAN（メディアンフィルタ）
    - FilterMedian_8u
- 関数内部でSPEを使用
  - SPEを自動的に使用することが可能に

6

SPEの自動使用

- 関数内部でSPEプログラムを起動
  - ユーザは関数を呼び出すだけでSPEを使用可能
- 複数SPEにも対応
  - 各SPEへの処理分割は自動的に行う



7

処理分割

単一処理

- Convert_8u32f(GAMMA)
  - コンバート
- DivC_8u(GREY)
- DivC_32f(GAMMA)
  - 定数で除算
- MulC_8u(GAMMA)
  - 定数で乗算

48bytesの倍数で分割

近傍処理

- FilterMedian_8u(MEDIAN)
  - メディアンフィルタを適用

1行の倍数で分割

8

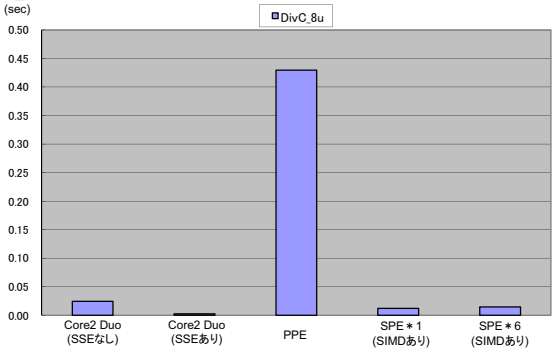
評価環境

CPU	Core2 Duo	Cell/B.E.(PLAYSTATION3)
Frequency	2.66GHz	3.20GHz
OS	CentOS 5.4	Fedora 9
Memory	2GB	256MB
L2 cache	4MB	512KB
Compiler	gcc	gcc spu-gcc
Optimize Option	-O2	
Data Size	512x512(pixel)	

- Core2 Duo
  - SSEなし、SSEあり
  - スレッド並列化（2スレッド）
- Cell/B.E.
  - PPEのみ、PPE+SPE＊1、PPE+SPE＊6
  - SPEではSIMDを使用

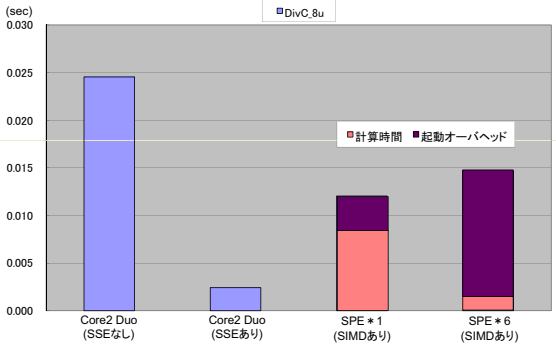
9

評価結果(GREY)



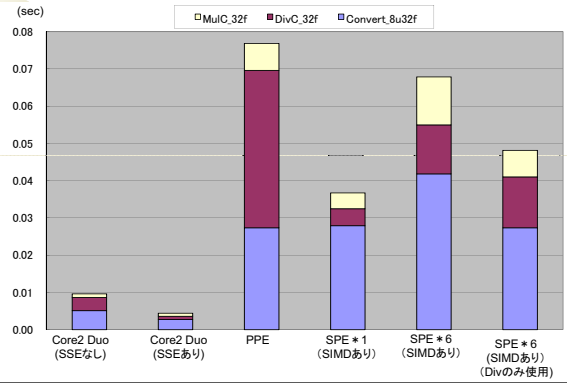
10

評価結果(GREY)

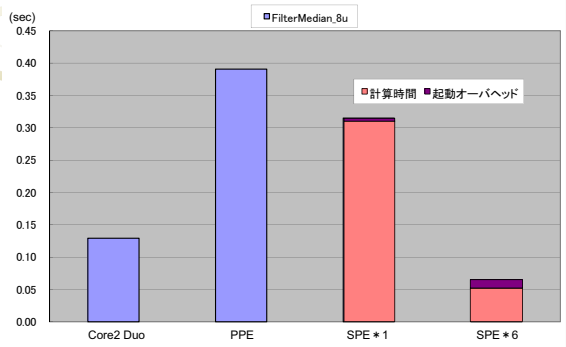


11

評価結果(GAMMA)



評価結果(MEDIAN)



13

評価考察

- SPEの起動オーバヘッド
  - 処理が軽いとボトルネックになった

SPEの起動オーバヘッドを削減する必要がある

- 関数外処理
  - 関数で実装されていない処理は高速化できない

SPEプログラムを生成する機構が必要である

14

汎用ライブラリ案

- SPEプログラムの関数化
  - ユーザは関数を使用するだけでSPEを使用できる

オーバレイタスク

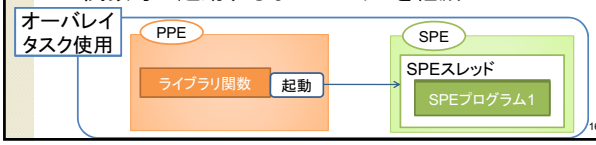
  - ライブラリ関数
    - ライブラリ側で用意された関数
  - ユーザ関数
    - ユーザが記述して生成する関数

関数からのSPEプログラム自動生成

15

オーバレイタスク

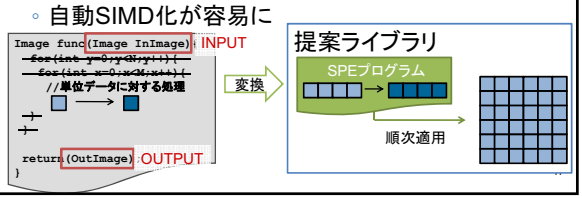
- Cell Tool Kitライブラリでサポート
- SPEプログラムをSPEタスクとして実行
  - SPEタスク
    - 通常の実行形式であるSPEスレッドを分割したもの
    - 適宜SPEタスクを置き換えながら実行される
- 低コストでSPEプログラムを起動できる
  - SPEタスクは数usecで実行可能
  - 関数毎に起動するオーバヘッドを軽減



16

関数からのSPEプログラム自動生成

- 関数をSPEプログラムに変換
  - 転送が必要なデータが明示化
    - DMA転送の自動挿入が容易に
- 並列性の抽出
  - 単位データに対する処理のみを記述
  - ライブラリ内で入力データ全てに適用する
  - 自動SIMD化が容易に



汎用ライブラリ案

- SPEプログラムの関数化
  - ユーザは関数を使用するだけでSPEを使用できる
  - オーバレイタスク
    - 関数毎の起動オーバヘッドを軽減
  - ライブラリ関数
    - ライブラリ側で用意された関数
  - ユーザ関数
    - ユーザが記述して生成する関数
  - 関数からの自動生成
    - DMA転送文の自動挿入
    - 自動SIMD化

18

関連研究

- Cell Tool Kit
  - SPE実行制御や複数SPE・PPE間の並列処理のためのAPIを提供
- OpenCV on the Cell
  - Cell/B.E.に最適化されたコンピュータビジョン向けライブラリ
- Cell/B.E.向けオフロード機構*
  - 汎用計算機上からCell/B.E.の計算資源を利用する

Cell/B.E.の構成を意識した記述が必要

*鎌田,西川,吉見,天野:"Cell Broadband Engine向けオフロード機能の提案", 情報処理学会研究報告Vol.2010-ARC-190 No.21 2010/8/4

19

まとめと今後

- まとめ
  - FramewaveをCell/B.E.向けに実装した
    - サンプルプログラムの一部を実装
    - SPE起動オーバヘッド
    - 関数外処理
  - 汎用ライブラリへの考察
    - オーバレイタスク
    - 関数からのSPEプログラム自動生成
- 今後
  - 汎用ライブラリの検討、開発
  - ヘテロジニアスマルチコアに広く対応

20

仮想計算機モニタを用いたマルウェア解析

大月 勇人 † 毛利 公一 †
† 立命館大学情報理工学部

1 はじめに

近年, コンピュータとネットワークの普及につれて, マルウェアの脅威が問題となっている. マルウェアの技術は日々着実に進歩しており, 次々と新しいマルウェアが出現している. 従来, マルウェアの解析は, 逆アセンブラやデバッガによる手法が一般的であった. しかし, 難読化やアンチデバッグが施されたマルウェアが増加し, 従来の方法による解析が困難になっている. このようなマルウェアは, OS のカーネル内部から直接監視・解析することが望ましい. しかし, Windows はプロプライエタリソフトウェアであるため, そのような手段を取ることが難しい. そこで, 我々は, OS である Windows よりも下位のレイヤで動作する仮想マシンモニタ (VMM) に注目し, Windows 上で動作するマルウェアを解析する VMM である “Alkanet” の開発を行っている [1].

Alkanet は, プロセスが発行するシステムコールをフックする機能を持つ. さらに, Windows の管理するオブジェクトを利用することで, フックしたシステムコールの詳細を得ることを可能にしている.

2 Alkanet

2.1 全体構成

Alkanet は, 仮想マシン上の Windows で動作するマルウェアを監視・解析する VMM であり, ハイパーバイザ型の VMM である BitVisor[2] をベースとして実装している. Alkanet の全体構成を図 1 に示す. Alkanet では, 仮想マシン上の Windows で動作するプロセスが発行するシステムコールをフックすることで, マルウェアの挙動の監視・解析を行う.

システムコールのフックは, Windows のシステムコールのエントリポイントにソフトウェアブレイクポイントを埋め込むことで実現している. しかし, システムコールをフックしただけでは, 発行元のプロセスや, どのリソースに対する操作かといった詳細な情報が得られない. そこで, Alkanet では, Windows の使用するメモリ空間を参照し, Windows が管理するオブジェクトにアクセスすることで, 詳細な情報の取得を可能としている.

一般的にマルウェアには, 以下のような挙動が見られる.

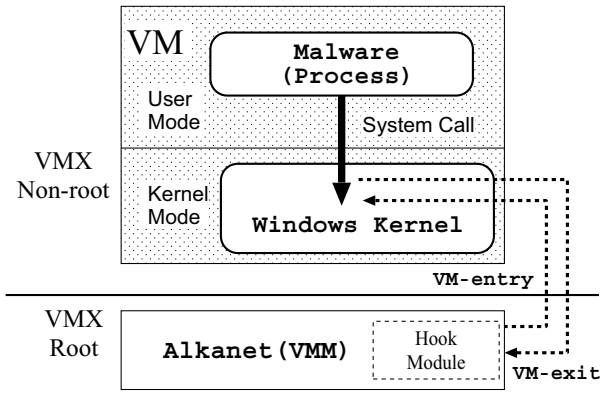


図 1 Alkanet の全体構成

重要なファイルやレジストリの改竄

バックドアの設置や別のマルウェアのダウンロード

既存のプロセスと似た名前でのプロセスの起動や別のプロセスへのコードの挿入

マルウェアが行う上記の挙動は, 主に表 1 に示すシステムコールを発行することによって行われる. したがって, Alkanet は, これらのシステムコールをフックの対象としている.

2.2 システムコールのフック

Alkanet は, Intel 製 CPU 上で動作する 32bit 版 Windows XP SP3 を対象としている. この環境におけるシステムコールは, sysenter 命令によってカーネルモードへの切り替えが行われる. sysenter 命令が実行されると, EIP レジスタは, Model Specific Register である IA32_SYSENTER_EIP の値に上書きされる. つまり, IA32_SYSENTER_EIP に格納された値が, システムコールのエントリポイントである. Alkanet におけるシステムコールのフックの流れを図 2 に示す. Alkanet は, 初期化処理として IA32_SYSENTER_EIP からシステムコールのエントリポイントを取得し, int3 命令を埋め込む. これにより, ゲスト OS 内でシステムコールが発行されると, ブレイクポイント例外が発生し, VMM 側に制御を移すことが可能になる. また, VMM からゲスト OS に制御を戻す際には, int3 命令で上書きされた命令をエミュレーションする.

Analyzing Malware by Virtual Machine Monitor
Yuto Otsuki †
Koichi Mouri †
†College of Information Science and Engineering, Ritsumeikan Univ.

表 1　Alkanet が監視する挙動

挙動	フックするシステムコールの例
ファイルの操作	NtCreateFile, NtReadFile, NtWriteFile
レジストリの操作	NtCreateKey, NtQueryValueKey, NtSetValueKey
ネットワークの操作	NtDeviceIoControlFile
プロセスの作成/終了	NtCreateProcess, NtCreateProcessEx, NtTerminateProcess
ドライバのロード	NtLoadDriver, NtUnloadDriver
別プロセスへのコード挿入	NtCreateThread, NtWriteVirtualMemory

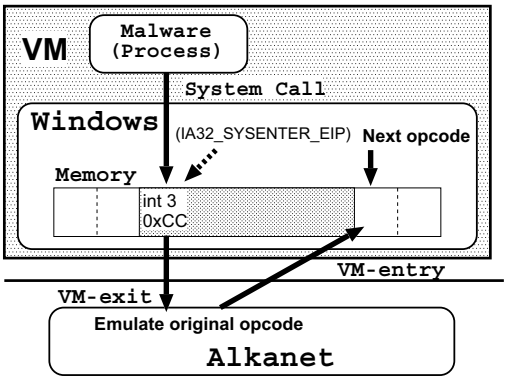


図 2　システムコールフックの流れ

Windows では, システムコール発行時, EAX レジスタにシステムコールの番号, EDX レジスタに引数のリストのアドレスがそれぞれ格納される. よって, Alkanet は, フック時にこれらのレジスタの値を用いて, 発行されたシステムコールと引数のアドレスを特定している.

2.3　発行元プロセスの特定

Alkanet は, フック時の CR3 レジスタの値を用いて, フックしたシステムコールの発行元プロセスを特定する. CR3 レジスタには, ページディレクトリテーブルの物理アドレスが格納されている. Alkanet は, この値がプロセスごとに異なっていることを利用して, システムコールを発行したプロセスを特定する. しかし, 物理アドレスの値のみでは, マルウェアの挙動を追うことが困難である. そこで, Windows の管理するメモリ空間を読み出し, プロセスの情報を取得することで, これを解決する. 具体的には, プロセスを管理するプロセスオブジェクトの双方向リストを用いる. このリストと CR3 レジスタの値とを用いて, システムコール発行元プロセスに対応するプロセスオブジェクトを特定する. これにより, 特定したプロセスオブジェクトを用いた情報の取得が可能となる.

2.4　ハンドルの解決

Windows は, オブジェクトマネージャと呼ばれるコンポーネントにより, ファイルやレジストリなどのリソースをオブジェクトと呼ばれる形で統一的に管理している. プロセスがオブジェクトをオープンすると, プロセスの持つハンドルテーブルにそのオブジェクトを示すエントリが追加される. そして, そのエントリを参照するためのハンドルがプロセスへと返される. 以降, プロセスは, このハンドルを用いて, オブジェクトへのアクセスを行う.

各種オブジェクトの操作を行うシステムコールでは, 引数として操作するオブジェクトのハンドルが与えられる. Alkanet は, プロセスの持つハンドルテーブルを参照し, 与えられたハンドルの示すオブジェクトを特定する. このようにして, システムコールの引数を基に, 実際にオープンされるファイルや書き換えられるレジストリのパスなどを取得する.

3　Alkanet によるマルウェア解析

3.1　解析対象

Alkanet の性能評価として, 実際に Alkanet 上でマルウェアの挙動解析を行った. なお, 検証時はネットワークには接続していない.

検体として, CCC DATAset 2010[3] の中で活動が記録されているマルウェア検体の中から, ClamAV[4] に Worm.Palevo-2648 と Trojan.Downloader.Bredolab-1420 として検出されるものを用いた. マルウェアのイメージ名は, それぞれ Palevo.exe と Bredolab.exe である.

3.2　Palevo.exe

Alkanet から取得したログの一部を図 3 に示す. 1753〜1755 行目では, 存在しないユーザのゴミ箱フォルダに psyjo3.exe を作成している. 自身をオープンしていることから, 自身のコピーを作成していると思われる. また, 同様に Desktop.ini を作成する挙動も確認された.

1765, 1766 行目では, 特定のレジストリのキーを参照し, 書き換えている. これらは, スタートアップアプリケーションを設定するキーである. これらのレジストリを確認すると, Windows 起動時に psyjo3.exe が起動するよう設定がされていた. また, ログオンプロセスが参照するキーを改竄する挙動も確認された.

1771, 1772 行目では, explorer.exe のメモリ空間を書き換え, 新たにスレッドを作成している. これは, explorer.exe に自身の持つコードを挿入し, 実行させると

いう挙動である．この後, explorer.exe は, 拡張子とアプリケーションの関連付けを行うレジストリを変更したり, 特定のポートを開くなどの不審な挙動を示した．

しかし, 現在の Alkanet は, 実行単位をプロセスで区別しているため, 正規の explorer.exe の挙動とコードを挿入されたことによる挙動とを厳密に区別することは難しい．この問題は, Windows における実行単位であるスレッドの情報や, 実行されているイメージの情報などを取得することで解決できる．

3.3 Bredolab.exe

Alkanet から取得したログの一部を図 4 に示す．2204〜2210 行目では, 名前付きパイプにアクセスし, lsass.exe と通信を行っている．また, svchost.exe とも同様に通信を行っていた．しかし, これらのプロセスのその後の挙動が, 正規のものであるか否かの区別は困難である．

3169, 3170 行目では, ドライブのマウントに関するレジストリのキーの変更を行っている．このレジストリは, ドライブの自動再生機能を制御する Autorun.inf のキャッシュとして利用される [5]．{a66607c8-a9f7-11df-a9a4-806d6172696f} という GUID を持つドライブを調査すると, C ドライブとしてマウントされるハードディスクであった．これらの操作によって, このマルウェアがシステム起動時に自動的に実行されるようになる．

また, この他に以下のような挙動が見られた．

DirectX やシステムの情報の収集

お気に入りや最近使ったフォルダなど設定の書換え

キャッシュや特定のドメインに対するセキュリティレベルなど, Internet Settings の書換え

ネットワークを扱うデバイスファイルへのアクセス

自身のファイルの削除

3.4 考察

検体の解析結果から, マルウェアのシステムコールをフックし, 実際に操作されるオブジェクトの情報を取得できることが確認できた．これにより, ファイルやレジストリの改竄, 別のプロセスを利用した攻撃などの基本的な挙動を解析できた．したがって, Alkanet がマルウェア解析に有効であることが確認できた．一方, 現在の実装では, マルウェアによる攻撃を受けたプロセスの挙動が, 正規の挙動であるか否かの判断が難しいことが確認された．ただし, スレッドやイメージに関する情報を用いることで解決可能である．

今後の課題として, カーネルモードで動作するマルウェアへの対応が挙げられる．このようなマルウェアは, システムコールを使用せず, カーネルの関数を直接呼び出す．したがって, システムコールのフックによる解析が困難である．この問題は, 各カーネル関数のエントリポイントでのフックを行うことによって解決する．また, カーネルモードマルウェアは, カーネルのメモリ空間にアクセスできるため, オブジェクトの隠蔽や改竄といったことも可能である [6]．したがって, Alkanet の利用するオブジェクトの保護, 隠蔽されたオブジェクトの追跡などの対応が必要である．

4 おわりに

本論文では, VMM として動作し, Windows 上で発行されるシステムコールをフックし, マルウェアの挙動を監視するシステム Alkanet について述べた．さらに実際にマルウェアを動作させて得た解析結果と, その考察を述べた．今後の課題として, 別のプロセスを利用した攻撃や, カーネルモードで動作するマルウェアへの対応が挙げられる．

参考文献

[1] 野村 他: “仮想計算機モニタを使ったマルウェアの挙動解析,” CSS2009, Vol.2009, No.11, pp.787–792, 情報処理学会 (2009).

[2] 筑波大学: “The Homepage of BitVisor,” `http://www.bitvisor.org/` (2009).

[3] 畑田 他: “マルウェア対策のための研究用データセット 〜MWS 2010 Datasets〜,” MWS2010 (2010).

[4] ClamAV: “Clam AntiVirus,” `http://www.clamav.net/` (2010).

[5] US-CERT: “US-CERT Technical Cyber Security Alert TA09-020A – Microsoft Windows Does Not Disable AutoRun Properly,” `http://www.us-cert.gov/cas/techalerts/TA09-020A.html` (2009).

[6] J. Butler: “DKOM (Direct Kernel Object Manipulation),” Black Hat Windows Security 2004, `http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf` (2004).

```
                                        :
1753 496 Palevo.exe 37 NtCreateFile
        \??\C:\Documents and Settings\yotuki\My Documents\Palevo.exe
1754 496 Palevo.exe 37 NtCreateFile
        \??\C:\RECYCLER\S-1-5-21-0243556031-888888379-781863308-1455\psyjo3.exe
1755 496 Palevo.exe 274 NtWriteFile
        \RECYCLER\S-1-5-21-0243556031-888888379-781863308-1455\psyjo3.exe
                                        :

1765 496 Palevo.exe 41 NtCreateKey Software\Microsoft\Windows\CurrentVersion\Run
1766 496 Palevo.exe 247 NtSetValueKey
        \REGISTRY\USER\S-1-5-21-1715567821-1993962763-682003330-1003
        \SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\RUN
                                        :

1771 496 Palevo.exe 277 NtWriteVirtualMemory PID: 1880, ProcessName: explorer.exe
1772 496 Palevo.exe 53 NtCreateThread PID: 1880, ProcessName: explorer.exe
                                        :
```

図 3　Palevo.exe のシステムコール取得結果

```
                                        :
2204 812 Bredolab.exe 37 NtCreateFile \??\PIPE\lsarpc
2205 1000 lsass.exe 183 NtReadFile \lsass
2206 812 Bredolab.exe 274 NtWriteFile \lsarpc
2207 1000 lsass.exe 274 NtWriteFile \lsass
2208 1000 lsass.exe 183 NtReadFile \lsass
2209 812 Bredolab.exe 183 NtReadFile \lsarpc
2210 1000 lsass.exe 183 NtReadFile \lsass
                                        :

3169 812 Bredolab.exe 41 NtCreateKey
        Software\Microsoft\Windows\CurrentVersion\Explorer\MountPoints2
        \{a66607c8-a9f7-11df-a9a4-806d6172696f}\
3170 812 Bredolab.exe 247 NtSetValueKey
        \REGISTRY\USER\S-1-5-21-1715567821-1993962763-682003330-1003\SOFTWARE\MICROSOFT
        \WINDOWS\CURRENTVERSION\EXPLORER\MOUNTPOINTS2\{A66607C8-A9F7-11DF-A9A4-806D6172696F}
                                        :
```

図 4　Bredolab.exe のシステムコール取得結果

仮想計算機モニタを用いたマルウェア解析

立命館大学 毛利研究室
大月 勇人

もくじ

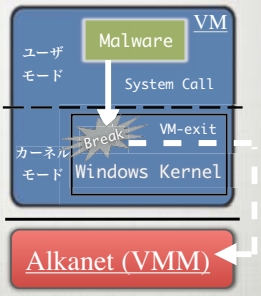
はじめに

- 近年, マルウェアの脅威が問題となっている
  - 次々と新種が出現している
- マルウェア側も解析防止の対策をとっている
  - 難読化、デバッガ検知、…
- 既存の方法では, 十分な解析が困難

マルウェアを解析する仮想計算機モニタ
“Alkanet”

Alkanet の概要

- VM 上のマルウェアが発行するシステムコールを VMM でフック
  → 挙動の監視・解析
- システムコール発行元プロセスの情報を得るために
  Windows 内部のオブジェクトの情報を読み出し、利用する



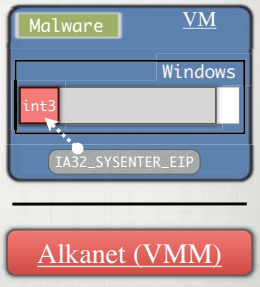
監視する挙動

挙動	システムコールの例
ファイルの操作	NtCreateFile, NtReadFile, NtWriteFile
レジストリの操作	NtQueryValueKey, NtSetValueKey
プロセスの作成/終了	NtCreateProcessEx, NtTerminateProcess
ネットワークの操作	NtDeviceIoControlFile
ドライバのロード	NtLoadDriver, NtUnloadDriver
別プロセスへのコード挿入	NtCreateThread, NtWriteVirtualMemory

フックの流れ(初期化処理)

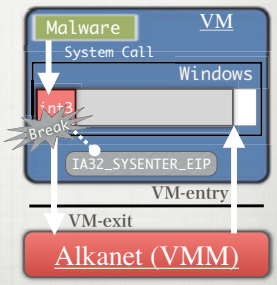
- 32 bit 版 Windows XP SP3, Intel 製 CPU 環境では、sysenter 命令が用いられる

1. IA32_SYSENTER_EIP の値取得
2. int3 命令埋め込み



フックの流れ

1. マルウェアが
   システムコールを発行
2. ブレイクポイント例外
   → VM-exit
3. さまざまな情報を取得
4. 上書きされたの命令を
   エミュレート
5. VM-entry



7

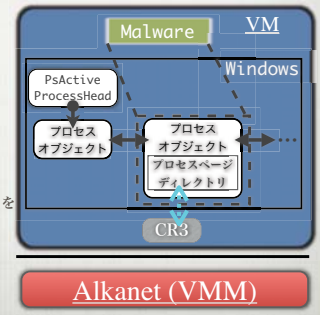
収集する情報の概要

- 発行されたシステムコール
  - EAX にシステムコール番号が格納されている
- 発行元プロセス情報
  - CR3 とプロセスオブジェクトのリストから特定する
- 操作されるリソース
  - システムコールの引数を解析する
  - EDX に引数リストを示すアドレスが格納されている

8

プロセスの特定

- CR3 レジスタの値を用いて
  システムコール発行元の
  プロセスオブジェクトを
  特定する
- CR3 と
  各プロセスオブジェクトの
  プロセスページディレクトリとを
  比較

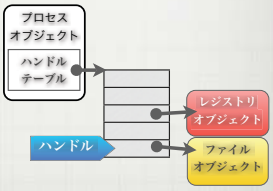


9

Windows における
ハンドルとオブジェクト

- Windows では、リソースを
  オブジェクトという形で管理
  - ファイル、レジストリ、…
- プロセスが
  オブジェクトをオープン

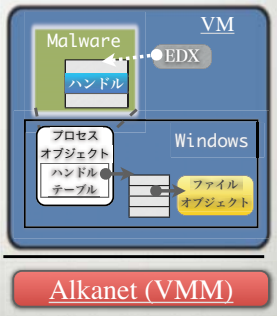
  ハンドルテーブルに
  エントリが追加される
- プロセスは、ハンドルを用いて、
  オブジェクトへアクセスする



10

ハンドルの解決

1. EDX の値を取得
   システムコールの
   引数のリストのアドレス
2. 引数よりハンドルを取得
3. ハンドルを用い、
   発行元プロセスの
   ハンドルテーブルを参照
4. オブジェクトを取得



11

実際にマルウェアを解析する

- 2体のマルウェアの解析結果について報告する
  - Palevo.exe, Bredolab.exe
- これらのマルウェアは、
  CCC DATAset 2010 に活動が記録されている
- 検証時、ネットワークには接続していない

12

Palevo (1/3)

ログ番号	PID	イメージ名	SNo	システムコール名	備考
1753	496	Palevo.exe	37	NtCreateFile	\??\C:\Documents and Settings\yotuki\My Documents\Palevo.exe
1754	496	Palevo.exe	37	NtCreateFile	\??\C:\RECYCLER\S-1-5-21-0243556031-888888379-781863308-1455\psyjo3.exe
1755	496	Palevo.exe	274	NtWriteFile	\RECYCLER\S-1-5-21-0243556031-888888379-781863308-1455\psyjo3.exe

- 存在しないユーザのゴミ箱に psyjo3.exe を作成している
- 同様に Desktop.ini を作成する挙動も確認された

13

Palevo (2/3)

ログ番号	PID	イメージ名	SNo	システムコール名	備考
1765	496	Palevo.exe	41	NtCreateKey	Software\Microsoft\Windows\CurrentVersion\Run
1766	496	Palevo.exe	247	NtSetValueKey	\REGISTRY\USER\S-1-5-21-1715567821-1993962763-682003330-1003\SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\RUN

- スタートアップアプリケーションを登録するキーに値を設定している
  - psyjo3.exe が起動するよう設定
- 同様にログオンプロセスが参照するキーについても値を設定していた

14

Palevo (3/3)

ログ番号	PID	イメージ名	SNo	システムコール名	備考
1771	496	Palevo.exe	277	NtWrite VirtualMemory	PID: 1880, ProcessName: explorer.exe
1772	496	Palevo.exe	53	NtCreate Thread	PID: 1880, ProcessName: explorer.exe

- explorer.exe のメモリ空間を書換え、さらにスレッドを作成
  - 自身のコードを挿入し、実行
- この後、explorer.exe が不審な挙動を始める
  - 関連付けの変更、ポートのオープン、……

15

Bredolab (1/3)

ログ番号	PID	イメージ名	SNo	システムコール名	備考
2204	812	Bredolab.exe	37	NtCreateFile	\??\PIPE\lsarpc
2205	1000	lsass.exe	183	NtReadFile	\lsass
2207	1000	lsass.exe	274	NtWriteFile	\lsass
2208	1000	lsass.exe	183	NtReadFile	\lsass
2209	812	Bredolab.exe	183	NtReadFile	\lsarpc
2210	1000	lsass.exe	183	NtReadFile	\lsass

- 名前付きパイプを用いて、lsass.exe と通信している
- 同様に svchost.exe との通信が確認された

16

Bredolab (2/3)

ログ番号	PID	イメージ名	SNo	システムコール名	備考
3169	812	Bredolab.exe	41	NtCreateKey	Software\Microsoft\Windows\CurrentVersion\Explorer\MountPoints2\{a66607c8-a9f7-11df-a9a4-806d6172696f}\
3170	812	Bredolab.exe	247	NtSetValueKey	\REGISTRY\USER\S-1-5-21-1715567821-1993962763-682003330-1003\SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\EXPLORER\MOUNTPOINTS2\{A66607C8-A9F7-11DF-A9A4-806D6172696F}

- Autorun.inf のキャッシュを作成
- {a66607c8-a9f7-11df-a9a4-806d6172696f} はCドライブ

17

Bredolab (3/3)

- DirectX やシステムの情報の収集
- お気に入りや最近使ったフォルダなど設定の書換え
- Internet Settings の書換え
  - キャッシュや特定のドメインのセキュリティレベルなど
- ネットワークを扱うデバイスファイルへのアクセス
- 自身のファイルの削除

18

考察 (1/2)

- Alkanet によるシステムコールフックにより、
  マルウェアの基本的な挙動を解析可能である
  - ファイルやレジストリの改竄、
    別プロセスを利用した攻撃、……
- マルウェアによる攻撃を受けたプロセスの挙動が、
  正規の挙動であるか否かの判断が難しい
  - スレッドや実行イメージの情報を取得することで解決する

19

考察 (2/2)

- 今後の課題は、カーネルモードマルウェア
- カーネルの関数を直接コールできる
  - システムコールのフックによる解析ができない
  - 対策: カーネルの関数のエントリポイントでのフック
- オブジェクトを改竄・隠蔽できる
  - 対策: オブジェクトの保護、隠蔽されるオブジェクトの追跡

20

おわりに

- Alkanet
  - マルウェアが発行するシステムコールを
    VMM でフックし、挙動の監視・解析を行う
  - CR3 とプロセスオブジェクトのリストを用いて、
    発行元プロセスを特定する
  - プロセスのハンドルテーブルを参照し、
    ハンドルの示すオブジェクトの特定する
- Palevo.exe、Bredolab.exe の解析結果
  - 自身をファイルやレジストリに複製、自動起動の設定、
    別のプロセスを利用した攻撃などが確認できた
- 今後の課題
  - 別のプロセスを利用した攻撃、カーネルモードマルウェアへの対応

21

関数ポインタ解析による侵入検知精度向上

冨永 悠生 †　　桑原 寛明 ‡　　國枝 義敏 ‡
† 立命館大学大学院理工学研究科 ‡ 立命館大学情報理工学部

1　はじめに

バッファオーバフローなどの脆弱性を利用した攻撃は，システムの乗っ取りや情報漏洩などの被害を発生させる．この問題を解決するシステムとして，ホスト型侵入検知システムが存在する．多くのホスト型侵入検知システムでは，煩雑なセキュリティポリシを管理者が記述する必要がある．セキュリティポリシを記述するには，管理者がプログラムの挙動を理解する必要があるため，管理者にとって大きな負担となる．また，管理者によるセキュリティポリシの記述ミスに起因する脆弱性が発生する可能性がある．そこで，これらの問題に対処するため，コンパイラの静的解析情報を利用するホスト型侵入検知システム [1]（以下，既存システム）を提案している．既存システムは，コンパイラの解析機能を利用することでセキュリティポリシの自動生成を行うため，管理者の負担を小さく抑えつつソースコードに基づいた侵入検知を行う．

攻撃例として，コードインジェクション攻撃やリターンアドレスの書き換え攻撃を利用した不正なシステムコール発行がある．この攻撃を防ぐために，既存システムはシステムコールの発行を契機にシステムコールと関数のリターンアドレスを検査する．制御フローが変更されてシステムコールが発行された場合，システムコールや関数のリターンアドレスは不正なものとなる．セキュリティポリシには，プログラムのシステムコールと関数の正しいリターンアドレスが記述されており，その情報を基に検査を行う．実際の情報とセキュリティポリシに記述された情報が異なる場合は，不正にシステムコールが発行されたと判断し，プログラムを停止する．既存システムでは，関数ポインタを利用して呼びだされる関数（以下，関数ポインタ呼び出し先）が特定できない．そこで，関数ポインタを利用する場合は，全ての関数の呼び出しを認めている．そのため，バッファオーバフローなどによる関数ポインタの値を書き換える攻撃（以下，関数ポインタ書き換え攻撃）により不正なシステムコールが発行されたとしても，検知することができない．

本稿では，関数ポインタ書き換え攻撃を防ぐために，静的解析情報を用いて関数ポインタ呼び出し先を絞り込む手法を提案する．本手法では，データフロー解析で関数ポインタ変数の到達定義を求めることで，呼び出し先の絞り込みを行う．これにより，実行時に関数ポインタを使用した関数の呼び出し先を制限できるため，従来検知できなかった攻撃が検知可能になる．

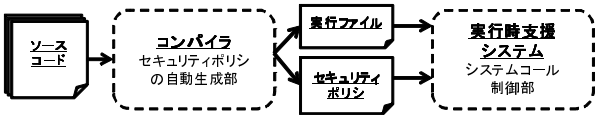


図 1: 静的解析情報を利用するセキュアシステム

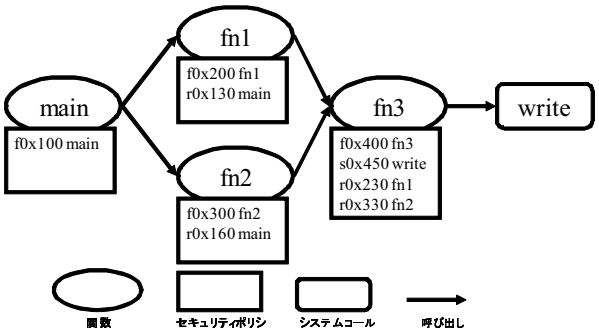


図 2: セキュリティポリシイメージ

2　静的解析情報を利用する侵入検知システム

既存システムは，図 1 のようにコンパイラと実行時支援システムが連携したシステムとなっている．

2.1　セキュリティポリシ生成部

セキュリティポリシ生成部では，プログラムのソースコードに対して静的解析を行い，関数とシステムコールごとに取り得るすべてのリターンアドレスと，発行されるシステムコールの番号を記述したセキュリティポリシを生成する．

自動生成されたセキュリティポリシのイメージを図 2 に示す．セキュリティポリシは各関数にポリシが付加されたコールグラフである．図 2 の例では，fn3 が fn1 と fn2 から呼び出されており，fn3 のポリシに各呼び出しのリターンアドレスを記述する．fn3 ではシステムコール発行が行われるため，システムコールのリターンアドレスを記述する．

2.2　システムコール制御部

システムコール制御部では，プログラム実行時のシステムコール発行を契機に，セキュリティポリシに基づいてシステムコール番号と各リターンアドレスの検査を行う．リターンアドレスの検査では，スタック上の各関数のリターンアドレスがセキュリティポリシに記述されたものと一致しなければ，異常動作と判断する．システムコール制御部では，システムコール検査とスタック検査を行う．

A Function Pointer Analysis to Improve a Precision of Intrusion Detection
Yuuki Tominaga†, Hiroaki Kuwabara‡and Yoshitoshi Kunieda‡
†Graduate School of Science and Engineering, Ritsumeikan Univ
‡College of Information Science and Engineering Ritsumeikan Univ

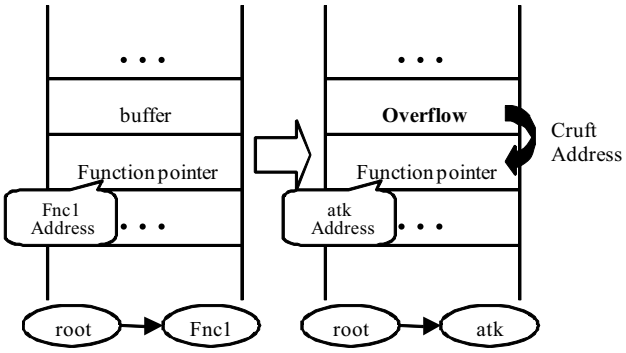


図 3: 関数ポインタを使用した攻撃例

システムコール検査は，攻撃者によるコードインジェクション攻撃を防ぐ目的を持つ．システムコール発行時のシステムコール番号と発行位置をセキュリティポリシの記述と比較することで，元のプログラムにない不正なシステムコール発行を検知する．

スタック検査は，攻撃者による不正な関数の呼び出しを利用したシステムコール発行を防ぐ目的を持つ．システムコール発行時のスタックに積まれたリターンアドレスを検査することで，セキュリティポリシに定義されていない不正な関数呼び出しを検知する．

2.3 問題点

システムコール制御部には，関数ポインタ書き換え攻撃を検知できない問題がある．この攻撃による不正な関数呼び出しの例を図 3 に示す．この攻撃はスタック上に積まれた関数ポインタ変数をバッファオーバフローにより任意の関数を示すアドレスに書き換える．この結果，不正な関数の呼び出しが行われる．既存システムのセキュリティポリシでは，関数ポインタを使用した関数呼出しはプログラム中の任意の関数を呼び出すことが可能であるとしている．そのため，2.2 節で述べたスタック検査とシステムコール検査では不正な関数ポインタ呼び出しはセキュリティポリシに定義されている正常な動作と判断し，関数ポインタ書き換え攻撃を検知できない．また，他の静的解析情報を利用した侵入検知システム [2] においても同様にこの問題が存在する．

3 関数ポインタ絞り込み手法

本稿では，関数ポインタ書き換え攻撃を検知するためにデータフロー解析を利用した関数ポインタ絞り込み手法を提案する．一般的にポインタ解析は難しい [3] ため，各関数ポインタについてそれを利用して呼び出される関数を特定するのではなく，呼び出される可能性のある関数を列挙する．具体的には，データフロー解析で関数ポインタ変数の到達定義を求める．到達定義は，関数ポインタに対して代入が行われる値の集合である．到達定義の要素が一通りの場合は，実行時に許可する呼出し先を唯一に制限できる．これにより，関数ポインタ変数の書き換えによる不正な関数呼び出しを検知することが可能となる．到達定義の要素が複数

```
1  int func(char* c){
2    int(*fncP)(char* c);
3    fncP = dummy;
4    if(...){
5      fncP = read;
6    }else{
7      fncP = wread;
8    }
9    fncP(c);
10 }
```

図 4: 本手法が有効なソースコード例

存在する場合は，それぞれの関数を呼び出すことを許可する．これにより，攻撃者は実行時に許可される関数しか呼び出すことができなくなる．以上の結果，既存システムに比べて侵入検知精度が向上すると考えられる．

例として，図 4 のソースコードに本手法を適用する。図 4 では，3 行目と 5 行目，7 行目において，関数ポインタ変数 fncP に値の代入が行われている．この場合の 9 行目の fncP の到達定義は 5 行目の read と 7 行目の wread である．そのため，9 行目の fncP による関数呼び出し先の絞り込み結果は read と wread となる．

4 評価

本手法を適応するために，glibc 内の関数ポインタを調査した．glibc 内には，関数ポインタを使用した呼び出し数が 692 個存在する．人間による簡単な絞り込みを行った結果，関数ポインタを絞り込むことができる関数ポインタの数は全体の 12.6 %の 87 個存在した．

5 おわりに

本稿では，静的解析情報を利用するホスト型侵入検知システムの侵入検知精度向上のために，関数ポインタ絞り込みによる関数ポインタ書き換え攻撃を検知する手法を提案した．今後の課題として，絞り込みの効果の測定を行う．その次に絞込みが行えていない関数ポインタに対して動的に関数ポインタの絞り込みを行う手法を考える．

参考文献

[1] 服部眞也, 毛野高彦, 桑原寛明, 國枝義敏. 静的解析情報を利用したセキュアシステムの侵入検知精度向上, 情報処理学会第 71 回全国大会, 1K-6, pp.69–70, 2009.

[2] 槙本裕司，齋藤彰一, 古屋雄介, 白井宏憲, 上原哲太郎, 松尾啓志. ライブラリ関数呼び出し監視による侵入防止システムの実現. 情報処理学会論文誌 コンピューティングシステム (ACS), Vol. 3, No. 1, pp. 38-49, 2010.

[3] G.Ramalingam.The Undecidability of Aliasing. ACM Transactions on Programming Languages and Systems, 16(5):1467-1471,1994.

High Performance Computing Software System Laboratory

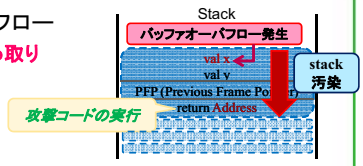
関数ポインタ固定化による侵入検知精度向上

立命館大学　理工学研究科
冨永 悠生，桑原 寛明，國枝 義敏

High Performance Computing Software System Laboratory

現状

- プログラムの脆弱性を突いた攻撃
  - バッファオーバフロー
    - システムの乗っ取り
    - 情報漏洩



- ゼロデイ攻撃
  - セキュリティパッチの適応までを対象とした攻撃

High Performance Computing Software System Laboratory

対策

- StackGuard
  - カナリアコードをリターンアドレスの前に挿入
    - リターンアドレスの書き換えを検知
- Libsafe
  - バッファオーバフローが起こりやすい関数を安全に
- セキュアOS
  - 強制アクセス制御
    - リソースへのアクセスをセキュリティポリシに基づいて制御
  - 煩雑なセキュリティポリシを管理者が定義
    - 管理者の負担の増加
    - ヒューマンエラーの混入

High Performance Computing Software System Laboratory

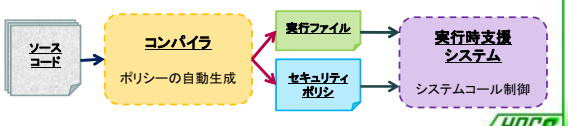
先行研究

- Wagnerらの手法
  - システムコールの呼び出し順序を検査
    - 全ての制御フローに対して検査
- Fengらの手法
  - システムコール発行時のスタックを検査
    - リターンアドレスのスナップショットを取得
- 阿部らの手法
  - Wagnerらの手法を改良
  - システムコールまでの関数遷移を検査
    - システムコール発行を契機に検査

High Performance Computing Software System Laboratory

静的解析情報を利用したセキュアシステム

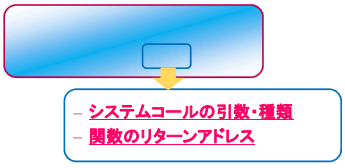
- 概要
  - コンパイラ
    - 自動的にセキュリティポリシ（規則）を生成
  - 実行時支援システム
    - セキュリティポリシに基づいたシステムコールの発行を制御



High Performance Computing Software System Laboratory

実行時支援システム

- 検知方法
  - シグネチャ検知
    - 「異常」の規則を保持

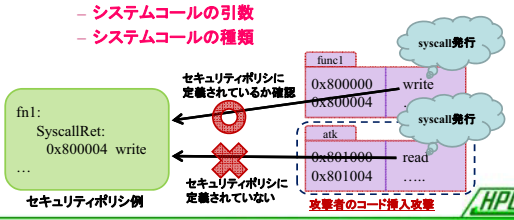


侵入検査

- 検査タイミング
  - システムコール発行
- 検査内容
  - システムコール検査
  - スタック検査

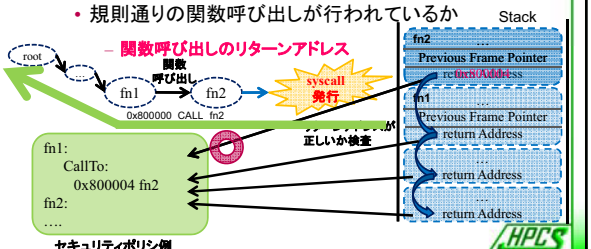
侵入検知（1）

- システムコール検査
  - 検査内容
    - 規則通りのシステムコール発行が行われているか
      - システムコールの引数
      - システムコールの種類



侵入検知（2）

- スタック検査
  - 検査内容
    - 規則通りの関数呼び出しが行われているか
      - 関数呼び出しのリターンアドレス

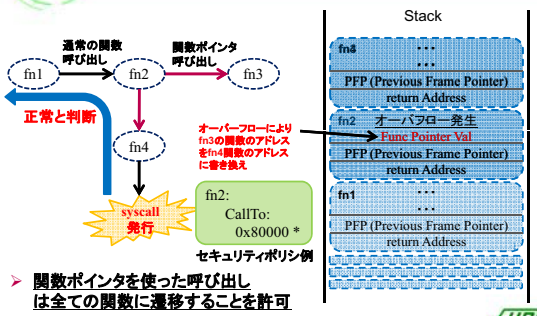


問題点

- 対処出来ない攻撃
  - 検知できない制御フローの変更

  - スタック偽装

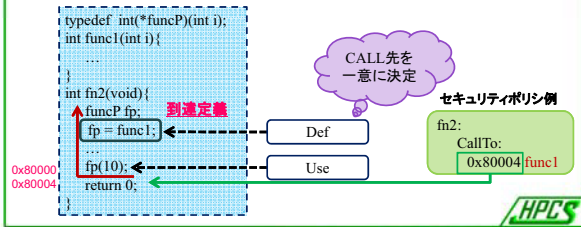
関数ポインタを使った攻撃



- 関数ポインタを使った呼び出しは全ての関数に遷移することを許可

提案手法

- 静的解析情報で関数ポインタの固定化を行う
  - データフロー解析で定義地点を特定
    - 定数伝播により絞り込みを行う

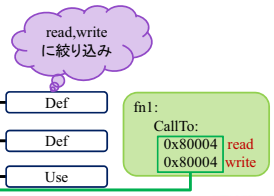


提案手法の適用

■データフロー解析を使った例

```
int read(FILE *fp,char *c);
int write(FILE *fp,char *c);
typedef  int(*funcP)(FILE *fp, char *c);

int fn1(void){
 funcP fnt;
 if( …… ){
  fnt = read;
 }else{
  fnt = write;
 }
 fnt(fp,&c);
 return 0;
}
```

到達定義



fn1:
CallTo:
0x80004 read
0x80004 write

関数ポインタ固定化による効果

■関数ポインタの呼び出し先を絞ることが可能

➢ライブラリglibcには関数ポインタを利用した呼び出しが多数存在

➢write,read,open,close等のシステムコールを呼び出す関数は関数ポインタを経由して呼び出しが行われている

- 呼び出し先の絞り込みにより攻撃を難しく

評価

■glibc内の関数ポインタの呼び出し数

➢検査方法

- 固定化した関数呼び出しをカウント
- 人の手による簡単な解析，手続き間解析は行わず

➢結果

- 87ヶ所の関数ポインタを固定化

12.6%削減

glibc

	関数ポインタ	固定化
固定化無し	692ヶ所	---
固定化有り	605ヶ所	87ヶ所

今後の課題

■手続き間解析による絞り込み可能力所の拡大

➢手続き間解析を考慮した絞り込みを行う

■解析器作成，厳密な評価取り

■対処出来ていない攻撃の対処

➢制御フローの変更

➢スタック偽装攻撃

おわりに

■関数ポインタの固定化により関数の呼び出し先の絞り込み

➢特定の関数にしか関数呼び出しができないように固定化

■glibc内の関数ポインタを調査

➢絞り込みが可能な関数ポインタをカウント

- 87カ所の関数ポインタの絞り込みが可能と判明

システムコール引数値の保護による侵入検知精度向上

山崎 悟史 †　　桑原 寛明 ‡　　國枝 義敏 ‡
† 立命館大学大学院理工学研究科　　‡ 立命館大学情報理工学部

1 はじめに

バッファオーバーフロー脆弱性に代表されるプログラムの脆弱性を利用することにより，プログラムに異常な動作を引き起こす攻撃が深刻な問題となっている．このような攻撃が成功すると情報漏洩やデータの改ざんが発生する可能性があり，非常に危険なものであると認識されている．

この問題を解決する手段として，強制アクセス制御機構を備えたセキュア OS [1][2] の使用が挙げられる．強制アクセス制御とは，リソースへの正常なアクセスを定義したセキュリティポリシをあらかじめプロセスごとに用意し，それに基づいて実行時にアクセス制御を強制するものである．この機構を用いることによって，バッファオーバーフロー攻撃等によりプログラムの制御を奪われたとしても，セキュリティポリシで許可されないリソースへのアクセスを防ぐことが可能となる．しかし，多くのセキュア OS ではアクセス制御を定義するセキュリティポリシが非常に複雑であるため，すべてのプログラムに対して正確にセキュリティポリシを記述するのは困難であり，管理者にとって負担の大きい作業である．

このような管理者の負担を軽減し，セキュリティポリシを正確に記述するために，コンパイラが生成する静的解析情報を利用するセキュアシステム [3][4] を提案している（以下,「既存システム」とする）．既存システムでは，監視対象となるプログラムのソースコードをコンパイル時に解析することにより，プログラムの挙動を記述したセキュリティポリシを生成する．しかし，既存システムを含む静的解析情報を利用する一般のセキュアシステムには，システムコール引数値の改ざんやスタック偽装による攻撃を検知できない可能性がある．

本稿では，上記の検知できない攻撃に対処するため，データフロー解析結果に基づきシステムコール引数値を固定する手法を提案する．提案手法を用いることによって，実行時に定義されたシステムコール引数値とシステムコール発行時に実際に使用された値を比較することが可能となり，攻撃者によるシステムコール引数値の改ざんが検知可能となる．

2 コンパイラと OS の連携によるセキュアシステム

2.1 概要

既存システムの動作概要を図 1 に示す．既存システムはソースコードのコンパイル時に，静的解析により実行時監視のために必要な情報を抽出し，セキュリティポリシに記述する．現在の既存システムにおいて，セキュリティポリシに記述される情報は，システムコールの種類と発行アドレス，引数と関数呼び出しの戻りアドレスの 4 種類である．OS 内に存在するシステムコール監視部では，監視対象プロセスがシステムコールを発行したとき，セキュリティポリシに基づいて上記 4 種類の観点から検査を行う．この検査において，セキュリティポリシに記述されていないシステムコールを異常なシステムコールとして検知し，そのプロセスの実行を中断する．既存システムはセキュリティポリシを自動的に生成するため，管理者による記述ミスの防止やシステム運用時の負担軽減が期待できる．

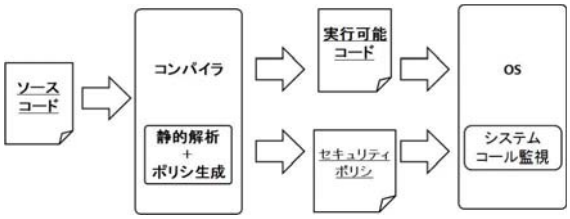


図 1: 既存システムの動作概要

```
1 int main(int argc, char *argv[]){
2     char filename[MAX_FILE_NAME],buf[16];
3     ...
4     strcpy(filename, argv[1]);
5     ...
6     /* Buffer Overflowによる値の上書き */
7     strcpy(buf, argv[2]);
8     ...
9     fd = open(filename, ...);
```
図 2: システムコール引数値の改ざんが生じる例

2.2 問題点

既存システムで用いるセキュリティポリシでは，実行時のシステムコール引数値の改ざんに対処するため，ソースコード中でシステムコール引数が定数の場合にはその値を記述している．しかし，システムコール引数が変数の場合には，実行時にいかなる値もとり得るとセキュリティポリシに記述し，実行時検査においてすべての値を許可している．そのため，図 2 に示すようにシステムコール引数である変数に 4 行目で正常な値が定義された後，実際にその変数を使用する 9 行目までの間でその変数が改ざんされたことを検知できない．また，攻撃者がスタック偽装によって，プログラマの意図した制御フローを経ずにスタック上にシステムコール引数値を設定していたとしても，スタック上の各関数とシステムコールのリターンアドレスに基づく呼び出し関係がセキュリティポリシに記述されたとおりであれば，システムコール監視部はその攻撃を検知できない．つまり，バッファオーバーフロー等のプログラマの意図しない方法でシステムコール引数値が定義されたとしても，システムコール監視部はそのシステムコールの発行を認める場合があるという問題点がある．このような攻撃の見逃しは，静的解析情報を利用する他のセキュアシステムにおいても発生する．

Improvement of Accuracy of Intrusion Detection by Protecting System Call Arguments
Satoshi Yamasaki†, Hiroaki Kuwabara‡and Yoshitoshi Kunieda ‡
†Graduate School of Science and Engineering, Ritsumeikan Univ.
‡College of Information Science and Engineering, Ritsumeikan Univ.

3 システムコール引数値の固定手法

本章では，2.2 節で述べた問題点を解決するために，実行時にシステムコールの引数値を保護する手法を提案する．

3.1 システムコール引数値の固定位置

提案手法では，システムコール引数値が定義された直後に，定義された値をシステムコール監視部に通知するシステムコールである bind_sys_arg を挿入することにより，システムコール発行時に使用できる引数値を通知した値で固定する．システムコール引数値の定義点はソースコードをデータフロー解析により求める．例えば図 2 では，bind_sys_arg の挿入位置は 4 行目の strcpy 関数の直後となる．ここで，一度定義された変数が複数のシステムコールで使用される場合，複数個の bind_sys_arg を挿入してそれぞれのシステムコール引数値を固定する．固定した値自体を攻撃者によって改ざんされることを防ぐため，bind_sys_arg によって通知された値は OS 内部のシステムコール監視部で管理する．システムコール監視部は，システムコール検査の際に定義点での値とシステムコール発行時の引数値を比較することで引数値の改ざんを検知する．また，提案手法で挿入した bind_sys_arg の発行を確認することで，監視対象のシステムコールの引数値を定義するための制御フローを経たことを確かめる．

3.2 固定した値の解放

固定した値がシステムコール発行後もシステムコール監視部に残っていると，固定した値を用いてそのシステムコールを発行することを許可する状態が続く．この状態を攻撃者に悪用されることを防ぐため，使い終わったシステムコール引数値についてはシステムコール監視部から解放する必要がある．しかし，プログラム中に反復構造が存在し，一度固定した値を複数回使用する場合，システムコール終了後に即座に使用した値を解放することはできない．なぜなら，2 度目以降のシステムコール発行において，システムコール引数値が未定義の状態でシステムコールが発行されたという誤検知が発生するためである．

提案手法ではシステムコール引数値が定義されるスコープとシステムコールが発行されるスコープに基づいて固定した値の解放位置を決定する．この 2 つのスコープが同じスコープであった場合，システムコール発行後に即座に使用した値を解放する．これは，反復構造によって再びシステムコールが発行されるときには，引数値の定義についても再度行われるためである．2 つのスコープが異なるスコープであった場合，システムコールが発行されるスコープを抜けた段階で使用した値を解放する．これは，システムコールが発行される内側のスコープにおいて，再度引数値を定義することなく，複数回システムコールが発行される可能性があるためである．内側のスコープが分岐であった場合には，システムコールが存在する側の節ではシステムコール発行後に即座に値を解放し，システムコールが存在しない側の節ではその節の開始時に値を解放することで，攻撃者による悪用が可能な区間を短くする．

```
1 strcpy(filename, argv[1]);
2 bind_sys_arg(00000100,1,filename);// 引数値の固定
3 ...
4 /* ライブラリ関数の引数値の検査 */
5 check_arg(filename, mode);
6 fd = fopen(filename, mode);
```

図 3: ライブラリ関数の引数値の保護

3.3 ライブラリを考慮したシステムコール引数値の保護

多くのシステムコールはライブラリ関数を経由して発行されるため，システムコールの引数値を保護するためにはライブラリ関数の振る舞いを考慮して引数値を固定しなければならない．しかし，多くのライブラリ関数ではポインタやインラインアセンブラが用いられているため，静的解析によってその振る舞いを求めることが困難である．また，ライブラリ関数内に新たにシステムコールを挿入した場合，ライブラリの動的リンクに対応できなくなる等の問題が発生する．

提案手法ではライブラリ関数の引数値を保護することによって，ライブラリ関数を経由して発行されるシステムコールの引数値を保護する．具体的には，システムコール引数値と同様にライブラリ関数の引数値をシステムコール監視部に通知した後，図 3 の 5 行目に示すように，ライブラリ関数の呼び出し前にその引数値を検査するためのシステムコールを挿入する．これによって，実行時にシステムコール監視部が，ライブラリ関数の引数値が改ざんされていないことを確認する．

4 おわりに

本稿では，システムコール引数値を保護することによって，その値がプログラマの意図しない方法で定義されたことを検知する手法について述べた．今後は，提案手法を既存システムに実装し，大きなオーバーヘッドが予想される大規模プログラムでの評価を行い，実行時のオーバーヘッドを削減する方法を検討する．

参考文献

[1] Security-Enhanced Linux. http://www.nsa.gov/selinux/.

[2] TOMOYO Linux. http://tomoyo.sourceforge.jp/.

[3] 表雄仁, 森山壱貴, 桑原寛明, 毛利公一, 齋藤彰一, 上原哲太郎, 國枝義敏. コンパイラと OS の連携による強制アクセス制御向けプロセス監視手法. In *Computer Security Symposium 2007(CSS2007)*, 第 10 巻, pp. 109–114, 2007.

[4] 森山壱貴, 表雄仁, 桑原寛明, 毛利公一, 齋藤彰一, 上原哲太郎, 國枝義敏. コンパイラと OS の連携による強制アクセス制御向け静的解析. In *Computer Security Symposium 2007(CSS2007)*, 第 10 巻, pp. 103–108, 2007.

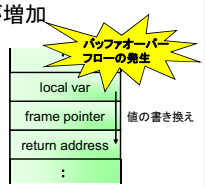
システムコール引数値の保護による侵入検知精度向上

立命館大学大学院 理工学研究科
高性能計算機ソフトウェアシステム研究室
山崎悟史 桑原寛明 國枝義敏

背景

- プログラムの脆弱性を利用した攻撃が増加
  - バッファオーバーフロー脆弱性
    - アタックコードの実行

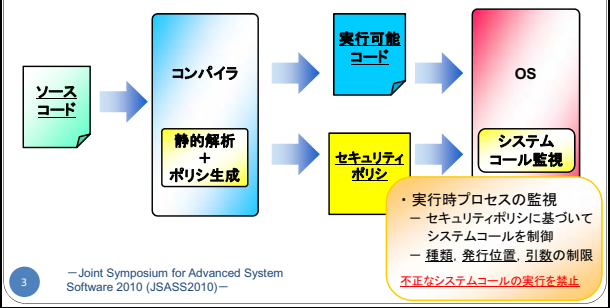
  ⇒ データの改竄や情報漏洩



- セキュアOSによる対策
  - セキュリティポリシを基にした強制アクセス制御
  - 設定が難しく煩雑
    - 管理者の負担，ポリシの記述ミス

コンパイラとOSの連携によるセキュアシステム

- コンパイル時にセキュリティポリシを自動生成



・実行時プロセスの監視
― セキュリティポリシに基づいてシステムコールを制御
― 種類，発行位置，引数の制限
不正なシステムコールの実行を禁止

生成するセキュ

・s エントリ
― システムコールの戻りアドレス(発行位置)
― システムコール番号(種類)
― システムコール引数

```
int main( ){
  ...
  fd = open("/etc/passwd", O_RDONLY);
  rcount = read(fd, buf, buf_size);
  close(fd);
  ...
  func(filename, buf);
  ...
}

func(){
  ...
}
```

```
f  00401050  001  main
s  00401094  005  "/etc/passwd" 00
s  004010b0  003  fd buf buf_size
s  004010be  006  fd
...
f  004010d8  002  func
r  004010d1  001<main>
...
```

・r エントリ
― 関数の戻りアドレス(呼び出し関係)

セキュリティポリシ

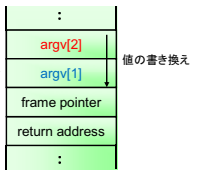
現在のシステムでは検知不可能な攻撃が存在
⇒ セキュリティポリシを拡張することで検知を可能にする

検知不可能な攻撃(1)

- システムコール引数値の改ざん

```
...
strcpy(filename, argv[1]);
...
strcpy(var, argv[2]);
...
fd = open(filename,
     O_WRONLY|O_CREAT|O_TRUNC, 0666);
wcount = write(fd, buf, buf_size);
close(fd);
...
```

バッファオーバーフロー



⇒ 任意のファイルへの書き込みが可能

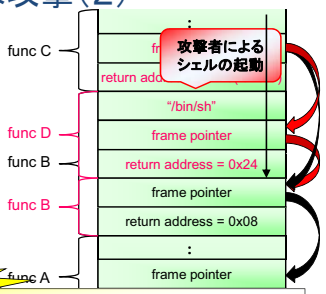
定義点の値を記憶しておくことで改ざんを検知可能

検知不可能な攻撃(2)

- スタック偽装

```
func_A:
0x00 ・・・
0x04 call func_B
0x08 ・・・

func_B:
0x10 if
0x14   call func_C
0x18 ・・・
0x1C else
0x20   call func_D
0x24 ・・・

func_C:
0x30 /*バッ
```

バッファオーバーフロー



引数値定義のための制御フローを経ていることを確認することでスタックの偽装を検知可能

提案手法

- システムコール引数値の定義時点の値を保存する
  - コンパイル時
    - データフロー解析により定義点を求める
    - 定義点直後にシステムコールを挿入
      - システムコール発行時に使用できる引数値を固定
  - 実行時

・新たなシステムコールを実装
　－ 定義時の値をカーネル領域に保存

引数値と定義時の値を比較

```
...
strcpy(filename, argv[1]);
bind_sys_arg(00000100, 1, filename);
...
strcpy(var, argv[2]);
...
fd = open(filename,
    O_WRONLY|O_CREAT|O_TRUNC, 0666);
wcount = write(fd, buf, buf_size);
close(fd);
```

```
...
f  004010d8  0
s  004010f8  00
a  2  1  "file...
a  1  0  01101
a  1  0  0666
```

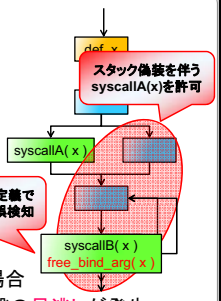
システムコール発行時に使用が許可される値

ポリシ作成時点では変数

・a エントリ
　－ 引数の型（1：整数，2：文字列，3：ポインタ）
　－ 実行時に値が決定するか（true：1，false：0）
　－ 値（ポリシ作成時に定数）

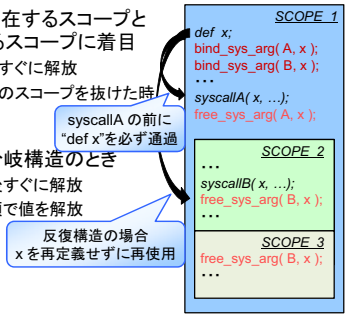
固定化した値の管理

- 再帰関数への対応
  - 値の世代管理
  - 新たなシステムコールによる通知
- 固定化した値の解放
  - システムコールが実行されなかった場合
    ⇒ 前回定義した値が残っていれば攻撃の見逃しが発生
  - 一度固定した値を複数回使用する場合
    ⇒ 単純に値を解放すると誤検知が発生



固定化した値の解放のタイミング

- システムコールの存在するスコープと引数値が定義されるスコープに着目
  - 同じスコープ：使用後すぐに解放
  - 異なるスコープ：内側のスコープを抜けた時点で解放
- 内側のスコープが分岐構造のとき
  - 発行される側：使用後すぐに解放
  - 発行されない側：先頭で値を解放



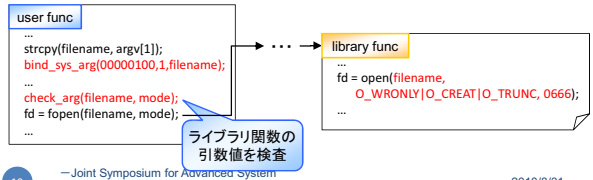
syscallA の前に“def x”を必ず通過

反復構造の場合
x を再定義せずに再使用

ライブラリを考慮した引数値の保護

- 多くのライブラリ関数は静的解析困難
  - ポインタ，インラインアセンブラ

ライブラリ関数の引数値を保護することによって
ライブラリ関数を経由するシステムコールの引数値を保護



ライブラリ関数の引数値を検査

† Timothy K. Tsai and Navjot Singh. Libsafe: Protecting Critical Elements of Stacks. Aug 2001

評価

- 環境

CPU	Core Duo T2300 1.66GHz
Memory	1,024 MByte
OS	Fedora 10
Kernel	Linux kernel-2.6.27

- 対象
  - wc_nolib
    - ライブラリを使用しないワードカウントプログラム
    - 入力：20MBの英文

評価 – wc_nolib

- システムコール挿入数
  - bind_sys_arg：13，free_sys_arg：9
- 発行システムコール内訳

システムコールの種類	read	write	open	close	execve	bind_sys_arg	free_sys_arg	合計
発行回数	1240	8	1	1	1	19	11	1281

- 実行時間

	実行時間(msec)	オーバーヘッド割合
通常カーネル	240.890	1
リターンアドレス検査のみ	244.985	1.017
リターンアドレス検査＋引数検査	246.113	1.022

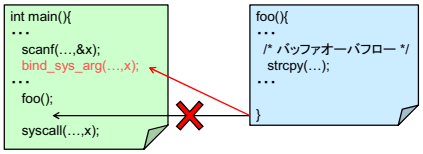
考察 - 追加システムコールの保護

- 新たなシステムコールも他のシステムコールと同等
  - セキュリティポリシに基づき実行時に検査
    - リターンアドレス検査：発行位置の制限
    - 引数値検査：固定化対象の制限

システムコール名	機能	第1引数	第2引数	第3引数
bind_sys_arg	定義された引数値の保存	long syscall_ID	long nth	value
		定数	定数	変数
free_sys_arg	保存されている引数値の解放	long syscall_ID	long nth	×
		定数	定数	×
check_arg	引数値の検査	long syscall_ID	value	…
		定数	変数	…

考察 - 引数固定化手法における見逃し

- 固定化前の値の改ざん
  - 引数値が定義される式の右辺値が既に汚染されているとき
- 子関数から親関数へのリターン



まとめ

- システムコールの引数値を保護する手法を提案
  - 静的解析により引数値の定義点を求める
  - 定義された引数値を使用するときに定義点の値と比較
  - ライブラリを使用しないwcで2.2%のオーバーヘッド
- 今後の課題
  - 見逃しへの対応
  - オーバーヘッド削減
  - システムコール挿入の自動化

Linux Security Module を用いた Privacy-aware OS *Salvia* の実装

鍛治 輝行　6171090017-4　*tkaji@asl.cs.ritsumei.ac.jp*
立命館大学大学院理工学研究科　毛利研究室

1　はじめに

近年，機密情報が電子化され，その情報が計算機で管理されている．この機密情報が，記憶媒体やネットワークを通じて，漏洩する事件が多発している．こうした事件は，悪意のあるユーザによる侵入攻撃や，正当な権限を持つユーザによる管理ミスや誤操作によって発生している．

我々は，こういった情報漏洩事件を防止するセキュア OS として，Privacy-aware OS *Salvia*[1] の開発を行っている．本稿において，セキュア OS は，「アクセス制御機能を強化したオペレーティングシステム」であると定義する．Linux 用のセキュア OS には，SELinux や TOMOYO Linux をはじめとする様々な種類が存在する．

こういった様々なセキュア OS の導入を容易にするため，Linux には，Linux Security Module[2](以下，LSM と略す) というフレームワークが導入された．セキュア OS を LSM に対応したカーネルモジュールとして開発することで，Linux にセキュア OS の機能を容易に付加することが可能となる．

先に述べた，SELinux や TOMOYO Linux は，LSM に対応する形で実装されている．それに対し，*Salvia* は，Linux カーネルのソースコードに変更を加えることで実装しているため，カーネルコンパイルなどをする必要があり，導入が困難であるという問題点がある．

そこで，ユーザによる *Salvia* の導入を容易にするため，LSM に対応する形で開発を行っている．LSM は，カーネルソースに複数のフックポイントを挿入しており，そのフックポイントにおいて制御を行うことにより，セキュア OS のアクセス制御が可能となる．しかし，LSM のフックポイントは，アクセス制御を行う上で十分な数が挿入されているとはいえない．そのため，現在，LSM に対応した *Salvia* を実装するにあたり，LSM を拡張する形で実装を行っている．

以下，本稿では，2 章で LSM の概要について述べる．3 章では，*Salvia* の概要について述べる．4 章では，*Salvia* を LSM に対応させるための実装について述べる．5 章では，4 章で述べた実装から得られた知見に対する考察について述べ，6 章で，本稿のまとめとする．

2　Linux Security Module

LSM は，カーネルに様々なセキュリティモジュールの導入を容易に行えるように策定されたフレームワークである．LSM は，カーネル内の様々な処理をフックし，登録されているセキュリティモジュールへのコールバック関数の呼出を行う．それぞれの処理のフック後，作成した関数を呼び出すことにより，独自のアクセス制御を行うことが可能となる．LSM によって処理をフックできるポイントは，カーネル内に 150 以上挿入されている．LSM によってフックが行える主な処理を以下に述べる．

- i ノードへのアクセスの可否判定
- ファイルへのアクセスの可否判定
- Linux Kernel Module のロードの可否判定
- プロセス生成の可否判定

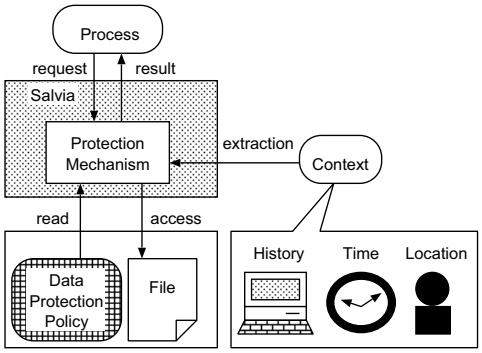


図 1　*Salvia* のデータ保護モデル

LSM は，Linux カーネル 2.6 以降，正式に採用されている．そのため，LSM に対応したセキュリティモジュールは，Linux カーネル 2.6 以降であれば，容易に導入が可能となる．

3　Privacy-aware OS *Salvia*

現在，我々が開発を行っている *Salvia* は，OS で情報の漏洩を防止する．*Salvia* は，ファイルを保護の対象としており，そのファイルをオープンしたプロセスを制御の対象としている．*Salvia* は，この制御対象のプロセスに対し，ファイル，ソケット，パイプなどの計算機資源へのアクセスを制御することにより，データの漏洩を防止する．OS でプロセスを監視することにより，アプリケーションの信頼性に関わらず，統一的な制御が可能であるという利点がある．プロセスは，保護対象のファイルごとに用意した保護ポリシに基づき，制御される．ファイルごとに利用目的や提供範囲は異なるため，保護ポリシもファイルごとにデータ提供者が用意することにより，データ提供者の意志を反映することが可能となる．*Salvia* は，ファイルアクセス時に，コンテキストを取得し，保護ポリシに記述された保護方法と比較することで，計算機資源へのアクセスの可否を判定する．プロセスを制御する際に用いるコンテキストには，端末の位置，現在の時刻，ユーザ ID，過去の動作履歴などがある．図 1 に，*Salvia* のデータ保護モデルを示す．図中の保護機構では，プロセスが発行するシステムコールを制御することにより，アクセス制御を実現している．

4　LSM に対応した Privacy-aware OS *Salvia* の構築

本章では，*Salvia* のアクセス制御に必要な処理と，その処理をするために必要な LSM フックについて述べる．

4.1　ファイルのオープン

Salvia は，保護ファイルをオープンしたプロセスに制御を課す．そのため，ファイルのオープン処理をフックし，オープン対象ファイルが保護ファイルかどうかを判別する必要がある．保護ファイルであった場合，対応する保護ポリシを取得し，プロセスを制御対象とする．取得したポリシは，カーネル内に確保したメモリに，リストで管理を行う．ファイルのオープン処理の

フックには，LSM の inode_permission() フックを用いる．ファイルをオープンする際，そのファイルに対応する i-node へのアクセスが行われる．inode_permission() フックは，その i-node へのアクセスの可否を判定する箇所に挿入されており，ファイルがオープンされる際，必ず一度呼び出される．ここで上記の処理を行うことにより，プロセスの制御をすることができる．

4.2 ファイルの読み書き

制御対象プロセスがファイルへ読み書きを行う際，そこから情報漏洩が発生する可能性がある．そのため，この処理をフックしファイルへのアクセスを制御する必要がある．ファイルへの読み書きは，file_permission() フックを用いることにより可能となる．このフックポイントで，ファイルの読み書きの処理をフックし，読み書きを行うプロセスが，制御プロセスか否かの判別を行う．制御プロセスであった場合，ファイルのオープン時に取得した保護ポリシに基づき，アクセスの可否を判定する．

4.3 プロセス間通信

Salvia では，ファイルへの読み書きだけでなく，同一計算機内のプロセス間通信の制御 [3, 4] も行う．例えば，書き込み禁止のファイルから読み込んだデータを保持したプロセスが，プロセス間通信によって別プロセスにデータを渡した場合，データを受け取ったプロセスがファイルに書き込むことにより，情報が漏洩する可能性があるためである．プロセス間通信の中でも，UNIX で用いられる代表的なプロセス間通信であるパイプ，共有メモリ，ソケット，FIFO の実装を行っている．

パイプや FIFO は，S_IFIFO という特殊な種類のファイルへ読み書きを行うことでデータの共有を行う．S_IFIFO の書き込みは通常のファイルの書き込みと同様，file_permission() フックを用いることにより可能となる．ここでアクセスするファイルの種類を調べることにより，パイプや FIFO への読み書きを区別し制御を行う．

共有メモリへの読み書きは，システムコールを介さずに行われ，カーネルが検知できない．そのため，プロセスが共有メモリにアタッチした際，その共有メモリを監視する．監視中の共有メモリに対するアクセスで制御をする事で，共有メモリアクセスの可否が判定可能となる．共有メモリのアタッチは，shm_shmat() によってフックが可能であるため，ここでアタッチする共有メモリを書き込み禁止に設定する．

ソケット通信は，ソケットファイルの作成，通信先の決定，データの送信という処理の流れで実行される．これらの処理の中で，実際にデータが漏洩する可能性がある処理であるデータの送信処理をフックし制御することで，情報漏洩を防止することが可能である．ソケット通信におけるデータの送信は，socket_sendmsg() によってフックが可能である．ここで送信先の IP アドレスやポートなどを調べ制御を行う．

4.4 保護ポリシ

Salvia では，保護したいファイルに対応する保護ポリシを用意することで管理を行う．保護ポリシには，保護ファイルにアクセスするプロセスに課す制御とコンテキストを記述する．記述は XML 形式で行い，作成後，パーサを通して変換を行い保護ファイルとセットで管理を行う．これまでは，変換後の保護ポリシは別のファイルに書き込む事で管理をおこなっていた．現在は，変換後の保護ポリシは保護ファイルの拡張属性に格納している．この手法の利点として，ポリシがユーザの目に触れず，また管理者権限が無い場合アクセスできないというセキュリティ上の利点と，ファイルよりも高速に読み込むことができるという性能面の利点がある．図 2 に，*Salvia* の保護ポリシの記述例を示す．

```
<data_protection_policy>
 <default_access>
  <read>deny</read>
  <write>
  <write_access update="deny">deny</write_access>
  </write>
 </default_access>
 <data_protection_domain type="none">
  <ACL>
    <context>
      <group>
        <group_id type="own">1001</group_id>
      </group>
    </context>
    <access>
      <read>allow</read>
    </access>
  </ACL>
  </data_protection_domain>
</data_protection_policy>
```

図 2　*Salvia* の保護ポリシ

5 考察

Salvia を LSM に対応させる形で実装していくにあたり，カーネルソースに変更を加えて実現を行った箇所がある．それらの箇所は，共有メモリのデタッチ時や，プロセスの終了時である．LSM フックは，ファイルのアクセス前や，共有メモリアクセス前といった，情報の漏洩が発生する可能性がある処理の前に挿入されている．アクセス制御を行う上では，十分なフックが挿入されていると思われる．しかし，アクセス前に課した制御を取り除く際には，十分なフックが挿入されているとは言えない．

6 終わりに

本稿では，セキュア OS を Linux カーネルに容易に実装可能にするフレームワークである LSM について述べた．また，このフレームワークに基づき開発を行っている *Salvia* の実装について述べた．実装を進めるにあたり，LSM のフレームワークでは実現できない箇所が存在したため，その部分については，LSM を拡張する形で実装を行った．今後，更に実装を進め，その中で得られた知見を元に，様々なセキュリティモジュールに適したフレームワークの提案を行う．

参考文献

[1] 鈴来 和久, 一柳 淑美, 毛利 公一, 大久保 英嗣: Privacy-Aware OS *Salvia* におけるデータアクセス時のコンテキストに基づく適応的データ保護方式, 情報処理学会論文誌: コンピューティングシステム, Vol. 47, No. SIG3, pp. 1-15, 2006.

[2] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, Greg Kroah-Hartman : Linux Security Modules:General Security Support for the Linux Kernel, In Proc. of the 11th USENIX Security Symposium, pp. 17-31, 2002.

[3] 廣 真文: プロセス間のデータフローを制御するための共有メモリアクセス制御手法, 修士論文, 2007.

[4] 福井 一馬: データ保護を目的としたパイプのアクセス制御手法, 修士論文, 2007.

Linux Security Module を用いた Privacy-aware OS *Salvia* の実装

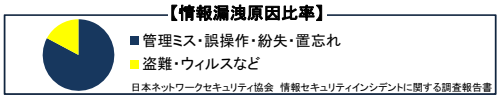
立命館大学大学院
毛利研究室
鍛治　輝行

発表内容

- 研究背景
- Privacy-aware OS *Salvia* の概要
- Linux Security Module の概要
- LSMを用いた*Salvia*の実装
- デモ
- 今後の課題

研究背景

- 電子化されたプライバシ情報の漏洩事件が多発
  - 正当な権限を持つ者による漏洩が原因の多数
- 既存の暗号化，認証，侵入検知などの技術では正当な権限を持つ者による漏洩を防ぐ事が困難



正当な権限を持つ者による情報漏洩を防ぐ為，Privacy-aware OS *Salvia*の開発

Privacy-aware OS *Salvia*

- アクセス制御機構を備えたOS
  - アプリケーションの信頼度に関わらず統一的に制御可能
- 保護方式
  - 制御対象：プロセス
    - データを漏洩させる動作（システムコール）を制御
  - 保護単位：ファイル
    - 保護したいファイルと保護ポリシを一組にして管理
  - コンテキスト（ユーザや計算機の状況）の利用
    - ユーザ，時間，計算機の場所，IPアドレス　など

*Salvia*の課題

- Linuxのカーネルソースに変更を加え開発
- *Salvia*導入時の制約
  （再構築が必要・バージョンが限定）

⇩

- 制約を解消し，
  *Salvia*の導入を容易にするため，
  *Salvia*をLinux Security Moduleに対応させる

Linux Security Module(LSM)

- カーネルにセキュリティ機能を拡張するためのフレームワーク
- カーネルの再構築無しに拡張可能
- Linuxカーネル2.6以降，正式に採用
- カーネルには，セキュリティモジュールへのコールバック関数呼び出しが挿入されている

LSMを用いた制御処理の流れ

- LSMのフックに対応する
  制御関数を登録する事で，制御を行う

主要なLSMフック

- OPEN
  - ファイルのオープン時に保護ポリシを取得
  - inode_permission(struct inode *inode , ...)
- READ
  - ファイルの読み込みを制御
  - file_permission(struct file *file, int mask)
- WRITE
  - ファイルへの書き込みを制御
  - file_permission(struct file *file, int mask)

LSMを用いた*Salvia*の機能（1/2）

- プロセスに対するアクセス制御
  - ファイルへの読み書き
  - 同一計算機内のプロセス間通信
  - 他計算機へのプロセス間通信
- コンテキストの利用
  - 時間（絶対時刻・相対時刻）
  - 書き込み先（リムーバブルメディア）

LSMを用いた*Salvia*の機能（2/2）

- ポリシによる設定
  - 保護ファイルに対応するポリシを作成し管理
- 保護ポリシ
  - 保護したいファイルへアクセスしたプロセスに
    課すアクセス制御を記述（XML形式）

Linuxカーネルの変更

- LSMのフックが挿入されていない箇所は
  カーネルに修正を加えることで対応
- LSMフックが挿入されている箇所は，
  情報の漏洩が起こる可能性がある処理の前
- アクセスした後の処理には，
  十分なフックが挿入されているとは言えない

	変更を加えたコード量	関数
Salvia	2000行以上	65関数
LSM *Salvia* (※	15行	関数呼び出しのみ

デモ

- ファイルの読み込みの制御
- ファイルへの書き込みの制御
- コンテキストに基づいた制御
  （リムーバブルメディア）

M O U R I 13

今後の課題

- *Salvia*の機能の実装
  - 他計算機へのプロセス間通信の制御
  - 履歴による制御
- ポリシの継承
  - 保護ファイルから読み込んだデータを別ファイルに書き込んだ場合，ポリシを継承し制御を引き継ぐ．

M O U R I

M O U R I 14

まとめ

- Privacy-aware OS *Salvia*
  - 正当な権限を持つ者による情報漏洩を防ぐ為
  - ポリシに基づきプロセスのアクセスを制御
  - *Salvia*導入時の制約（再構築・バージョンの限定）
- LSMを用いた*Salvia*の実装
  - 制約なしに*Salvia*を導入可能
- 今後の課題
  - *Salvia*で実装済みの機能の実装
  - ポリシの継承

M O U R I

AnT オペレーティングシステムの設計と実現

公文 宏樹 † 山内 利宏 † 谷口 秀夫 †

† 岡山大学大学院自然科学研究科

1 はじめに

ハードウェアの機能や性能を有効に利用でき，さらに多様なサービスの提供を支えるマイクロカーネル構造 OS として ***AnT*** オペレーティングシステム (以降, ***AnT*** と記述する) を開発している．この ***AnT*** について設計方針と特徴的な機能を紹介する．

2 設計方針

プログラムの設計においては，以下の 3 つを基本方針とする．

(1) 適応型構造
システムの環境に合わせて発展するために，環境を学習し，環境に適応できる機能やプログラム構造を有する．

(2) ソフトウェアの公開
開発したソフトウェアの普及を促進するため，ソフトウェアは公開する．

(3) 開発工数の削減
OS そのものの開発を重視するため，開発言語や環境は UNIX を利用する．

3 構造

プログラムは，OS とサービスからなる．OS は，内コアとプロセスとして動作する外コアからなる．サービスは，プロセスからなる．

内コアは，最小のシステムの動作を保証するプログラム部分である．外コアは，適応したシステムに必須なプログラム部分であり，動的に再構成な構造を有する．

サービスは，サービスを提供するプログラム部分である．

4 適応性と堅牢性

AnT は，計算機システムの開発者と利用者の両者に「使いやすい」かつ「壊れにくい」という特徴を有する基盤ソフトウェアであることを目指す．

開発者にとって「使いやすい」ためには，新たなサービス (機能) の追加つまり新たなプログラムの追加を用意にできる必要がある．利用者にとって「使いやすい」ためには，高性能であるとともに，利用したいサービスを簡単に利用できる必要がある．

開発者にとって「壊れにくい」ためには，開発中のプログラムの暴走によりシステム全体が壊れることを防ぐ必要がある．利用者にとって「壊れにくい」ためには，色々なサービス (機能) を使っても相互干渉によって不都合な動作にならないような高い信頼性とともに，高いセキュリティが必要である．

5 特徴的な機能

5.1 走行モード変更機能

信頼性が高くかつ頻繁利用される外コアのプログラムは，スーパバイザモードで実行する．スーパバイザモードで実行することにより，外コアのプロセスからの内コア呼び出しを直接行える．つまり，例外を発生させることなく，関数呼び出しのようにジャンプ命令のように呼び出すことができる．これにより，外コアと内コアの連携を高速化でき，ドライバ処理の高性能化を実現する．

5.2 高速なサーバプログラム間通信

外コア間の連携を行う際にコア間通信データ域 (ICA)[1] を利用する．ICA は，仮想空間のマッピング表の書き換えによりプロセス間での複写レスでの通信を可能にしている．外コア間の連携は，外コアへ渡す引数や通信制御の情報と外コアで扱うデータを別々の ICA(制御用 ICA/データ用 ICA) に格納することで，高速な通信を可能にしている [2].

6 まとめ

AnT の設計方針と特徴的な機能を述べた．特徴的な機能として，走行モード変更機構，および ICA を利用した高速なサーバプログラム間通信について述べた．

参考文献

[1] 梅本昌典, 田端利宏, 乃村能成, 谷口秀夫, “ ***AnT*** オペレーティングシステムにおけるメモリ領域管理の設計と実現 ”, 情報処理学会研究報告 2007-OS-104, pp.33-40 (2007.01).

[2] 岡本幸大, 谷口秀夫, “ ***AnT*** オペレーティングシステムにおける高速なサーバプログラム間通信機構の実現と評価 ”, 電子情報通信学会論文誌 D, vol.J93-D, no.10 (2010.10), 採録決定済.

Design and Realization of *AnT* Operating System

Hiroki KUMON†, Toshihiro YAMAUCHI†and Hideo TANIGUCHI†

†Graduate School of Natural Science and Technology, Okayama University

700-8503 Okayama, Japan

AnT オペレーティングシステムの設計と実現

公文 宏樹†1　山内 利宏†1　谷口 秀夫†1

†1 岡山大学大学院自然科学研究科

背景(1/2)

(1) 入出力ハードウェアの急速な進歩

(2) 計算機の必要性の向上に伴うサービス種別の増大

⇒ ハードウェアの機能や性能を有効活用し，多様なサービスの提供を支える基盤ソフトウェアの必要性

＜従来のモノリシックカーネル構造OSの問題点＞

(1) OSの不具合の修正のたびにカーネルの再起動の必要あり

⇒ 使いにくい

(2) 新ドライバ導入の際に他のOS機能に影響を与える可能性大

⇒ 壊れやすい

2

背景(2/2)

＜マイクロカーネル構造OSの利点＞

(1) OSの不具合の修正の際はカーネルの再起動の必要なし

⇒ 使いやすい(適応性の保有)

(2) 新ドライバ導入の際に他のOS機能に影響を与える可能性小

⇒ 壊れにくい(堅牢性の保有)

⇒ ***AnT*** オペレーティングシステムの開発

(An operating system with adaptability and toughness)

(1) ハードウェアとサービスを様々に組み合わせたシステムを管理可能

(2) ネットワークを高い通信効率とセキュリティで利用可能

(3) ソフトウェア開発をオープン化可能

3

設計方針

(1) 適応型構造

環境を学習し，環境に適応できる構造

(2) ソフトウェアの公開

(A) 見やすいコーディング(ハッシュ構造などを利用しない)

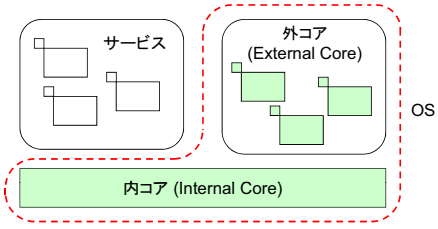
(B) 機能インタフェースの簡易化

(3) 開発工数の削減

既存のUNIX環境(開発環境やプログラム)の利用

4

基本構造



＜マイクロカーネル構造＞

- 内コア：最小システムの動作を保証する部分
  (メモリ管理，実行権制御など)
- 外コア：適応したシステムに必須な部分(OSサーバ)
  (ドライバ，ファイル管理など)
- サービス：利用者が要求する機能を提供

5

適応性と堅牢性

	使いやすい	壊れにくい
開発者	・新サービスの追加が容易 - 新プログラムの作成が容易 - 既存プログラムを流用可	・開発中プログラムの暴走によるシステム破壊の防止
利用者	・高性能 ・サービスを簡単に利用可	・プログラムの相互干渉による不具合の防止(高信頼性) ・高いセキュリティ
	高い適応性	高い堅牢性

高い適応性と堅牢性を持ち合わせた基盤ソフトウェアを目指す

6

特徴的な機能

＜*AnT* の特徴的な機能＞

(1) 走行モード変更機能

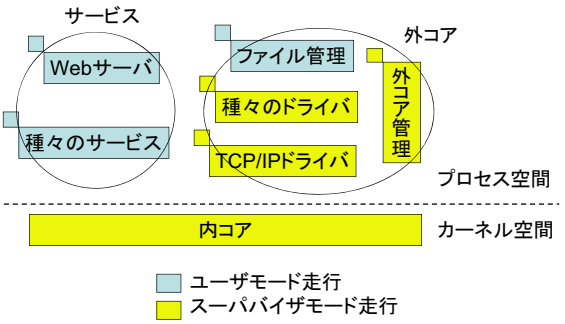
プロセスの実行を高速化するために動的に走行モードを変更することが可能な機能

(2) 高速なサーバプログラム間通信

サーバプログラム間通信の高速化のためICAを利用して複写レスでデータ授受を行う機能

7

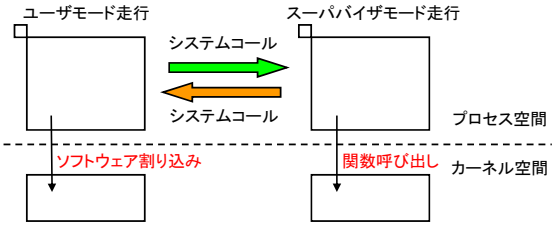
走行モード変更機能(1/2)



動的に自由に走行モード(ユーザ/スーパバイザ)の変更が可能

8

走行モード変更機能(2/2)



(1) プロセス空間中のプログラムはシステムコールにより動的に走行モード変更可能

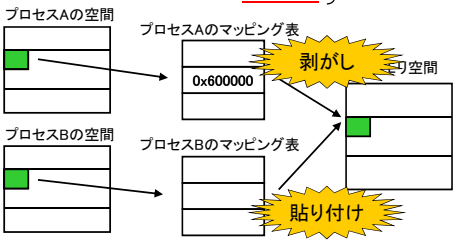
(2) スーパバイザモード走行により内コアとの連携のオーバヘッドを大幅削減

9

ICA

(Inter-core Communication Area)

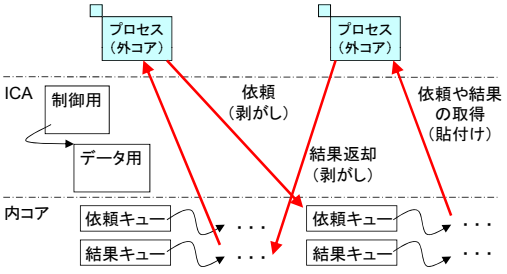
(特徴1) ページ単位による領域の確保と解放
(特徴2) 確保した領域（nページ）の実メモリ連続の保証
(特徴3) 2仮想空間の間での領域の貼り替え
マッピング表からの削除：剥がし
マッピング表への書き込み：貼り付け
} 貼り替え



10

基本機構

サーバプログラム間通信の基本機構
(1) 依頼元プロセスが依頼先プロセスの依頼キューに依頼を登録
(2) 依頼先プロセスは登録された情報を取得し処理を実行
(3) 依頼先プロセスは依頼元プロセスの結果キューに結果を登録



11

多段呼び出し

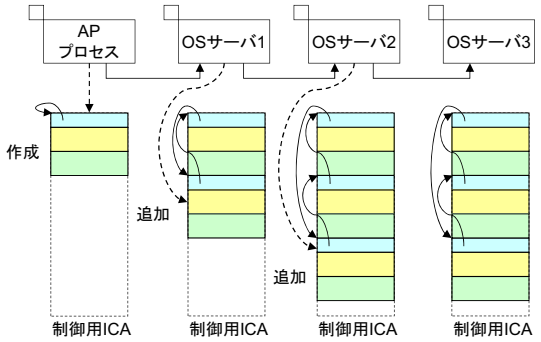
(1) OSサーバ間の連携により処理依頼が多段に発生
(2) 処理依頼の度にICAを確保する方法はオーバヘッドが大

(対処1) ひとつの制御用ICAに依頼情報を多段に格納
(対処2) 制御用ICAの先頭に最新の情報格納域へのポインタ(currentp)を持たせ依頼情報を把握
(対処3) 制御用ICAの先頭にないcurrentpは一つ前の呼び出し情報格納域へのポインタを保持

currentpにより積み重ねられた情報をたどることが可能

12

多段呼び出しの様子



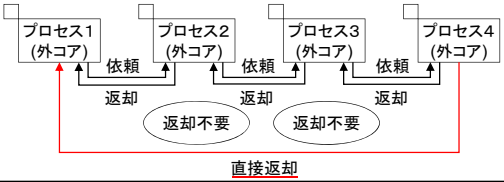
13

直接返却

＜要求＞
多段で処理依頼を渡した際の返却において
可能ならば依頼元プロセスに直接返却を行いたい

返却処理の高速化

(対処1) 依頼時に返却不要フラグを立てる
(対処2) 返却時にフラグを確認して返却する



14

まとめ

(1) ***AnT*** オペレーティングシステムの開発
(2) 3つの設計方針
  (A) 適応型構造
    (a) マイクロカーネル構造
  (B) ソフトウェアの公開
  (C) 開発工数の削減
(3) 高い適応性と堅牢性を持ち合わせた基盤ソフトウェア
  (A) 開発者と利用者の両方に使いやすく，壊れにくい
(4) 2つの特徴的な機能
  (A) 走行モード変更機能
  (B) ICAを利用した高速なサーバプログラム間通信

15

Tender オペレーティングシステムにおける世代管理機能

長井 健悟†，山内 利宏†，谷口 秀夫†

†岡山大学大学院自然科学研究科

1 はじめに

メモリ上の情報だけではなく，ファイルの情報も含めた計算機全体の動作状況を保存し，複数世代として管理できる機能（世代管理機能[1]）を ***Tender*** オペレーティングシステムにおいて設計した．本稿では，本機能の設計と実現内容および評価結果を報告する．

2 世代管理機能

世代管理機能を，以下の 4 つの機能によって構成する．

(機能 1) 世代システム

外部記憶装置上に永続化領域とは別にプレートの永続データを保存するための領域(世代保存領域)を確保しプレートの永続データを複数世代保存する．また，世代と世代保存領域の対応を世代管理表に記録する．

(機能 2) 世代保存機能

現在の計算機状態を 1 つの世代として保存する．

(機能 3) 世代復元機能

指定した世代の計算機状態を復元する．

(機能 4) 世代削除機能

指定した世代を削除する．

世代保存領域の実現方式として，以下の 2 つの方式を検討する．

(方式 1) ディレクトリ方式

永続化領域内のファイルを世代保存領域用のディレクトリに複写する．

(方式 2)アーカイブ・圧縮方式

永続化領域内のファイルをアーカイブ・圧縮して保存する．

ここでは，コピーオンライトや差分保存の実現が容易な(方式 1)を採用する．また，ファイル複写方式として，以下の 2 つの方式を検討する．

(方式 1) コピーオンライトでのファイル複写

世代保存処理や世代復元処理時に複写を行わず，ファイルの更新を行う際に複写を行う．

(方式 2) 前回の世代保存からの差分のみファイル複写

世代保存処理の際，全てのファイルを複写せず，前回の世代保存からの差分のみ保存する．

ここでは，世代保存処理と世代復元処理の両方に要する処理時間を短縮可能な(方式 1)を採用する．コピーオンライトを実現するために，ハードリンクを用いて，永続化領域と世代保存領域間でファイルを共有する．

3 評価

世代保存機能におけるコピーオンライトによる処理時間の短縮とディスク占有量の削減の効果を示す．コピーオンライト方式とファイル複写処理をファイル間複写により行う単純コピー方式を比較する．以下の手順で処理時間の測定を行った．

(1) サイズ S の仮想ユーザ空間を N 個生成する．

(2) 世代 g1(新規世代)を世代保存する．

(3) 世代 g2(差分世代)を世代保存する．

また，測定条件を以下に示す．

(条件 1)仮想ユーザ空間のサイズを変化させた場合

(条件 2)仮想ユーザ空間の合計サイズを固定させた場合

ファイル実体を複写せず，リンクのみを作成するため，コピーオンライト方式において，仮想ユーザ空間のサイズ増加に伴う処理時間の増加は非常に小さい．差分世代保存処理時間は単純コピー方式では，新規世代保存時と大差ないが，コピーオンライト方式では，新規世代保存時よりも約 4 秒増加する．

また，以下の手順でディスク占有量の測定を行った．

(1) 世代 g1(新規世代)を世代保存する．

(2) 世代 g2～g5(差分世代)を連続で保存する．

5 回目の世代保存時において，ハードリンク方式と単純コピー方式のディスク占有量は 43MB と 26MB であり，ハードリンク方式のディスク占有量は単純コピー方式の約 60%となる．

4 おわりに

永続データを複数世代保存可能にする世代管理機能について述べた．世代システムにおけるファイル複写はコピーオンライトにより行われる．また，コピーオンライト方式と単純コピー方式を比較し，世代保存機能の処理時間とディスク占有量はともに前者の方が優れていることを示した．

参考文献

[1]長井健悟，山本悠太，山内利宏，谷口秀夫：***Tender*** の世代管理機能の実現，情報処理学会研究報告，Vol.2010-OS-115,No.2,pp.1-8,2010.

The Generation Management Function on ***Tender*** Operating System

Kengo Nagai†, Toshihiro Yamauchi† and Hideo Taniguchi†

†Graduate School of Natural Science and Technology, Okayama University

Tender オペレーティングシステムにおける世代管理機能

長井健悟† 山内利宏† 谷口秀夫†

†岡山大学大学院自然科学研究科

研究背景

計算機の電源切断により主記憶上のデータや処理状態は消失

既存OS：主記憶上のデータをファイルに保存し永続化

（問題1）応用プログラム(AP)はデータを永続化する必要がある

計算機の異常終了時データを永続化し損ねる可能性あり

（問題2）APの状態は永続化しない

計算機再起動後にAP利用者はAPを再起動する必要あり

Windows：休止機能，Linux：ハイバネーション機能

主記憶上の全データを外部記憶装置に書き出す

➡（問題1)と(問題2)に対処

（問題3）利用者が実行契機を与える必要がある

予見できない緊急停止には対応できない

➡

Tender では動作継続制御により計算機処理を永続化

動作継続制御

＜基本方式＞

計算機停止前の処理を計算機再起動後に継続して実行する機能

(1) 定期的に計算機状態を外部記憶装置に書き出す

(2) OS やAP が利用する仮想記憶空間上のデータを不揮発化

＜問題点＞

最新の計算機状態のみを保存

➡異常な計算機状態の保存により，復元後に正常な走行不能

(1) ウィルスやワームなどの有害なプログラムの感染

(2) APの重大なバグの発生

➡

永続データ(計算機状態と通常のファイル)の世代管理機能を提案

➡複数時点の永続データの保存

世代管理機能の処理時間とディスク占有量を評価

Tender オペレーティングシステム

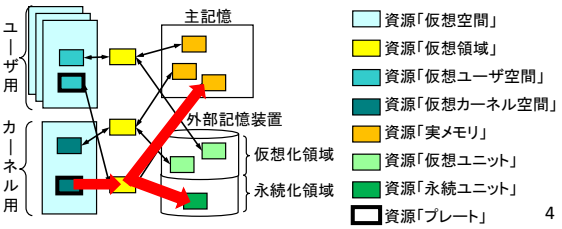
OS の各機能と管理対象を資源として細分化し，分離

＜プレート機能＞

(1) OSが自動的に仮想記憶空間上のデータを永続化

(2) 永続化の対象を資源「プレート」とし，永続化領域に保存

(3) 計算機の電源投入時にプレートを復元



Tender オペレーティングシステム

OS の各機能と管理対象を資源として細分化し，分離

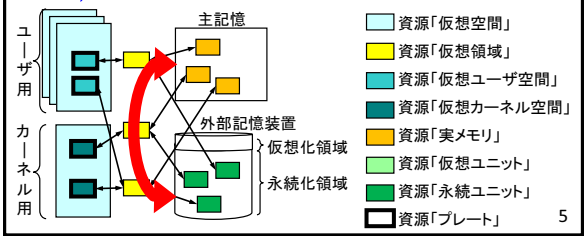
＜動作継続制御＞

(1) OSやAPが利用する仮想記憶空間上のデータをプレート化

(2) プレート化されたデータを外部記憶装置上の領域に書き出し

(3) OSの開始処理において，仮想記憶空間上にプレートを復元

➡計算機停止前の処理を計算機再起動後に継続して実行



世代管理機能への要件と要望

(要件1) プレートの永続データを計算機状態単位で複数世代管理できる

複数の計算機状態を保存するためには，プレートの永続データを計算機状態単位で複数世代管理できる必要がある

(要望1) 既存の資源管理部やAPへの変更を最小限にとどめる

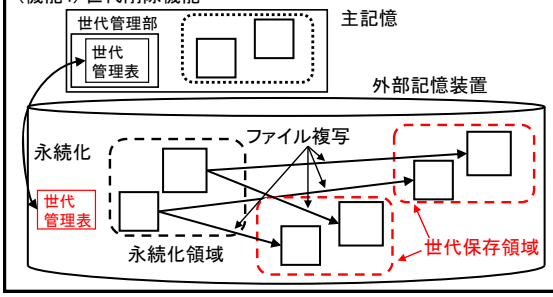
既存の資源管理部やAPの仕様に変更を加えた場合，バグの混入の原因になる

(要望2) 永続データの保存と復元に要する処理時間を削減する

永続データの保存と復元に要する時間が長くなると，他の処理の実行を妨げ，利便性が低下する

世代管理機能の構成

(機能1) 世代システム
(機能2) 世代保存機能
(機能3) 世代復元機能
(機能4) 世代削除機能



7

世代管理機能の構成

(機能1) 世代システム
(機能2) 世代保存機能
(機能3) 世代復元機能
(機能4) 世代削除機能

(1) 世代保存領域上にプレートの永続データを複数世代保存
(2) 世代情報(世代と世代保存領域の対応)を世代管理表に記録

(要件1) プレートの永続データを計算機状態単位で複数世代管理できる

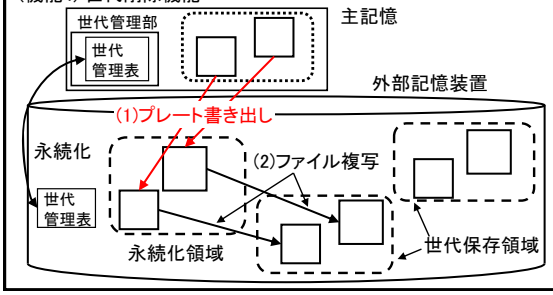
(3) 永続データの保存と復元は既存の資源管理部から独立

(要望1) 既存の資源管理部やAPへの変更を最小限にとどめる

8

世代管理機能の構成

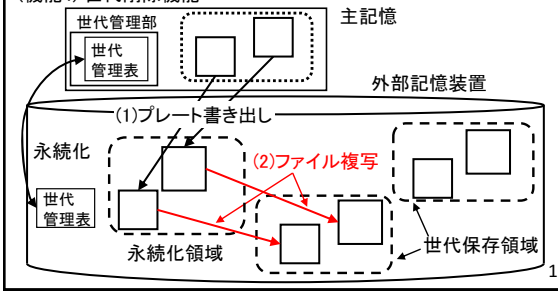
(機能1) 世代システム
(機能2) 世代保存機能
(機能3) 世代復元機能
(機能4) 世代削除機能



9

世代管理機能の構成

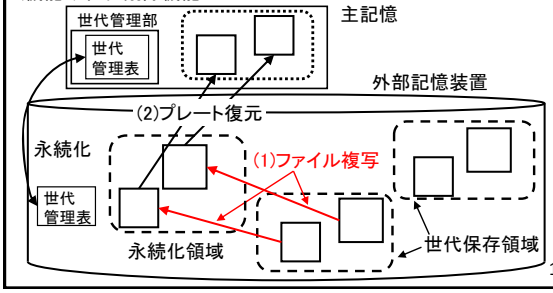
(機能1) 世代システム
(機能2) 世代保存機能
(機能3) 世代復元機能
(機能4) 世代削除機能



10

世代管理機能の構成

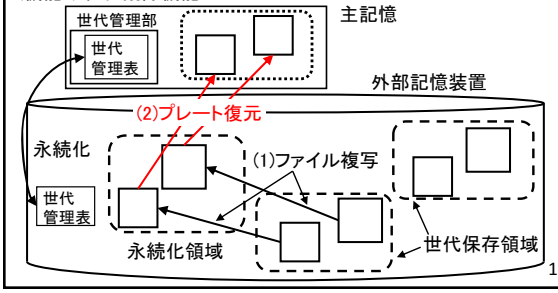
(機能1) 世代システム
(機能2) 世代保存機能
(機能3) 世代復元機能
(機能4) 世代削除機能



11

世代管理機能の構成

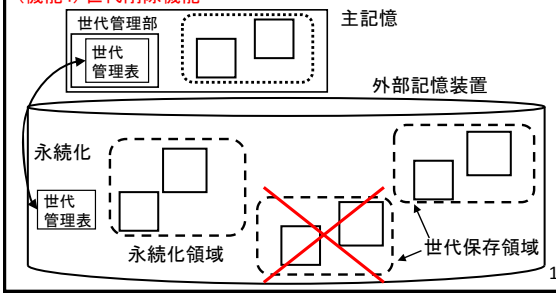
(機能1) 世代システム
(機能2) 世代保存機能
(機能3) 世代復元機能
(機能4) 世代削除機能



12

世代管理機能の構成

(機能1) 世代システム
(機能2) 世代保存機能
(機能3) 世代復元機能
(機能4) 世代削除機能



13

世代保存領域の実現方式とファイル複写方式

＜世代保存領域の実現方法＞
(方式1) ディレクトリ方式
　永続化領域内のファイルを世代保存領域用のディレクトリに複写
(方式2)アーカイブ・圧縮方式
　永続化領域内のファイルをアーカイブ・圧縮して保存
コピーオンライトや差分保存の実現が容易な(方式1)を採用

＜ファイル複写方式＞
(方式1) コピーオンライトでのファイル複写
(方式2) 前回の世代保存からの差分のみファイル複写
世代保存処理と世代復元処理の両方に要する処理時間を短縮可能な(方式1)を採用

14

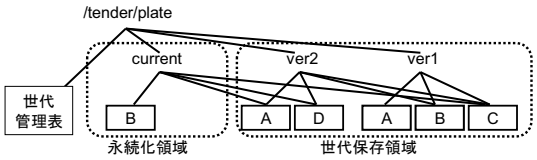
世代保存領域の実現方式とファイル複写方式

＜世代保存領域の実現方法＞
(方式1) ディレクトリ方式
　永続化領域内のファイルを世代保存領域用のディレクトリに複写
(方式2)アーカイブ・圧縮方式
　永続化領域内のファイルをアーカイブ・圧縮して保存
コピーオンライトや差分保存の実現が容易な(方式1)を採用

＜ファイル複写方式＞
(方式1) コピーオンライトでのファイル複写
(方式2) 前回の世代保存からの差分のみファイル複写
世代保存処理と世代復元処理の両方に要する処理時間を短縮可能な(方式1)を採用

15

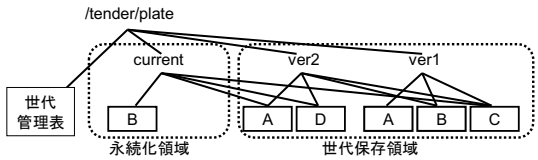
コピーオンライトによるファイル複写



ファイルシステムを利用し，各世代を保存
ハードリンクを利用し永続化領域と世代保存領域間でファイルを共有
(1)複数の名前で同一のファイルを参照可能にする機能
(2)NTFSやFFSなどの多くのファイルシステムで提供
永続データの削除はファイルのハードリンクの削除により実現
➡他の世代保存領域とファイル共有時にファイルは削除されない
ファイル更新時，リンク数によりファイルを生成するか否かを判定

16

コピーオンライトによるファイル複写



(1) 世代保存領域をディレクトリとして実現
(2) 永続化領域と世代保存領域間のファイル複写をハードリンクを用いたコピーオンライトにより高速化

(要望2) 永続データの保存と復元に要する処理時間を削減する

17

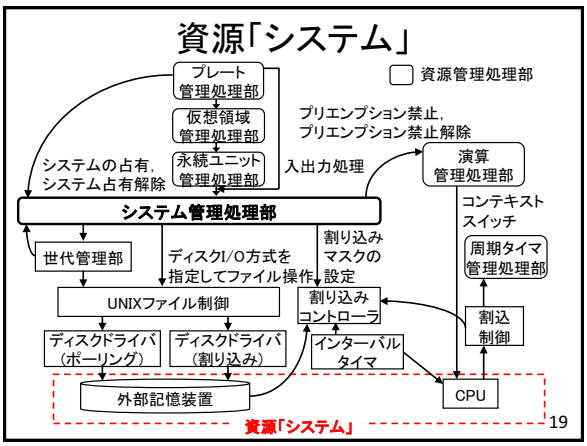
永続データ書き出しにおける要件と対処

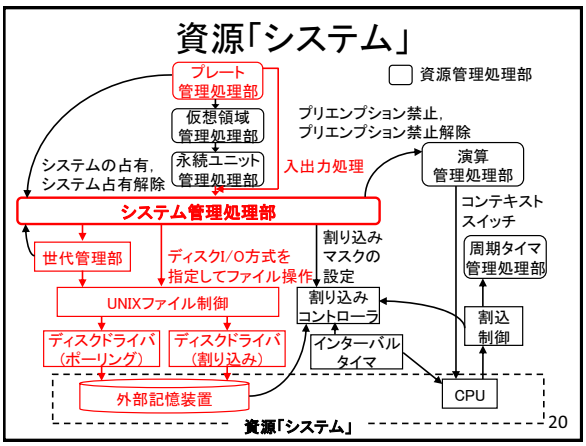
＜全プレート書き出しによる計算機処理の継続における要件＞
(要件1) データ間の整合性を保証すること
(要件2) 実入出力終了待ちのプロセスが存在しないこと
(要件3) ファイルシステムのメタデータを操作中でないこと

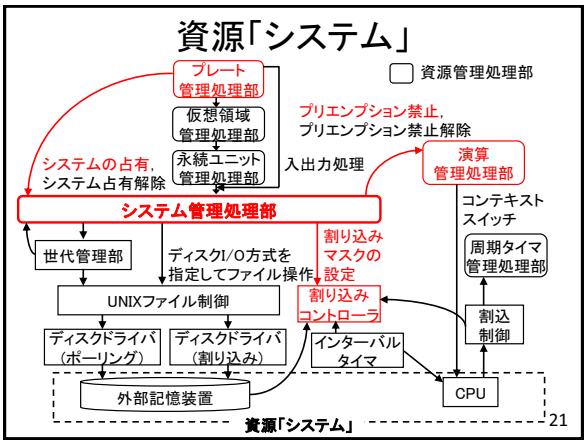
資源「システム」を実現することにより対処

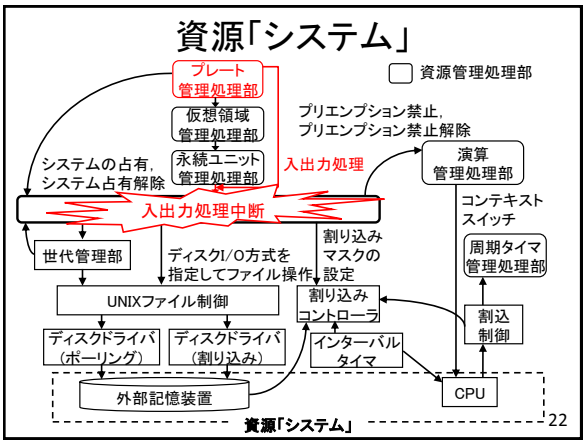
＜資源「システム」＞
ハードウェア資源の占有制御機構
(1) ハードウェア資源はプロセッサ，外部記憶装置などから構成
(2) 以下の機能を提供
(機能1) ハードウェア資源への入出力処理機能
(機能2) ハードウェア資源の占有制御機能

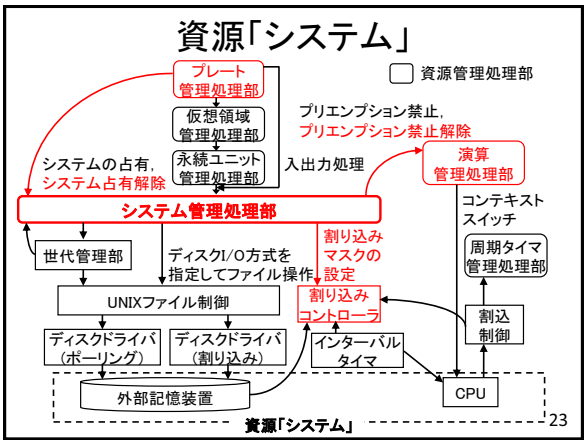
18











評価

＜評価目的＞

コピーオンライトによる処理時間の短縮とディスク占有量の削減の効果を示す

➡コピーオンライト方式と単純コピー方式を比較

ファイル複写処理をファイル間複写により行う

＜評価内容＞

(1) 単一ファイルの処理時間を測定

(2) 世代保存，復元，削除に要する処理時間を測定

(3) 世代保存時のディスク占有量を測定

＜評価環境＞

CPU	Celeron D 2.8GHz
メモリ	768MB
ハードディスク	7200rpm Ultra ATA/100 HDD
OS	*Tender*

24

世代保存機能の処理時間

＜世代保存処理時間の測定＞

複数のリンクを持つファイルの更新時，ファイルを新規生成し，更新

(1) 新規世代保存時は他世代とのファイルの共有無

(2) 差分世代保存時は他世代とのファイルの共有有

➡ 新規世代保存処理時間 ＜ 差分世代保存処理時間

(手順1) サイズSの仮想ユーザ空間をN個生成

(手順2) 世代保存処理を2回実行し，各処理時間を測定

＜評価条件＞

(条件1) 仮想ユーザ空間のサイズを変化させた場合

(1) Nを8とし，Sを0MB，0.125MB，0.25MB，0.5MB，1MBと変化

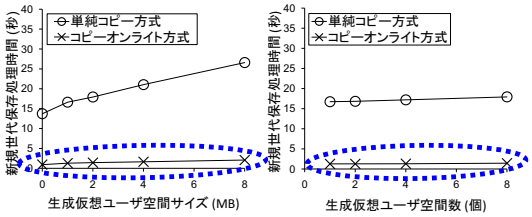
(条件2) 仮想ユーザ空間の合計サイズを固定させた場合

(1) Nを1，2，4，8 と変化

(2) S = 2MB/N とし，仮想ユーザ空間の合計サイズを固定

25

世代保存機能の処理時間（新規世代）



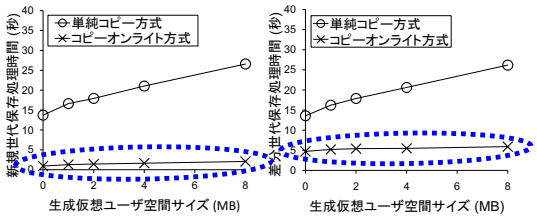
(1) コピーオンライト方式の処理時間の増加率は非常に小さく，生成する仮想ユーザ空間の合計サイズの影響を受けにくい

➡ ファイル実体を複写せず，リンクのみを作成するため

(2) 仮想ユーザ空間の合計サイズは処理時間への影響が大きい

26

世代保存機能の処理時間（差分世代）



(1) 単純コピー方式では，新規世代保存時と大差はない

(2) コピーオンライト方式では，新規世代保存時よりも約4秒増加

➡ 複数のハードリンクを持つファイルの更新処理が発生するため

（差分世代保存時までに，一部のプレートが更新されている）

27

ディスク占有量の測定

＜ディスク占有量の測定＞

複数世代保存時，全世代の合計ディスク占有量が問題

➡ コピーオンライトによるディスク占有量の削減効果を評価

＜評価手順＞

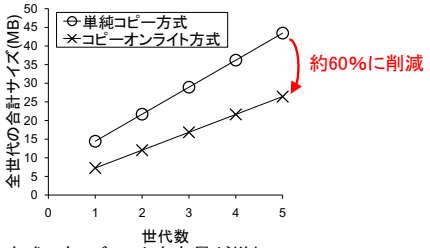
(1) 世代g1(新規世代) を世代保存する

(2) 以下の処理をn を2 から5 として繰り返す

世代 g“ n ”(差分世代) を世代保存

28

ディスク占有量



コピーオンライト方式でも，ディスク占有量が増加

➡ 差分世代保存時までに一部のプレートが更新されるため

第1回目の世代保存時から第5回目の世代保存時まで更新されないプレートがさらに10MB存在した場合：約35%に削減

29

おわりに

＜永続データの世代管理機能＞

(1)全てのプレートの永続データの複製を複数世代分保存

(2)データ複写はハードリンクを用いたコピーオンライトで実現

＜評価＞

(1) 処理時間の短縮

(A) 世代保存機能，世代復元機能，および世代削除機能の処理時間が全て減少

(2) ディスク占有量の削減

(A) 連続で世代保存を5回行った場合の占有量：60%に削減

＜残された課題＞

(1) 資源「システム」によるプロセッサと外部記憶装置以外のハードウェアの管理と占有方法の検討

(2) 他世代とハードリンクを持たない独立した世代の生成機能の検討

30

WMSNsにおける消費電力とQoSを考慮したハイブリッド型MACプロトコル

濱千代 貴大† 横田 裕介†† 大久保 英嗣††
† 立命館大学大学院理工学研究科 †† 立命館大学情報理工学部

1 はじめに

無線マルチメディアセンサネットワーク (WMSNs) は，マルチメディアを扱う無線センサネットワーク (WSNs) の一種である．WMSNs を用いることで，多くのアプリケーションが実現可能となる [1]．例えば，監視，ヘルスケア，老人補助，人物認証などがあげられる．WMSNs は，一般に複数の種類のセンサが混在するネットワークとして構築される．

このような WMSNs のアプリケーションでは，一般に，稼働時間の長期化，および優先度の高いデータ (マルチメディアデータなど) をリアルタイムに取得したいという要求がある．前者は，センサノードがバッテリで動作しているため，使用可能な電力が制限されているためである．また，後者は，監視アプリケーションにおける動画データや老人補助アプリケーションにおけるバイタルデータなど，緊急性の高いデータを対象とすることが多いからである．

そこで，WMSNs において，省電力化，および通信の QoS を考慮したプロトコルが必要となる．しかし，従来の WSNs では，省電力化が研究の中心であり，通信の QoS に関する研究が十分でない．また，インターネットで利用されている QoS 配信手法を WMSNs で用いることは，困難である．これは，帯域，電力資源，処理能力，メモリなどが制限されているため，複雑な処理を行うことができないためである．

以上の背景から，本稿では，WMSNs における消費電力と QoS を考慮したハイブリッド型 MAC プロトコルを提案する．省電力化を実現するため，サイズの大きいパケット (マルチメディアパケット) が発生した場合，送受信ノード以外の，近傍のノードをスリープさせる．また，QoS を実現するため，高優先度の通信を低優先度の通信より先行して送信することを保証する．これは，定期的に通信を停止し，優先度が高い通信が存在するか判断することで実現する．これらにより，提案手法は，マルチメディアデータと非マルチメディアデータが混在する WMSNs において，電力効率と応答性のよい通信方式を実現する．

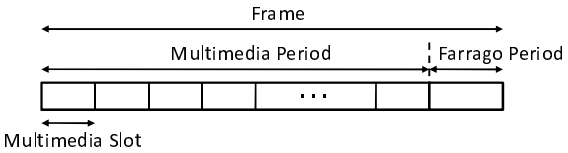


図 1 MMMAC のフレーム構成

2 MMMAC

提案する Multimedia MAC プロトコル (以下，MMMAC) は，省電力化を実現するため，マルチメディアパケットと非マルチメディアパケットで異なる通信方式を用いる．また，QoS を実現するため，バックオフを用いた優先制御を行う．以下，2.1 節で MMMAC の想定環境について述べる．また，2.2 節でマルチメディアパケットの送信，2.3 節で非マルチメディアパケットの送信について述べる．

2.1 想定環境

MMMAC では，時間がフレームを単位として分割されているとする．また，一つのフレームは，Multimedia Period(MP) と Farrago Period(FP) に分割される．さらに，MP は，複数の Multimedia Slot(MS) に分割される．MMMAC のフレーム構成を図 1 に示す．MP においてマルチメディアパケットの送信を行い，FP において非マルチメディアパケットの送信を行う．

マルチメディアデータを含むパケットは，通常のデータを含むパケットと比べ，サイズが大きい．このため，本研究では，マルチメディアパケットを，ある特定のサイズを超えるパケットとする．また，それ以外のパケットを，すべて非マルチメディアパケットとする．

2.2 マルチメディアパケットの送信

マルチメディアパケットの送信には，スロット占有方式を用いる．これは，優先権を得たノードがスロットを占有する手法である．優先権は，最初に通信を行ったノードが取得する．

マルチメディアパケットの送信手順を図 2 に示す．ま

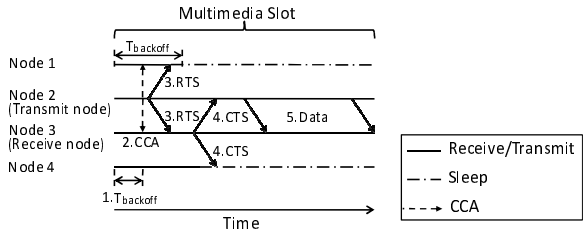


図 2　マルチメディアパケットの送信手順

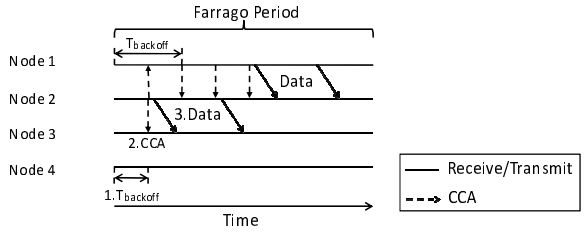


図 3　非マルチメディアパケットの送信手順

ず，送信を行いたいノード (図 2 のノード 1，2) は，バックオフ時間 $T_{backoff}$ だけ送信を待つ．$T_{backoff}$ は，通信の優先度に応じて異なり，通信の優先度が高いほど短い．これにより，優先度順に通信を行うことができる．ここで，通信の優先度は，アプリケーションによって事前に定められるものとする. バックオフタイマが切れた後，送信ノードは，Clear Channel Assessment(CCA) により帯域が空いているか否かを確認する．CCA は，受信する信号の強さに基づき，チャネルがクリアか否かを判断する技術である. 帯域が空いていれば，送信ノードは，RTS(Request To Send) メッセージを送信する．RTS メッセージを受信したノードのうち，送信先ノード (図 2 のノード 3) 以外のノードは，次の MS までスリープする. これにより，コンテンションに敗れたノード (図 2 のノード 1) や送受信を行わないノードがアイドルリスニングを行うことがなくなるため，省電力化を実現できる．RTS メッセージを受信した送信先ノードは，CTS(Clear To Send) メッセージを送信する. 送信ノードは，CTS メッセージを受信した後，データ送信を開始する. また，MS の開始から一定時間パケットを受信しないノードは，このスロットで送信するノードがいないとみなし，スリープする.

2.3　非マルチメディアパケットの送信

非マルチメディアパケットの送信には，CSMA 方式を用いる. 非マルチメディアパケットの送信手順を図 3 に示す. マルチメディアパケットの送信と同様に，まず，送信を行いたいノード (図 3 のノード 1，2) は，バックオフ時間 $T_{backoff}$ だけ送信を待つ. その後，CCA によって帯域が空いているか確認し，帯域が空いていればデータ送信を行う. ここで，マルチメディアパケットの送信と異なる点は，次の 2 点である. 1 点目は，コンテンションに敗れたノード (図 3 のノード 1) がスリープを行わない点である. このノードは，一定期間後に再度 CCA によって帯域が空いているかを確認し，空いていればデータの送信を行う. これは，FP において送信されるデータサイズが小さいため，CCA を繰り返すことによる無駄な電力消費が少ないと考えられるからである. 2 点目は，RTS/CTS を用いない点である. これは，1 点目と同様に FP で送信されるデータサイズが小さいため，再送によるコストが低いと考えられるからである.

ノード数		10
シミュレーション時間		10000 sec
マルチメディアパケット	パケットサイズ	512〜1024 bytes
	送信間隔	0.2〜4.0 sec
非マルチメディアパケット	パケットサイズ	16 bytes
	送信間隔	1.0 sec
TXPower		17.4 mA
RXPower		19.7 mA

表 1　評価に用いたパラメータ

3　初期的な評価

MMMAC の初期的な評価として, センサネットワーク用シミュレータ SENSE[2] 上で, MMMAC と既存の MAC プロトコルである B–MAC[3] および LMAC[4] を動作させ, 比較を行った. ここで, MMMAC として, FP の長さが異なる 3 つの場合を用意した. 評価項目は, 消費電力, マルチメディアパケットの遅延時間, およびマルチメディアパケットの通信成功率である. また, 評価に用いたパラメータを表 1 に示す.

本評価では, 監視アプリケーションを想定した環境を用いた. 評価に用いたノード 10 台のうち, 1 台を基地局, 2 台をカメラセンサを搭載したノード (カメラノード), および 7 台を振動センサを搭載したノード (振動ノード) を想定したアプリケーションプログラムを動作させた. カメラノードは, 基地局に画像データを 0.2〜4 秒周期で送信する. また, 振動ノードは, 最も近いカメラノードに, 1 秒周期で振動データを送信する.

図 4〜6 に評価結果を示す．評価結果より, MMMAC

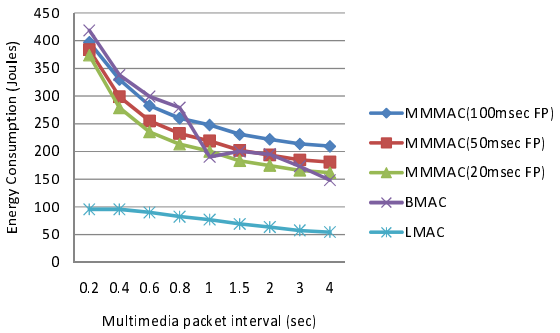


図 4　ノードの平均消費電力

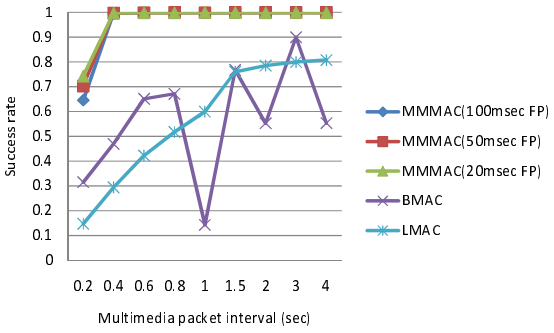


図 6　マルチメディアパケットの通信成功率

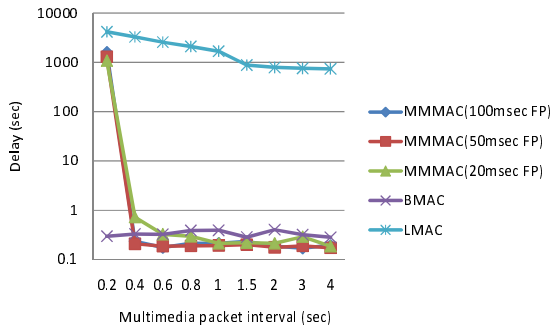


図 5　マルチメディアパケットの遅延時間

は, B–MAC と同程度の消費電力で, 低遅延および高信頼性通信が実現できたことがわかる. また, LMAC は, MMMAC および B–MAC と比較して, 約 1/3 から 1/4 の消費電力で動作する一方, 1000 倍以上の遅延が発生していることがわかる. これは, LMAC では, スロット毎に送信可能なノードが制限されるため, 帯域を効率よく利用できないからである. これらより, MMMAC は, B–MAC と LMAC と比較した場合, 省電力化および QoS を両立させる必要がある WMSNs で用いるのに適した MAC プロトコルであるといえる.

4　おわりに

本稿では, WMSNs における消費電力と QoS を考慮したハイブリッド型 MAC プロトコルの提案を行った. また, 提案する MAC プロトコルの初期的な評価を行った. 今後は, ACK の有無やスロットの時間などの各 MAC プロトコルの設定, およびノード数やノード密度などの実験環境を変動させた場合の MMMAC の評価を行う予定である. さらに, 実機への実装を行う予定である.

参考文献

[1] Ian F. Akyildiz, Tommaso Melodia, Kaushik R. Chowdhury, A survey on wireless multimedia sensor networks, Computer Networks, Vol.51, No.4, pp.921–960, March 2007.

[2] Gilbert Chen, Joel Branch, Michael Pflug, Lijuan Zhu, and Boleslaw Szymanski, SENSE: A Sensor Network Simulator, Advances in Pervasive Computing and Networking, 2004.

[3] J. Polastre, J. Hill, and D. Culler, Versatile low power media access for wireless sensor networks, in ACM SenSys '04, New York, USA, ACM Press, 2004, pp. 95–107.

[4] L. van Hoesel and P. Havinga, A lightweight medium access protocol (LMAC) for wireless sensor networks, Workshop on Networked Sensing Systems (INSS 2004), Tokyo, Japan, June 2004.

WMSNsにおける消費電力とQoSを考慮したハイブリッド型MACプロトコル

立命館大学大学院　理工学研究科 2年
大久保・横田研究室
濱千代　貴大

1

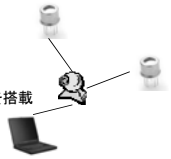
発表内容

- はじめに
- WMSNsで求められる要件
- 既存のMACプロトコルの問題点
- 提案するMultimedia MAC (MMMAC)
  - マルチメディアパケットの送信
  - 非マルチメディアパケットの送信
- 関連研究
- 初期的な評価
- おわりに

2

はじめに (1/2)

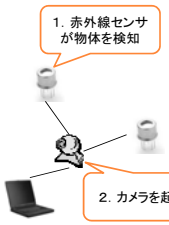
- 無線マルチメディアセンサネットワーク (WMSNs)
  - マルチメディアデータを扱う無線センサネットワーク
  - 多くのアプリケーションが利用可能
    - 監視, ヘルスケア, 老人補助など
- WMSNsのセンサノード
  - 2種類のノードが存在
    - 高消費電力のマルチメディアセンサを搭載
    - 低消費電力のスカラセンサを搭載
  - バッテリで動作



3

はじめに (2/2)

- WMSNsの特徴
  - 2種類のノードが役割分担によって協調し省電力化を実現
- WMSNsで発生するパケット
  - マルチメディアパケット
    - サイズが非常に大きい
    - 動画, 音声など
  - 非マルチメディアパケット
    - サイズが小さい
    - スカラデータ, 制御パケットなど



4

WMSNsアプリケーションの要求

- 稼働時間の長期化
  - センサノードがバッテリで動作しているため使用可能な電力が制限
- 優先度の高いデータをリアルタイムで取得
  - 緊急性の高いデータを対象
    - 動画データ
    - バイタルデータ

5

WMSNsにおける既存のMACプロトコルの課題

- 省電力化が十分でない
  - マルチメディアパケットの送信中に長期間オーバヒアリングが発生
- QoSを考慮していない
  - 高優先度通信に遅延が発生

6

課題の解決方針

- 省電力化
  - オーバヒアリングによる無駄な電力消費を削減
    - マルチメディアパケットが発生した場合，近傍のノードをスリープ
- QoS
  - 高優先度の通信が先行して送信することを保証
    - 定期的に通信を停止し、高優先度通信の存在を判断

立命館大学大学院 大久保・横田研究室

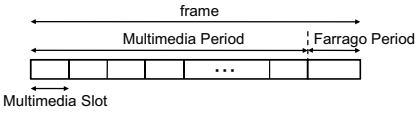
Multimedia MAC (MMMAC)

- パケットのサイズにより異なる通信方法を使用
  - マルチメディアパケットの送信：スロット占有方式
    - 優先権を得たノードがスロットを占有し，送受信を行わないノードはスリープ
    - 消費電力が低い一方，帯域使用率が低い
  - 非マルチメディアパケットの送信：CSMA方式
    - 帯域が空いていれば送信
    - 消費電力が高い一方，帯域使用率も高い
  - パケットによって送信方法を切り替え，両方式の弱みを克服
- バックオフによる優先制御
  - 高優先度通信に短いバックオフ時間を設定

立命館大学大学院 大久保・横田研究室

想定環境

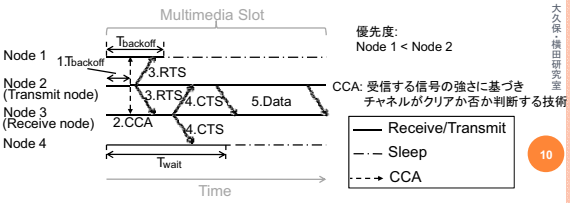
- 各ノードは時刻同期済み
- 時間をフレームを単位として分割
  - Multimedia Period(MP)でマルチメディアパケットを送信
    - MPは複数のMultimedia Slot(MS)から構成
  - Farrago Period(FP)で非マルチメディアパケットを送信



立命館大学大学院 大久保・横田研究室

マルチメディアパケットの送信

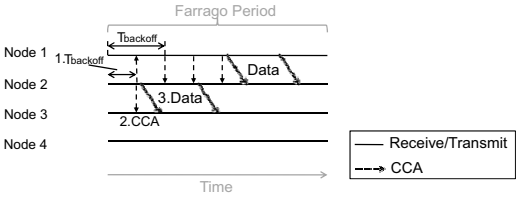
- スロット占有方式を用いて送信
  - 優先権を得たノードがスロットを占有、送受信を行わないノードはスリープ
  - 例：高優先度のノード2がノード3にデータを送信



立命館大学大学院 大久保・横田研究室

非マルチメディアパケットの送信

- 既存のCSMA手法を用いて送信
  - 帯域が空いていれば送信
  - 例：ノード1, 2がデータを送信



立命館大学大学院 大久保・横田研究室

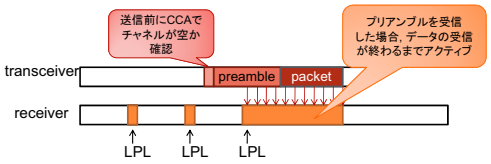
関連研究

- B-MAC
  - CSMAベース
- LMAC (Lightweight Medium Access Protocol)
  - TDMAベース
- [Saxena他]で提案された手法
  - WMSNs用MACプロトコル
  - CSMAベース

立命館大学大学院 大久保・横田研究室

B-MAC (1/2)

- 概要
  - CSMAベースのMACプロトコル
  - Low Power Listening (LPL) を用いて低消費電力を実現
    - 定期的にチャネルをサンプリング
    - 受信ノードのアイドルリスニングが減少

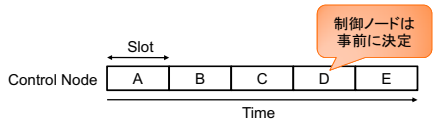


B-MAC (2/2)

- 提案手法との比較
  - 消費電力
    - マルチメディアパケットの消費電力が高い
      - LPLにより定期的にアイドルリスニングを行う可能性
  - 高優先度通信の遅延
    - 他のノードがマルチメディアパケットを送信していた場合，非常に長い遅延が発生
      - 優先制御を行っていないため
  - 帯域使用率
    - 提案手法と同様に高い
      - プリアンブルの送信期間のみ無駄

LMAC (1/2)

- 概要
  - TDMAベースのMACプロトコル
    - 時間をスロットで分割
    - 各スロットにおいて制御ノードのみがデータ送信可能
  - 通信を行わないノードの無線をオフにし省電力化
    - 制御ノードはスロットの開始時に受信ノード一覧を送信

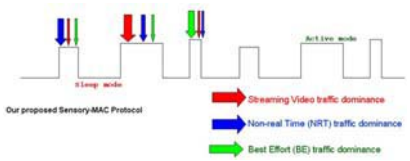


LMAC (2/2)

- 提案手法との比較
  - 消費電力
    - マルチメディアパケットの消費電力は同程度
      - 共に送受信を行うノード以外がスリープ
  - 高優先度通信の遅延
    - 長い遅延が発生
      - 優先制御を行っていない
      - 制御ノードしか送信できない
  - 帯域使用率
    - 他のノードに割り当てられたスロットを利用できないため低い

[Saxena他]で提案された手法 (1/2)

- 概要
  - WMSNs用に開発されたCSMAベースのMACプロトコル
  - デューティサイクルによりアクティブとスリープを切り替え
  - 通信時間が多い通信の種類に基づきデューティサイクルを決定



[Saxena他]より抜粋

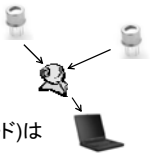
[Saxena他]で提案された手法 (2/2)

- 提案手法との比較
  - 消費電力
    - マルチメディアパケットの消費電力が高い
      - アクティブ状態において常にアイドルリスニングやオーバヒアリングの可能性
  - 高優先度通信の遅延
    - マルチメディアパケット：多い
      - デューティサイクルに基づき動作
    - 非マルチメディアパケット：少ない
      - アクティブ状態までの時間が提案手法のFPまでの時間より短い
  - 帯域使用率
    - デューティサイクルに基づくため低い

初期的な評価

- 評価方法
  - シミュレータ上でWMSNsアプリケーションを動作させ提案手法と他のMACプロトコルを比較
    - B-MAC[Polastre04], LMAC[Hoesel04]
- 評価環境
  - SENSE[Chen04]：センサネットワーク用シミュレータ
- 評価項目
  - 消費電力
  - マルチメディアパケットの遅延時間
  - マルチメディアパケットの通信成功率

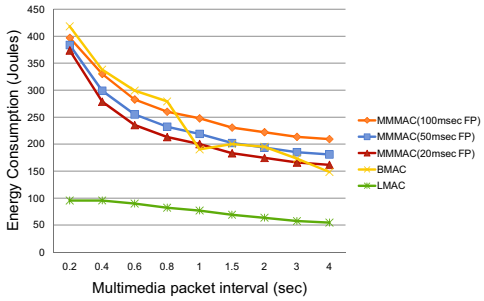
アプリケーションシナリオ



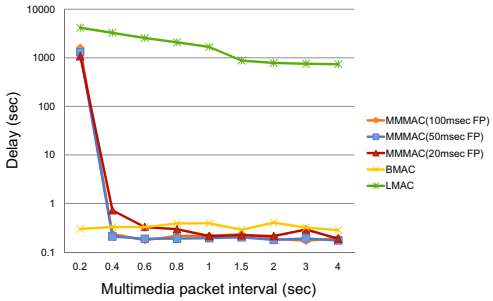
- 監視アプリケーションを想定
  - カメラノード(カメラセンサを搭載したノード)は基地局にマルチメディアパケットを送信
  - 振動ノード(振動センサを搭載したノード)は近隣のカメラノードに非マルチメディアパケットを送信

ノード数	基地局	1台
	カメラノード	2台
	振動ノード	7台
マルチメディアパケット	パケットサイズ	512～1024 bytes
	送信間隔	0.2～4.0 sec
非マルチメディアパケット	パケットサイズ	16 bytes
	送信間隔	1.0 sec

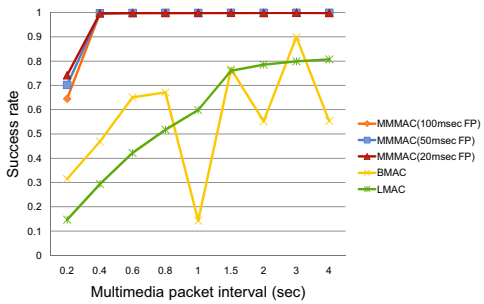
評価結果 (1/3) –消費電力–



評価結果 (2/3)
–マルチメディアパケットの遅延–



評価結果 (3/3)
–マルチメディアパケットの通信成功率–



考察

- WMSNsにおいてMMMACが最適
  - B-MACと同程度の消費電力で低遅延・高通信成功率
  - LMACより極めて低遅延
- 想定より消費電力が大きい
  - FPで長期間のアイドルリスニングが発生
  - バックオフ中の電力消費

おわりに

- 発表内容
  - 無線マルチメディアセンサネットワーク(WMSNs)
    - WMSNsアプリケーションの要求
  - Multimedia MAC(MMMAC)
    - マルチメディアパケットの送信: スロット占有方式
    - 非マルチメディアパケットの送信: CSMA方式
  - 初期的な評価
  - 関連研究
- 今後の予定
  - 詳細な評価
    - MACプロトコルの設定や評価環境を変動

立命館大学大学院 大久保・横田研究室

センサネットワークにおけるメタデータを用いた統一的問い合わせ手法

藤原 秋司 †† 横田 裕介 † 大久保 英嗣 †
† 立命館大学情報理工学部 †† 立命館大学大学院理工学研究科

1 はじめに

現在, 我々は MGS(Multi-device control Gateway Service for sensor networks) と呼ぶデバイス制御フレームワークの開発を行っている. MGS では, 多種の無線センサネットワーク (以下, WSN と記す) とデバイスを組み合わせた環境で, WSN やデバイスの動的な追加や削除に対応する. また, ユーザが WSN の違いや構成の変化を意識せずに利用することを可能とする. このような多くの WSN およびデバイスから構成されるゲートウェイサービスでは, システムの大規模化が考えられ, システムを利用するユーザが複数存在する, マルチユーザ環境を想定する必要がある. マルチユーザ環境では, 1 つの WSN に対し複数の問い合わせが発行されることが考えられる. しかし, 多くのセンサネットワークシステムでは, 同時に複数の問い合わせを処理することができない. さらに, 複数の問い合わせに対応しているセンサネットワークシステムでも, 取得したいセンシングデータの属性や, サンプリング周期の重複から無駄なセンシングが発生する. 従って, ネットワークトラフィックの増加や, WSN の寿命が短くなるという問題がある.

本研究では, ユーザの要求と WSN の制限を同時に満たすよう, 事前に問い合わせを最適化し, この問題を解決するための手法を提案する. また, 無駄なセンシングを最小限に止めることで, 効率的な WSN の利用を実現する.

以下, 2 章で関連研究について述べ, 3 章で提案手法について述べる. また, 4 章で最適化機構の設計と実装について述べ, 5 章で提案手法の評価について述べる. 最後に 5 章で本稿のまとめを行い, 今後の予定について述べる.

2 関連研究

2.1 TGCS[2]

問い合わせの統合を行う際に, 最適なサンプリング周期を求めるアルゴリズム, TGCS を提案している. TGCS は最大公約数を用いるが, 誤差を許容することで最大公約数より大きなサンプリング周期を求めることができる. 従って, 最大公約数をサンプリング周期として用いる場合より, センサノードの負荷を減らすことが可能となる. しかし, TGCS で用いられている許容誤差は WSN 単位で設定されており, ユーザにより許容できる誤差が異なるマルチユーザ環境には適切でないと考えられる.

2.2 Two–tier multiple query optimization[3]

センサノードレベルとゲートウェイレベルで 2 段階の最適化を行う手法を提案している. MGS と同レベルであるゲートウェイレベルでは, *Benefit* と呼ばれる基準によるコストモデルを用いた問い合わせの最適化を行っている. 各問い合わせのコストを求め, 最適化の有用性を求めることで, 最適化を行うかどうかという判定を行っている. なお, サンプリング周期には, 各サンプリング周期の最大公約数を用いる. コストモデルを用いることで, 問い合わせを統合する際に発生するコストを抑えることができる. しかし, コストによっては問い合わせを統合しない場合があるため, WSN に対して複数の問い合わせが発行される. 従って, 適用できる WSN が限られるという問題がある.

3 問い合わせの最適化手法

本章では, ユーザが発行する問い合わせ (以下, ユーザクエリと記す) から実際に WSN に発行される問い合わせ (以下, ネットワーククエリと記す) を生成するまでの手順を示す. また, 最適化に用いる WSN 情報を記したメタデータの構造, ユーザクエリの表記規則について述べる.

3.1 概要

本提案手法では, ユーザクエリに SQL を拡張した表記法を用い, サンプリング周期に許容誤差を持たせる. また, 異種 WSN 混在環境下では, 1 つの場所に複数の WSN が混在していることも考えられる. 従って, ユーザクエリ内で要求するデータを取得可能な WSN を複数指定することで, その中から最も効率のよい WSN を自動的に選択し, 問い合わせの発行を行う. また, サンプリング周期に一定以上の差があるような場合, 例えば, 1 分周期で温度, 60 分周期で照度と指定するような問い合わせの統合では, 1 分周期で温度, 照度を取得するという問い合わせに統合される. この場合, 統合することで 59 回分の照度データが無駄に回収される. このような問い合わせの統合には, 必要な時のみ動的に問い合わせを変更することで対応する. 本提案手法を用いることで, 無駄の少ない問い合わせの最適化が可能となる.

3.2 メタデータ

異種 WSN の管理と問い合わせの最適化を行う際にメタデータを用いる. メタデータでは, WSN 情報として,

センサノード用 OS, センサノード用ソフトウェア, センサノード数, 設置位置, ノード単位での問い合わせの可否, 最小サンプリング周期, サンプリング周期変更幅の管理を行う. また, センサノード情報として, センサノードの種類, センサ基板の種類, 取得データ, ノード ID の管理を行う.

3.3 ユーザクエリ

ユーザクエリには, 独自に拡張した SQL を用いる. サンプリング周期の許容誤差は “±” で表し, 分, 秒での表記に対応する. また, 複数のセンサネットワークの中から 1 つのセンサネットワークを選択してもよいという問い合わせは, “or” で表記し, 最初に記載した WSN に優先的に割り振るものとする. ユーザクエリ表記例は, “SELECT 温度, 照度 FROM SN_A or SN_B SAMPLE PERIOD 15 秒 ±5 秒” のようになる.

3.4 問い合わせの統合

3.4.1 サンプリング周期の共通部分における統合

各ユーザクエリのサンプリング周期の共通部分を用いて問い合わせの統合を行う. 例えば, 10 秒 ±5 秒と 15 秒 ±5 秒というユーザクエリがある場合, 10 秒〜15 秒という共通部分を採用した問い合わせに統合される. また, 取得項目, 対象とする WSN ごとにユーザクエリの分割, 整理を行う.

3.4.2 コストモデルと最大公約数 (GCD) を用いた統合

問い合わせの統合を行う際に, 無駄なセンシングデータの取得を最小限に抑えるため, 問い合わせのコストを管理する. コストが最小となるように問い合わせの統合を行うことで, 最も無駄の少ない問い合わせの発行先を決定する. 本提案手法で用いるコストを以下に示す. なお, 各 WSN ごとに求める各ユーザクエリのサンプリング周期の最大公約数を $GCD(s)$ とする.

$$Cost_{total} = Cost_{WSN_A} + Cost_{WSN_B} + \cdots \quad (1)$$

$$Cost_{WSN} = \frac{\alpha_{total}}{GCD(s)} \quad (2)$$

$$\alpha_{total} = \begin{cases} \sum_{i=1}^{n} \alpha_i & \alpha_i \neq 1 \text{ のとき} \\ 1 & \alpha_i = 1 \text{ のとき} \end{cases} \quad (3)$$

α は, 統合された問い合わせによって得られた結果を α 回に 1 回ユーザに返すことを表す値である. また, $\alpha_i (i = 1, 2, 3\cdots)$ は, 各ユーザクエリの α を表す. 例えば, ユーザクエリのサンプリング周期が 15 秒と 20 秒で発行されている場合, $GCD(s)$ からネットワーククエリは 5 秒周期となる. 従って, $\alpha_1 = 3, \alpha_2 = 4, Cost_{WSN} = \frac{7}{5}$ と求まる. また, $GCD(s) =$ ユーザクエリのサンプリング周期 となる場合は, 無駄なセンシングデータの回収が行われていない. 従って, $\alpha_{total} = 1$ とする.

問い合わせの対象となっている, 各 WSN のコストの合計である $Cost_{total}$ の値を比較することで, どの WSN に対して問い合わせを統合するのが最も効率的であるのかを判断する. また, ノード単位での問い合わせの発行が可能な場合, 現段階で採用されている問い合わせがノード単位で発行される.

3.4.3 最大公約数による強制的な統合

ノード単位での問い合わせが不可能な場合, 各 WSN ごとに問い合わせを 1 つに統合する必要がある. また, サンプリング周期が短い場合, 動的にネットワーククエリを変更すれば WSN への負荷が大きくなるため, 最大公約数による強制的な統合を行う.

3.4.4 動的なクエリ変更による統合

サンプリング周期が一定値より長い場合, 最大公約数による強制的な統合を用いると無駄なセンシングデータの回収量が多くなる. 従って, 動的に問い合わせを変更することで, 問い合わせの統合を実現する.

3.5 問い合わせ結果の分配

ネットワーククエリによって, 得られた問い合わせ結果から, 各ユーザクエリに応じた問い合わせ結果を抽出し, 各ユーザに返す必要がある. 2.4.2 項同様 α を用い, α 回に 1 回センシングデータを回収することで, 各ユーザは必要なセンシングデータのみを取得することが可能となる.

4 問い合わせの最適化機構の設計と実装

本章では, プロトタイプシステムの構成と各モジュールの役割について述べる. 本システムでは, 分散システムである Jini を用いることで, 各モジュールをサービスとして運用し, 異種 WSN の混在を可能としている. 以下, 各モジュールの動作について述べる (図 1 参照).

Client センシングデータを要求するユーザが利用する. ServiceProvider(以下, SP と記す) によって提供されているメタデータからユーザクエリを生成し, Gateway のデータベースにユーザクエリの登録を行う. また, Gateway に格納されるセンシングデータを回収し, センシングデータの単位変換を行う. 単位変換サービスは, 各 SP によって提供される.

Gateway Jini の LookupService と WSN への問い合わせ, センシング結果などを格納するデータベースから構成される. また, 問い合わせの最適化も Gateway で行う. データベースに登録されたユーザクエリを一

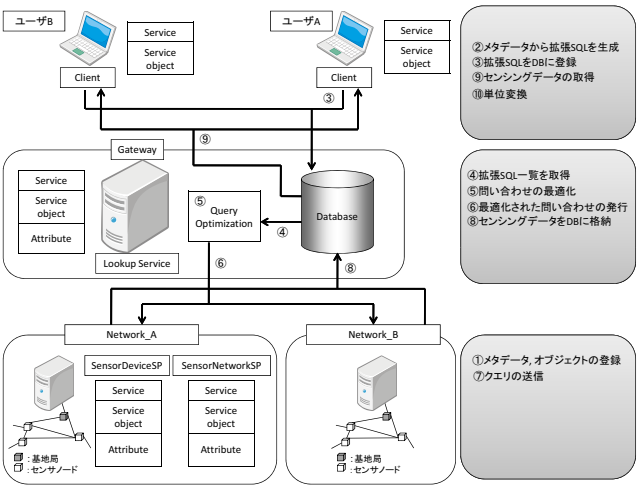


図 1　プロトタイプシステム構成

定周期ごとに読み取り, 各 SP によって提供されるメタデータを用いることで問い合わせの最適化を自動的に行う. また, 問い合わせの発行には, 各 SP から提供される, クエリ発行サービスを用いる.

ServiceProvider　WSN へのクエリ発行サービスである SensorNetworkSP と, 各センサ基板ごとに異なる単位の変換サービスである SensorDeviceSP から構成される. 各サービスを LookupService に登録することで, 他のモジュールからのサービス利用を実現する.

5　評価

本章では, 提案手法の有用性を示すために行った初期的な評価について述べる. 今回, 基本的な既存手法である, 各ユーザクエリにおけるサンプリング周期の最大公約数を用いて統合を行う手法を用いた場合, 最適化を行わない場合, 提案手法を用いた場合の 3 パターンにおける, 1 時間で WSN 内を流れるメッセージ数の比較を行った. また, 評価には TinyOS 用のシミュレータである TOSSIM[4] を用いた. 評価に用いる WSN は, 温度, 照度, 湿度を取得可能なノード 64 個から構成され, WSN が 2 つ (WSN_A, WSN_B) 混在する構成を想定する. なお, WSN に対して発行されるユーザクエリは, 取得項目, 対象 WSN, サンプリング周期, 許容誤差をランダムに生成したものを用い, 最適化の周期は 5 分とした. 単一センサネットワークに対する評価結果を表 1 に示し, 複数センサネットワークに対する評価結果を表 2 に示す.

表 1, 表 2 の結果から Query Message, Sensing Message ともに, メッセージ数の削減を実現できたことがわかる. よって既存手法では, 各ユーザクエリの要求を満たすのみだったのに対し, 提案手法では, 各ユーザクエリの要求を満たすだけでなく, ネットワーク内を流れるメッセージ数の削減も行うことができたといえる.

表 1　単一センサネットワークにおけるメッセージ数

	Query Message	Sensing Message
最適化なし	56561	2089901
既存手法	43572	2382624
提案手法	33663	1412940

表 2　複数センサネットワークにおけるメッセージ数

	Query Message	Sensing Message
最適化なし	56561	2089901
既存手法	59713	2378424
提案手法	44147	1487037

6　おわりに

本稿では, 異種 WSN 混在環境下における問い合わせの最適化機構について述べた. また, プロトタイプシステムの設計と評価について述べた. 今後は, ユーザクエリの許容誤差や最適化周期の値を変動させ, 評価を行う予定である. また, これらの評価が終わり次第, 実際のセンサネットワークシステムを用いた環境への実装を行う予定である.

参考文献

[1] Jini Network Tecnology, `http://incubator.apache.org/river/RIVER/index.html`.

[2] Müller, R. and Alonso, G., “Shared queries in sensor networks for multi-user support”, MASS, 2006.

[3] Xiang, S. and Lim, H.B. and Tan, K.L. and Zhou, Y., “Two-tier multiple query optimization for sensor networks”, Proceedings of the 27th International Conference on Distributed Computing Systems, 2007.

[4] Levis, P. and Lee, N. and Welsh, M. and Culler, D., “TOSSIM: Accurate and scalable simulation of entire TinyOS applications”, In Proc. of ACM SenSys, 2003.

異種センサネットワーク混在環境下における マルチユーザを考慮した問い合わせの 最適化手法

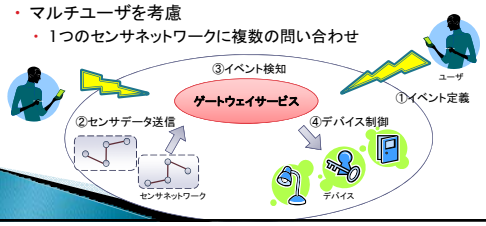
立命館大学大学院 理工学研究科
大久保・横田研究室
藤原 秋司

目次

- はじめに
  - 研究背景
- 関連研究
  - TGCS
  - Two-tier multiple query optimization
- 問い合わせの最適化手法
  - 概要
  - 動作手順
  - 設計
  - 評価
- おわりに

はじめに(1)

- MGS(Multi-device control Gateway Service for sensor networks)
  - デバイス制御フレームワーク
    - 複数のセンサ(問い合わせ処理に対応), デバイス
    - センサデータをトリガーとしてデバイスを制御
  - システムの大規模化
    - マルチユーザを考慮
      - 1つのセンサネットワークに複数の問い合わせ



はじめに(2)

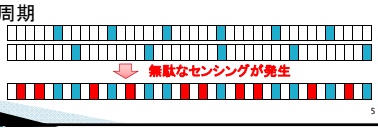
- 既存のセンサネットワークシステムの問題
  - 複数問い合わせに対応していないものが多い
  - 問い合わせ内容の重複による無駄なセンシング

- 事前に問い合わせの最適化
  - 複数の問い合わせを統合

関連研究(1)

- 基本的な統合方法
  - 取得項目
    - 各取得項目の和集合
  - サンプリング周期
    - 各サンプリング周期の最大公約数
- 統合例
  - 温度を15秒周期, 照度を20秒周期

  - 温度, 照度を5秒周期

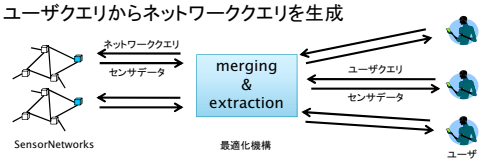


関連研究(2)

- TGCS
  - 特徴
    - サンプリング周期に一定の誤差を認めた最大公約数
  - 問題点
    - 誤差を前もってセンサネットワークごとに指定
- Two-tier multiple query optimization
  - 特徴
    - コストモデルによる問い合わせの統合を行うか否かの判定
  - 問題点
    - 1つのセンサネットワーク内に複数の問い合わせが発生

問い合わせの最適化手法の概要

- 問い合わせの最適化機構
  - ユーザクエリからネットワーククエリを生成



  - ユーザクエリ
    - 記述方法を定めた問い合わせ文(SQL文)
    - 問い合わせ単位でのサンプリング周期の許容誤差: “±”
    - 複数センサネットワークからの選択: “or”
  - 無駄なセンシング(コストの増大)を防止
    - 動的なクエリ変更

問い合わせの最適化手順

1. サンプリング周期の共通部分における統合
   - 15秒±5秒, 20秒±5秒⇒15秒～20秒
2. コストモデルと周期の最大公約数を用いた統合
   - 複数センサネットワークからの選択⇒コストの比較
   - 最大公約数による無駄の削減⇒統合
3. 最大公約数に基づく強制的な統合
   - サンプリング周期の短いユーザクエリ
4. 動的なクエリ変更による統合
   - サンプリング周期の長いユーザクエリ
5. 問い合わせ結果の分配

問い合わせの最適化手順

1. サンプリング周期の共通部分における統合
   - 15秒±5秒, 20秒±5秒⇒15秒～20秒
2. コストモデルと周期の最大公約数を用いた統合
   - 複数センサネットワークからの選択⇒コストの比較
   - 最大公約数による無駄の削減⇒統合
3. 最大公約数に基づく強制的な統合
   - サンプリング周期の短いユーザクエリ
4. 動的なクエリ変更による統合
   - サンプリング周期の長いユーザクエリ
5. 問い合わせ結果の分配

問い合わせの最適化手順

1. サンプリング周期の共通部分における統合
   - 15秒±5秒, 20秒±5秒⇒15秒～20秒
2. コストモデルと周期の最大公約数を用いた統合
   - 複数センサネットワークからの選択⇒コストの比較
   - 最大公約数による無駄の削減
3. 最大公約数に基づく強制的な統合
   - サンプリング周期の短いユーザクエリ
4. 動的なクエリ変更による統合
   - サンプリング周期の長いユーザクエリ
5. 問い合わせ結果の分配

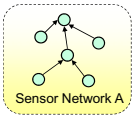
クエリ統合によるコストの比較

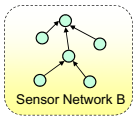
①SELECT 温度 FROM SN_A 20秒周期
②SELECT 温度 FROM SN_B 60秒周期
SN_AとSN_Bからデータが欲しい

③SELECT 温度 FROM SN_A or SN_B 15秒周期
SN_AもしくはSN_Bからデータが欲しい

ネットワーククエリ
SELECT 温度 FROM 20秒周期

ネットワーククエリ
SELECT 温度 FROM 60秒周期





- ③をSN_Aに統合した場合のネットワーククエリ
  - SELECT 温度 FROM 5秒周期
- ③をSN_Bに統合した場合のネットワーククエリ
  - SELECT 温度 FROM 15秒周期

SN_Bと統合したほうが無駄がない

コストモデル

- コスト
  - 無駄なセンシングが多い⇒コストが大きい
  - GCD(s):各サンプリング周期の最大公約数

$$Cost_{total} = Cost_{WSN_A} + Cost_{WSN_B} + \cdots \quad (1)$$

$$Cost_{WSN} = \frac{\alpha_{total}}{GCD(s)} \quad (2)$$

$$\alpha_{total} = \begin{cases} \sum_{i=1}^{n} \alpha_i & \alpha_i \neq 1 \text{ のとき} \\ 1 & \alpha_i = 1 \text{ のとき} \end{cases} \quad (3)$$

- α回に一回センシング結果を取得
  - 15秒, 20秒周期のユーザクエリ, 5秒周期のネットワーククエリ
    - $\alpha_1=3$, $\alpha_2=4$
  - 15秒, 60秒周期のユーザクエリ, 15秒周期のネットワーククエリ
    - $\alpha_1=1$, $\alpha_2=4$

コストモデル

- コスト
  - 無駄なセンシングが多い⇒コストが大きい
  - GCD(s):各サンプリング周期の最大公約数

$$Cost_{total} = Cost_{SN_A} + Cost_{SN_B} = \frac{7}{5} + \frac{1}{60} = \frac{85}{60}$$

$$Cost_{total} = Cost_{SN_A} + Cost_{SN_B} = \frac{1}{20} + \frac{1}{15} = \frac{7}{60}$$

- α回に一回センシング結果を取得
  - 15秒, 20秒周期のユーザクエリ, 5秒周期のネットワーククエリ
    - $\alpha_1=3$, $\alpha_2=4$
  - 15秒, 60秒周期のユーザクエリ, 15秒周期のネットワーククエリ
    - $\alpha_1=1$, $\alpha_2=4$

13

問い合わせの最適化手順

1. サンプリング周期の共通部分における統合
   - 15秒±5秒, 20秒±5秒⇒15秒～20秒
2. コストモデルと周期の最大公約数を用いた統合
   - 複数センサネットワークからの選択⇒コストの比較
   - 最大公約数による無駄の削減⇒統合
3. 最大公約数に基づく強制的な統合
   - サンプリング周期の短いユーザクエリ
4. 動的なクエリ変更による統合
   - サンプリング周期の長いユーザクエリ
5. 問い合わせ結果の分配

14

問い合わせの最適化手順

1. サンプリング周期の共通部分における統合
   - 15秒±5秒, 20秒±5秒⇒15秒～20秒
2. コストモデルと周期の最大公約数を用いた統合
   - 複数センサネットワークからの選択⇒コストの比較
   - 最大公約数による無駄の削減⇒統合
3. 最大公約数に基づく強制的な統合
   - サンプリング周期の短いユーザクエリ
4. 動的なクエリ変更による統合
   - サンプリング周期の長いユーザクエリ
5. 問い合わせ結果の分配

15

問い合わせ結果の分配

- 回収したセンシングデータに番号を付加
- 統合後の問い合わせとユーザクエリからαを算出
- αの倍数のセンシングデータをClientが回収
  - $\alpha_1 = 2, \alpha_2 = 3$

番号	温度
1	14.5
2	14.7
3	14.8
4	15.0
5	15.0
6	15.1
7	14.8
:	:

番号	温度
2	14.7
4	15.0
6	15.1
:	:

番号	温度
3	14.8
6	15.1
:	:

16

問い合わせの最適化機構の設計

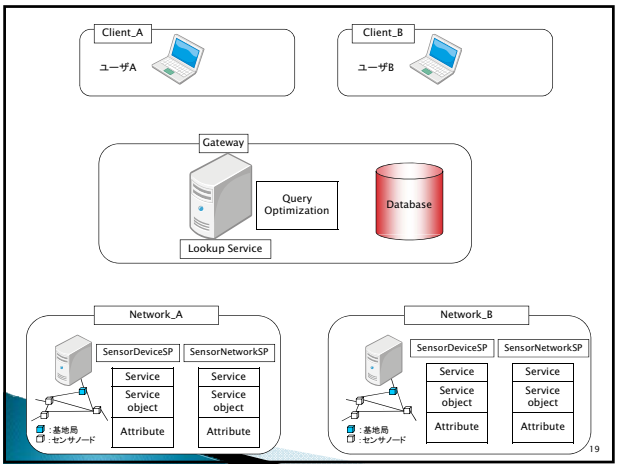
- 異種センサネットワークの混在
  - 分散システムJiniの利用
    - 各モジュールをサービスとして利用
      - Client, Gateway, ServiceProvider
  - モジュール構成
    - Client
      - ユーザが問い合わせの生成に利用
    - Gateway
      - 最適化機構, Database, Lookup Service
    - ServiceProvider
      - メタデータによる各センサネットワークの管理

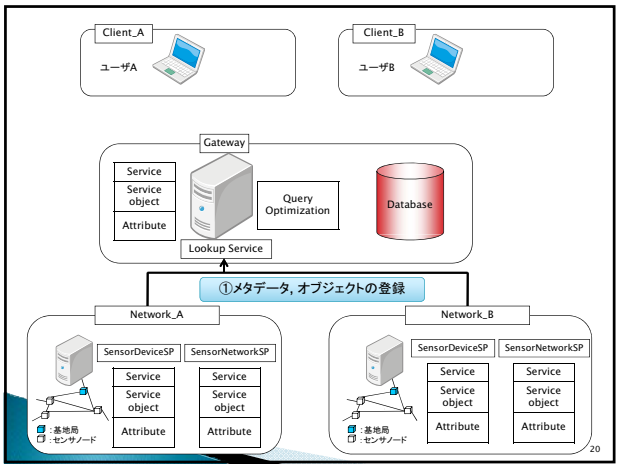
17

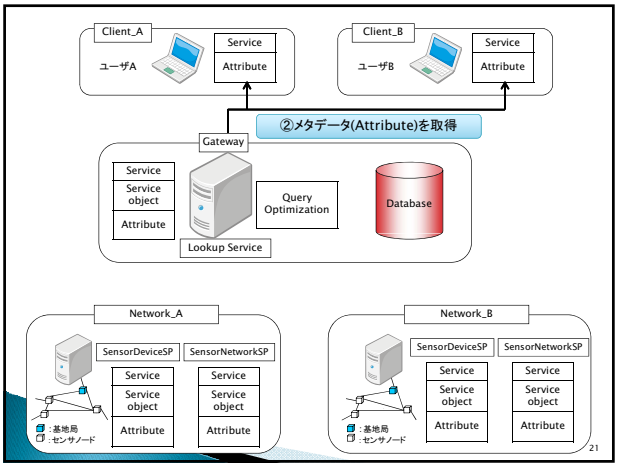
メタデータによるセンサネットワーク構成の管理

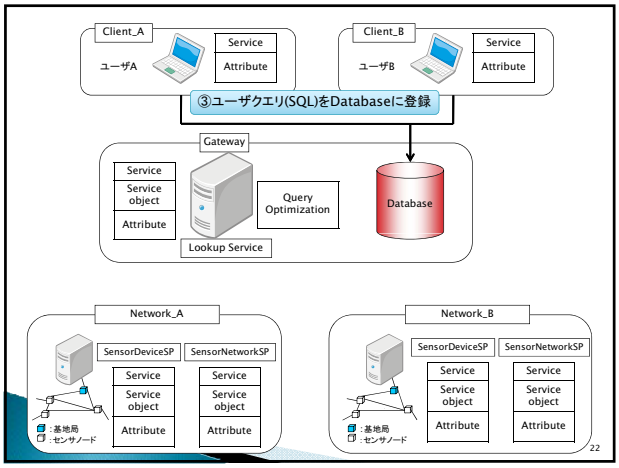
- センサネットワーク情報
  - センサノード用OS
  - センサノード用SW
  - センサノード数
  - 設置位置
  - ノード単位での問い合わせ
  - 最小サンプリング周期
  - サンプリング周期変更幅
- センサノード情報
  - センサノードの種類
  - センサ基板の種類
  - 取得可能データ
  - ノードID

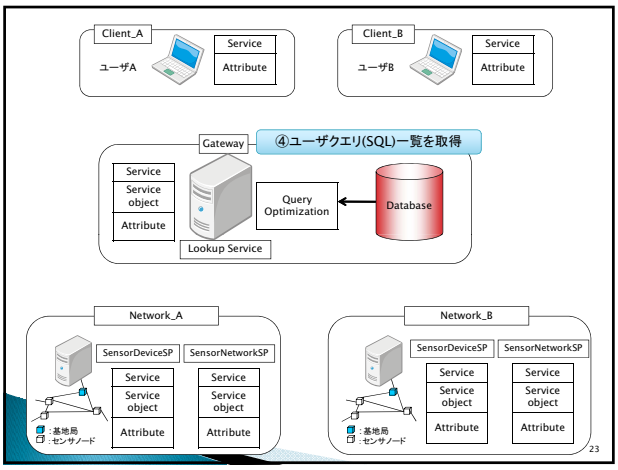
18

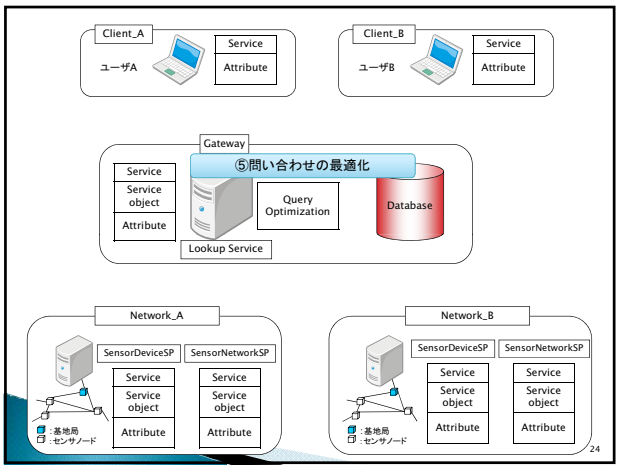

Client_A
ユーザA
Client_B
ユーザB
Gateway
Query Optimization
Database
Lookup Service
Network_A
Network_B
SensorDeviceSP
SensorNetworkSP
Service
Service object
Attribute
：基地局
：センサノード
19

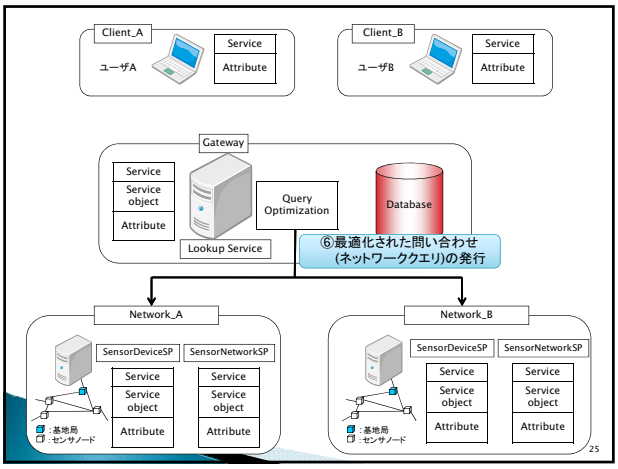

Client_A
ユーザA
Client_B
ユーザB
Gateway
Service
Service object
Attribute
Query Optimization
Database
Lookup Service
①メタデータ, オブジェクトの登録
Network_A
Network_B
SensorDeviceSP
SensorNetworkSP
：基地局
：センサノード
20

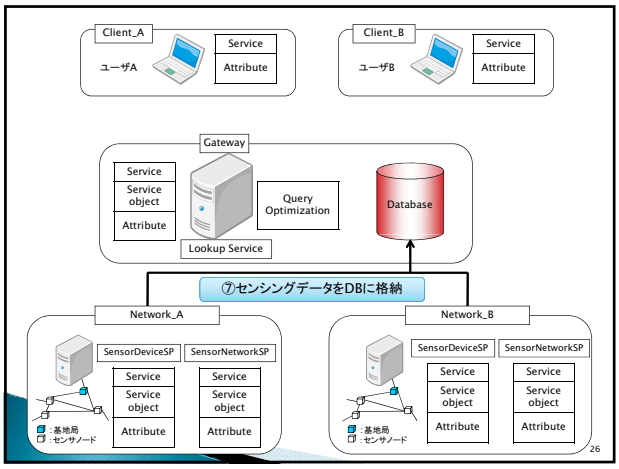

Client_A
ユーザA
Service
Attribute
Client_B
ユーザB
②メタデータ(Attribute)を取得
Gateway
Query Optimization
Database
Lookup Service
Network_A
Network_B
SensorDeviceSP
SensorNetworkSP
：基地局
：センサノード
21

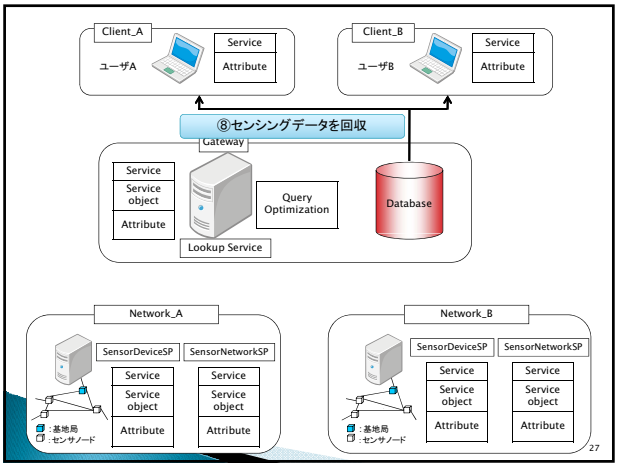

Client_A
ユーザA
Service
Attribute
Client_B
ユーザB
③ユーザクエリ(SQL)をDatabaseに登録
Gateway
Query Optimization
Database
Lookup Service
Network_A
Network_B
SensorDeviceSP
SensorNetworkSP
：基地局
：センサノード
22

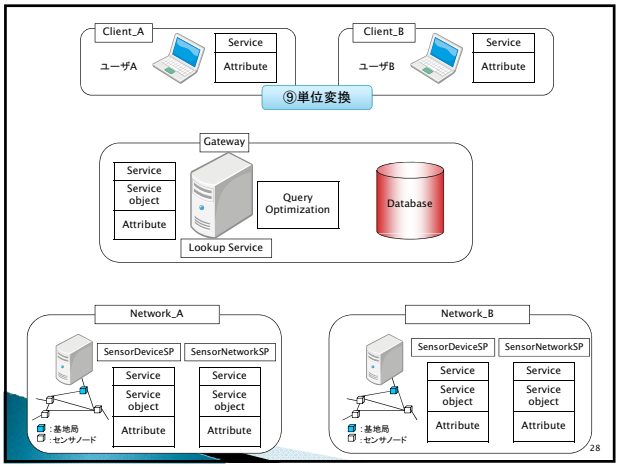

Client_A
ユーザA
Service
Attribute
Client_B
ユーザB
Gateway
④ユーザクエリ(SQL)一覧を取得
Query Optimization
Database
Lookup Service
Network_A
Network_B
SensorDeviceSP
SensorNetworkSP
：基地局
：センサノード
23


Client_A
ユーザA
Service
Attribute
Client_B
ユーザB
Gateway
⑤問い合わせの最適化
Query Optimization
Database
Lookup Service
Network_A
Network_B
SensorDeviceSP
SensorNetworkSP
：基地局
：センサノード
24









評価

- 各手法におけるネットワークメッセージ数の比較
  - 単一，複数センサネットワーク
    - 提案手法
      - 許容誤差
      - 最大公約数(サンプリング周期)
      - 複数センサネットワークからの選択
    - 既存手法
      - 最大公約数(サンプリング周期)
    - 最適化を行わない
      - 各ユーザクエリを単独に処理
- TinyOS用シミュレータTOSSIMを利用
  - センサノード: 64個，ランダム配置，3ホップ以下
  - 取得可能項目: 温度，照度，湿度
  - 2種類のセンサネットワークが混在: WSN_A, WSN_B

29

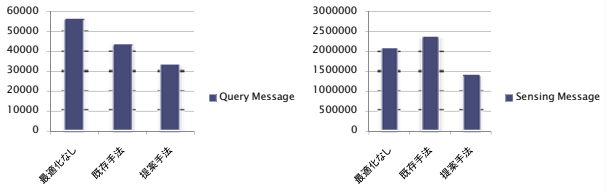
各種設定

- 評価時間: 1時間
- 最適化周期: 5分
- ユーザクエリ
  - 常に6つのユーザクエリが存在
  - 許容誤差: サンプリング周期の0%～50%
  - 取得項目，取得対象，サンプリング周期: ランダム

30

評価結果(1)

- 単一センサネットワーク

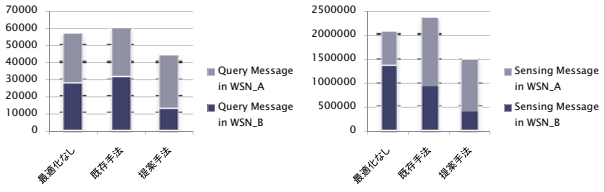


  - クエリメッセージ，センシングメッセージともに減少

31

評価結果(2)

- 複数センサネットワーク



  - クエリメッセージ，センシングメッセージともに減少

評価結果1，2よりユーザの要求を満たし，メッセージの削減を実現

32

おわりに

- 研究背景
- 関連研究
- 問い合わせの最適化手法
  - 動作手順
  - 設計
  - 評価
- 今後の予定
  - 最適化周期，許容誤差を変動させた評価
  - 実環境への実装

33

移動センシング環境におけるストリームデータ分割配送手法

村山 知弥 †† 横田 裕介 † 大久保 英嗣 †
† 立命館大学情報理工学部 †† 立命館大学大学院理工学研究科

1 はじめに

我々は，人と共にセンサノードが移動する，移動センシング環境に関する研究を行っている．人が移動しながら，センサデバイスを持ち歩くことで，バイタルデータなどの高精度な観測が可能となる．このような観測データは，高頻度で取得されるストリームデータであり，継続的に届けられるデータに対する高速な処理が必要とされている [1]．このため，これまでのストリームデータ処理システムでは，観測データの保持よりも応答時間の短縮が重視されてきた．しかし，移動センシング環境においてこれらの仕組みを用いた場合，ノードがネットワークから離脱した際，観測データが基地局に届く前に失われるという問題がある．

本研究では，周囲のセンサノードにデータを分配することでデータ損失を防ぐ，ストリームデータ分割配送手法を提案する．本手法により，ネットワーク離脱時および輻輳発生時の観測データを保持し，より多くのデータを基地局に届けることが可能となる．以下，本稿では，2 章で研究背景について述べた後，3 章で提案手法について，4 章で実装と評価について述べる．

2 研究背景

本研究では，被災地に Zigbee のような無線による低速回線ネットワークが構築され，それらのネットワークがインターネットのような高速回線ネットワークで接続されている環境を想定している．低速なネットワークから集められたデータは，高速なネットワーク上で処理され，特定のデータ利用者に提供される．このような環境では，高速なネットワークで負荷の大きな処理を行うことによって，システム全体の処理の効率化が可能となる．そのため，サービスの質を向上させるためには，できるだけ多くのデータを低速なネットワークから高速なネットワークに届けることが重要となる．

センサノードは，健康状態を観測するためにすべての人が持ち歩いており，充電が可能な端末が利用されるものとする．本研究では，典型的なシナリオとして「体温が一定値を超えた場合，その後の観測データをリアルタイムに基地局へ送信する」といったクエリが実行されることを想定している．ここでの観測対象者は，被災地における避難所などのように，周囲にノードが比較的多く

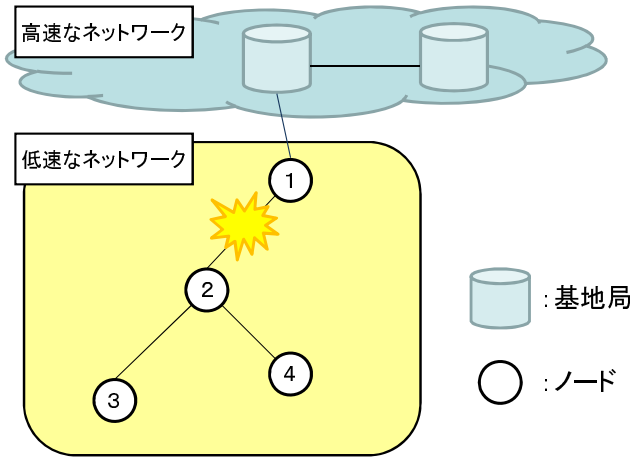


図 1 ネットワークの分断

存在する場所を移動しながらサービスを受けるものとする．

観測データとしては，一定時間，連続的に発生するストリームデータを想定している．こういったストリームデータを処理するシステムとして，DSMS（Data Stream Management System）[2, 3] があげられる．しかし，既存の DSMS では，高速なネットワークにおけるストリームデータ処理の応答性に重点を置いているため，低速なネットワークでのデータ損失による信頼性の低下を考慮していない．そのため，ノードの移動によりネットワークの状況が動的に変化する移動センシング環境においては，低速なネットワーク内で多くのデータが失われ，高速なネットワークにおいてサービスの質を維持できない可能性が高い．

例えば，図 1 のような環境においてノード 1 とノード 2 の通信ができなくなった場合，その後ノード 2 で観測したデータは基地局に届くことなく失われてしまう．既存の DSMS では，こういったデータ損失は考慮されていないが，これらのデータをノード 3 やノード 4 のように通信が可能なノードに転送することで，ノード 3 あるいはノード 4 が移動して基地局と通信可能となった場合に基地局に届けることが可能となる．ノード 2 で観測されたデータは，ノード 2 で保持しておくことも可能であるが，バイタルデータのように緊急性の高いデータをで

きるだけ早く届けるためには，周囲のノードを利用した分割配送が有効であると考えられる．このように周囲のノードのストレージを用いて分割配送することで，基地局へより多くのデータを到達させることが本研究の目的である．

3　移動センシング環境における ストリームデータ分割配送手法

本研究では，2 章で述べた問題点を解決するために，移動センシング環境におけるストリームデータ分割配送手法を提案する．本手法では，センサノードが基地局と通信できない状態であると判断された場合，ノード上のストレージがデータ保持のために利用される．本手法により，ノードの移動やネットワークの輻輳が発生する通信が不安定な環境において，データが失われることのない信頼性の高いネットワークが構築可能となる．

以下，3.1 節で本手法の概要について，3.2 節で分割配送処理の詳細について述べる．

3.1　ストリームデータ分割配送手法の概要

センサノードが通信不可能な範囲に移動した場合，あるいは通信速度を超える頻度でストリームデータが発生して輻輳が生じた場合，基地局と通信することができない分断されたネットワークが形成される．このようにノードがネットワークから離脱した状態となった場合，周囲のノードのストレージを利用した分割配送を行う．

各ノードが行うストリームデータ処理について，具体的な手順を以下に示す．

1. 観測処理

   センサからの入力値を読み，基地局に届けるべきデータが発生した場合，自身のストレージに格納してから基地局への送信を試みる．

2. 離脱検知処理

   送信先から受信確認（ACK）が返った場合は，送信したデータを自身のストレージから消去する．ACK が返らない場合は，ネットワークからの離脱と判断し，データを保持する．

3. 分割配送処理

   自身の保持するデータが一定量を超えた場合，他ノードのストレージの状況を調べる．他ノードのストレージ容量に空きがあるならば，自身の観測データを分割配送する．

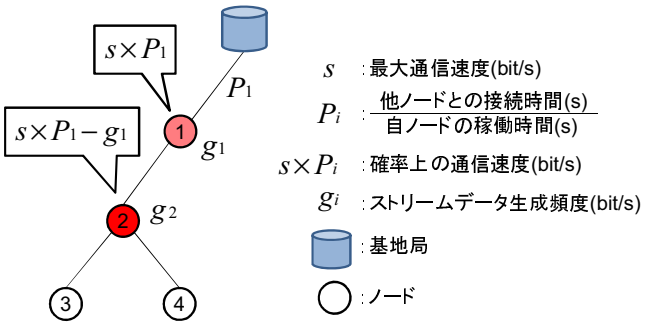


図 2　分割配送処理の開始

これらの処理を実現することで，ネットワーク離脱時に発生したストリームデータを，周囲のノードに分割して転送することが可能となる．

3.2　分割配送処理の詳細

まず，分割配送処理を開始するタイミングについて述べる．図 2 のようなネットワークの場合，ノード 1 が単位時間当たりに生成するデータ量を除いた残りが，ノード 2 に許可されるデータ送信量となる．ネットワークの最大通信速度を s(bit/s)，各ノードの接続確率を $P_i = \frac{\text{他ノードとの接続時間 } (s)}{\text{自ノードの稼働時間 } (s)}$ とした場合に，ノード 1 が平均的に送信可能なデータ数は $s \times P_1$(bit/s) となる．ここで，各ノードのストリームデータ生成速度を g_i(bit/s) とすると，ノード 2 に許可されるデータ送信量は $s \times P_1 - g_1$ となる．ノード 2 で生成されるストリームデータ g_2 が，この許可されたデータ送信量を超えた場合には，周囲のノードのストレージを利用した分割配送処理を開始する．ノード 3 やノード 4 でストリームデータを生成した場合も同様の基準を用いる．

次に，多重化を考慮した分割配送について述べる．本手法では，以下のように，オリジナルデータに加えてコピーのデータを多重化配送することでデータの損失を防ぐ．

1. ストリームデータ生成ノード（子ノード）は受信側ノード（親ノード）に対して，送信したいデータのサイズを通知する．
2. 親ノードは子ノードに対して，接続確率およびストレージの空き容量を通知する．
3. 接続確率を基準に，データの分割配送先を決定する．空き容量が十分であれば，コピーを生成して多重化配送する．

分割配送を行うときには，接続確率の高いノードから

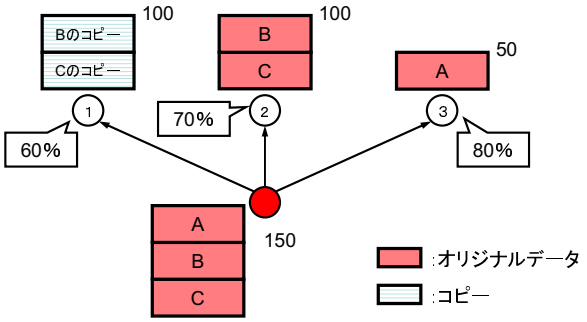


図 3　ストレージの空き容量に応じた多重化配送

順にオリジナルデータの配送を行う．オリジナルデータの配送が終了しても空き容量に余裕があるならば，最も接続確率の低いノードに配送されたオリジナルデータから順にコピーを生成し，多重化して配送する．

例えば，図 3 のような場合，生成した 150 件のオリジナルデータは，基地局との接続確率が 80 %であるノード 3 と 70 %であるノード 2 に配送される．その後，接続確率が 60 %のノード 1 のストレージには，150 件のデータのうち接続確率の低いノード 2 に配送された，データ B およびデータ C のコピーが配送される．このようにコピーを生成して多重化配送することにより，接続確率の低いノードに分割されたデータについても，基地局への到達率を上げることが可能となる．

4　実装と評価

本手法は，NS2[4] 上に実装し，シミュレーションによる評価を行う．具体的な評価項目としては，ノード上で生成されたストリームデータがどの程度基地局に到達したかを評価する予定である．全体のノード数，ストリームデータを生成するノード数やノードの移動性を考慮しながら，既存のストリームデータ処理システムおよび DTN（Delay and Disruption Tolerant Network）のルーティングプロトコルとの比較を行う．

5　おわりに

本稿では，移動センシング環境におけるストリームデータ分割配送手法を提案した．今後は，本手法の実装を進め，シミュレーションによる評価を行う予定である．

参考文献

[1] Lukasz Golab, and M. Tamer Ozsu: “Data Stream Management Issues - A Survey ,” SIGMOD Record, April 2003.

[2] The STREAM Group: “STREAM: The Stanford Data Stream Management System ,” IEEE Data Engineering Bulletin, March 2003.

[3] Abadi, D., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik: “Aurora: a new model and architecture for data stream management ,” The VLDB Journal, vol.12, pp.120-139, 2003.

[4] The Network Simulator -ns2. http://www.isi.edu/nsnam/ns/.

移動センシング環境における ストリームデータ分割配送手法

立命館大学　理工学研究科
大久保・横田研究室
村山知弥

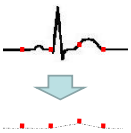
発表内容

- 背景
- 研究の目的
- 移動センシング環境におけるストリームデータ分割配送手法
  - センサノードの動作
  - 分割配送処理の詳細
- 実装と評価

はじめに(1/2)

- センサネットワーク
  - センサノードを配置してデータを取得
  - 例：1秒に1回気温データを取得する

- 移動センシング環境
  - 人と共にセンサノードが移動
  - バイタルデータの高精度な観測が可能
  - 高頻度かつ継続的に取得



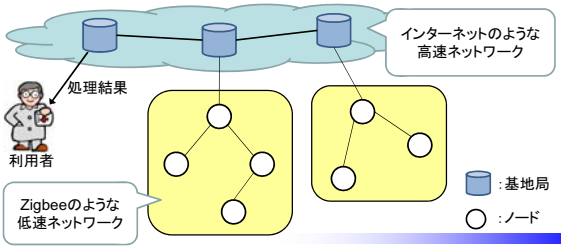
はじめに(2/2)

- ストリームデータ処理システム
  - 安定したネットワークを想定
  - 移動に伴うデータ損失の可能性

- ストリームデータ分割配送手法
  - より多くのデータが基地局に届く、信頼性の高いネットワークを構築

想定環境

- 被災地に低速ネットワーク
- 基地局間は高速ネットワークで接続

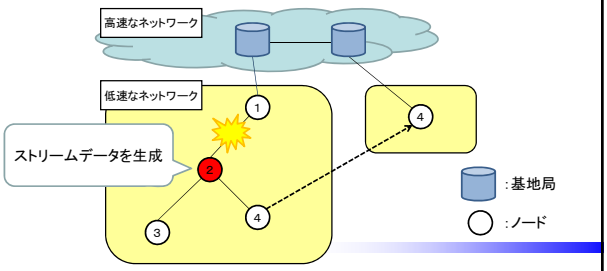


ストリーム処理における問題点

- シナリオ例
  - 被災地で「体温が一定値を超えた場合、その後の観測データをリアルタイムに基地局へ送信」といったクエリが実行される
- DSMS(Data Stream Management System)
  - 高速なネットワークを想定
  - ストリームデータに対する応答性を重視
- 移動センシング環境での問題点
  - ネットワークの状況の動的な変化
  - 低速ネットワークでのデータ損失

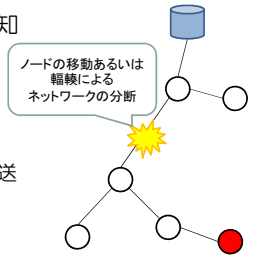
研究の目的

- より多くのデータが基地局(高速ネットワーク)に到達すること



移動センシング環境における ストリームデータ分割配送手法

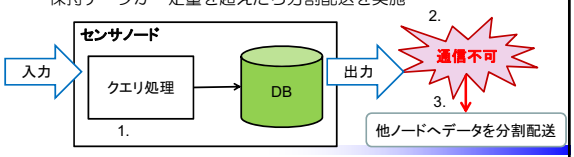
- ネットワークからの離脱を検知
  - 移動による離脱
  - 通信の輻輳による離脱
- 周囲のノードにデータを分配
  - 移動ノードを利用したデータ転送



データ損失のない信頼性の高いネットワークを構築

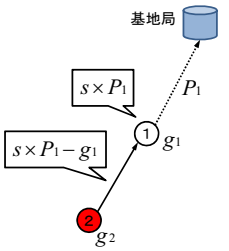
センサノードの動作

1. 観測処理
   観測データをストレージに格納してから基地局へ送信
2. 離脱検知処理
   ACKが返らなければネットワークからの離脱と判断
3. 分割配送処理
   保持データが一定量を超えたら分割配送を実施



分割配送の開始

- 通信速度を超えるストリームデータが生成した場合に分割配送処理を開始



s :最大通信速度(bit/s)

P_i : 他ノードとの接続時間(s) / 自ノードの稼働時間(s)

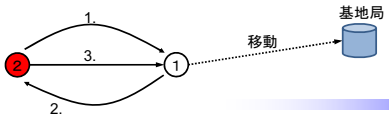
$s \times P_i$:確率上の通信速度(bit/s)

g_i :ストリームデータ生成頻度(bit/s)

観測ノードのストレージが一定量を超えており、
$g_2 > s \times P_1 - g_1$ ならば分割配送を開始

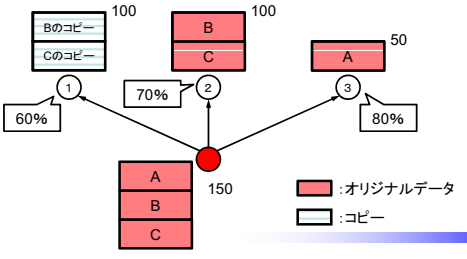
分割配送処理の流れ

1. 子ノードは親ノードに対して、送信したいデータサイズを通知
2. 親ノードは子ノードに接続確率 P_i 、ストレージ空き容量を通知
3. 子ノードは接続確率 P_i を基にデータを分割配送、空き容量が十分ならコピーと共に多重化配送



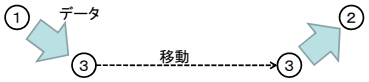
データ配送先の決定

- 接続確率の高いノードからデータを配送
- 接続確率の低いデータからコピーを生成



DTN(Delay and Disruption Tolerant Network)

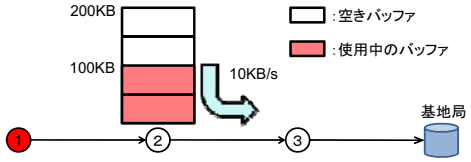
- 劣悪な通信環境を想定
- 通信するノード同士が接続されていなくても、移動ノードを用いたデータ転送が可能



- ストリームデータについては考慮されていない

提案手法でのデータ転送

- 200KBのうち空き容量が100KB
- ストレージ内データは10KB/sで送信処理



- 10秒後に200KB利用可能（20KB/sで送信）

実装と評価

- NetworkSimulator2上に実装
- シミュレーションによる評価
  - 具体的な評価項目
    - 生成されたデータに対する基地局に届いたデータの割合
  - パラメータを変えながら有効な環境を考察
    - 全体のノード数
    - ストリームデータ生成ノード数
    - ノードの移動性
- DSMSおよびDTNのプロトコルと比較

おわりに

- 発表内容
  - 移動センシング環境における問題点
  - ストリームデータ分割配送手法
    - センサノードの動作
    - 分割配送処理の詳細
- 今後の予定
  - シミュレータへの実装
  - シミュレーション評価

24 ピアの嗜好を考慮したコンテンツ検索手法

伊藤　雄一郎†
† 名古屋工業大学大学院　工学研究科　創成シミュレーション工学専攻

1　はじめに

近年、P2P(peer-to-peer) 技術はますます進歩しており、ストリーミング配信サービス、ネットワークゲームなどにもＰ２Ｐの技術が用いられている。その中でも P2P の技術が利用された有名な例は P2P ファイル共有アプリケーションが挙げられる。この P2P ファイル共有アプリケーションの例として、Gnutella[1]、BitTorrent[2]、Winny[3] などの例が挙げられる。これらのファイル共有アプリケーションは構造化されていない非構造型Ｐ２Ｐネットワークを想定している。これらのファイル共有アプリケーションでは、所望のコンテンツの検索効率を上げる研究が多くなされている。この検索効率を上げる研究として、ユーザの嗜好性を考慮したセマンティック P2P ネットワークの研究が盛んに行われている。セマンティック P2P ネットワークとは、同じような嗜好をもつピア同士がオーバーレイネットワーク上で隣接関係をもつネットワークのことを指している。この研究例としてコンテンのジャンル比情報を元にピア間の嗜好性の近さを決定し隣接関係を決定するセマンティック P2P ネットワーク構築法が提案されている。この手法の問題点として、ピアの嗜好性の近さの決定にジャンル比が大きいジャンルが隣接関係を決定する際に大きな影響を及ぼし、ジャンル比の低いジャンルに属するコンテンツ検索には不向きである点が指摘される。そこで本研究では、検索ホップ数に応じて検索クエリ転送先ポリシーを変えていき、ジャンル比率の低いジャンルに属するコンテンツ検索も一定の検索効率を保つオーバレイネットワーク構築法、コンテンツ検索手法を提案する。

2　既存手法

既存手法として、ピアの保持するコンテンツのジャンル比からピアの嗜好を決定しセマンティック P2P ネットワークを構築する手法が提案されている。[4] まず、ピア i は情報取得クエリを Gnutella 型のフラッディンにより伝播させることによって、周辺のピア情報（IP アドレス、ポート番号、保持コンテンツのジャンル比情報）を取得し、その情報を各ピアが保持しているピアリストと呼ばれるテーブルに格納する。ピアリストに格納されているピアから、重複率を計算し、重複率の高い 4 ピアを隣接ピアとしている。検索時には、その隣接ピアに検索クエリを転送する。重複率 $S(i,j)$ は次式で与えられる。

$$S(i,j) = \sum_{r \in G} min\{c_i(r), cj(r)\} \qquad (1)$$

なお、$min\{a,b\}$ は正数 a と b のうち小さな値をとる関数である。また、G はジャンルの集合を表し、$c_i(r)$ はピア i が保有するコンテンツについて、あるジャンル r に属するコンテンツの全保有コンテンツに対する割合を表している。

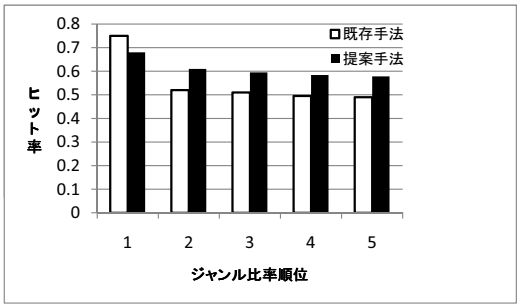
3　提案手法

提案手法として、隣接関係をもつピアを２つのポリシーにより決定し、検索時にホップ数に応じて検索クエリ転送先のポリシーを変えていく手法を提案する。まず、ピア i は情報取得クエリを Gnutella 型のフラッディングにより伝播し、ピア情報（IP アドレス、ポート番号保持コンテンツのジャンル比情報）を取得し、その情報を各ピアが保持しているピアリストと呼ばれるテーブルに格納する。ピアリストに格納されているピアから、２つの評価関数を用いて各評価関数ごとにスコアを算出し、各評価関数ごとにスコアの高い 4 ピアずつ検索クエリ転送先候補としてそれぞれ決定する。つまり、検索クエリ転送先を２つのポリシーにより決定し、ポリシー 1 により決定した転送先候補ピアを 4 ピア、ポリシー 2 により決定した転送先候補ピアを 4 ピアの合計 8 ピアを転送先候補ピアとしている。ポリシー 1 による検索クエリ転送先候補決定の際の評価関数 1 は（1）式、ポリシー 2 による検索クエリ転送先候補決定の際の評価関数 2 は次式で与えられる。

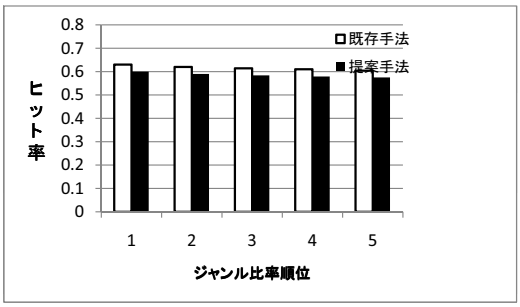
$$S(i,j) = \sum_{r \in G} |c_i(r) - cj(r)| \qquad (2)$$

検索時には検索クエリ転送先合計 8 ピアのうち、4 ピアに検索クエリを転送する。つまり、ポリシー 1 の転送先ピア数を m、ポリシー 2 の転送先ピア数を n とすると、$m+n=4$ となる。また、検索クエリのホップ数に応じて m,n の個数を変更していく。ホップ数が少ない場合には、m は大きく、ホップ数が大きい場合は n は大きくなる。1 ホップ目では $m=4, n=0$、2 ホップ目では、$m=3, n=1$、3 ホップ目では $m=2, n=2$ と変わっていく。このようにホップ数が 1 増えると、

† 名古屋工業大学, 名古屋市
Nagoya Institute of Technology,Gokis-cho,Showa-ku,Nagoya-shi,466-8555 Japan



ジャンル比偏りあり



ジャンル比偏りが一様に近い

Figure1:ジャンル比順位ごとのヒット率

	既存手法	提案手手法
偏りあり	0.619	0.628
偏りが一様に近い	0.638	0.592

Table2:システム全体のヒット率

m を 1 減らし、n を 1 増やし転送先ピアを変化させていく。これにより、ジャンル比が低いジャンルの属するコンテンツの検索もホップ数が大きくなるにつれヒット率が向上する。

4 評価

提案手法の有効性を検証するためにシミュレーションを行った。シミュレーションパラメータは Table1 に示す。

パラメータ	内容
ピア数	10000
ジャンル数	20
コンテンツ数	100000
ピアリスト容量	1000
ピアが保持するジャンル数	5
ピアが保持するコンテンツ数	20 100
情報取得クエリの TTL	5
検索クエリの TTL	4
検索時の TTL	5

Table1:シミュレーションパラメータ

評価では、各ピアごとにジャンル比の大きい順に順位をつけ、順位ごとのヒット率を評価した。各ピアは各順位に属するジャンルについてコンテンツの検索を行い、各順位ごとにヒット率を求めている。また、ここではピアのジャンル比分布を偏りがある場合と偏りが一様に近い場合の 2 通りの分布の場合の評価をとった。これは、ジャンル比分布の偏りにより検索効率の変化を調べるためである。この偏りは冪乗則に従うとし、係数を変化させ 2 通りの分布で評価をしている。また、ピアが保持すコンテンツ数、コンテンツの人気度は Zipf 則に従うものとした。

Figure1 はこの評価結果である。ジャンル比に偏りがある場合、ジャンル比順位 1 位では提案手法より既存手法のヒット率が高くなっている。一方で、ジャンル比順位が 2 位以下では、提案手法の方がヒット率が高くなっていることが分かる。本手法における目的であるジャンル比の低いジャンルについて検索効率を向上することができている。また、ジャンル比が一様に近づくにつれ提案手法の有効性が見られなくなっている。したがって、本手法はジャンル比に偏りがある場合有効であるといえる。これは、既存手法ではジャンル比が大きいジャンルが検索クエリ転送先の候補を決定する場合大きな影響を与えており、ジャンル比が小さいジャンルについてのヒット率を損なっていたためである。

さらに、検索時には実際に様々なジャンルのコンテンツを検索するため、ジャンル比の割合と同じ割合でコンテンツを検索した場合のヒット率の評価も行った。これも同様にジャンル比に偏りがある場合、偏りが一様に近い場合の 2 通りで評価を行っている。その評価結果は Table2 である。

5 まとめ

本論文では、ジャンル比の高いジャンルのコンテンツだけでなく、ジャンル比の低いジャンルのコンテンツも検索クエリ転送先ポリシーを変えていくことによって検索効率を上げる手法を提案した。本手法をシミュレーションにより、提案手法では、ジャンル比の低いジャンルもある程度優先した検索クエリ転送先決定法により、ジャンル比の低いジャンルのコンテンツについて既存手法に比べ高いヒット率が得られた。また、ジャンル比分布を変化させ、ジャンル比に偏りがある場合に提案手法は有効であることを確認した。

今後の課題は、ピア間のリンクの管理コスト、検索メッセージ数の削減などが挙げられる。

参考文献

[1] ”gnutella”,http://gnutella.wago.com/

[2] ”BitTorrenr”,http://www.bittorrent.com/

[3] ”Winny”,http://winny.info/

[4] 大林功実, 朝香卓也, 高橋達郎, 佐々木純, 品川準輝,” ピア-ピア間スループットを考慮したセマンティック P2P ネットワークのトポロジ構築法” 電子情報通信学会論文誌,Vol.J91-B No.1,pp.35-46,January 2008.

[5] 中河隆仁, 森友則, 朝香卓也, 高橋達郎” クリッピングと保持コンテンツ数の偏りを考慮したセマンティック P2P ネットワークの構築法法” 電子情報通信学会論文誌,Vol.J93-B No.2,pp.230-241,February 2010.

ピアの嗜好を考慮したコンテンツ検索手法

名古屋工業大学
伊藤　雄一郎

目次

- 研究背景
  - セマンティックP2Pネットワーク
- 既存手法
  - ジャンル比によるピアの嗜好を考慮したP2Pネットワーク
- 提案手法
  - ホップ数に応じた検索クエリ転送先決定
- シミュレーション評価
  - 各Rankごとのヒット率
  - システム全体のヒット率
  - ホップ数ごとのヒット数
- まとめ

研究背景

- P2P(peer to peer)技術の進歩
  - ファイル共有アプリケーション
    - Gnutella , BitTorrent , Winny　等
- ファイル共有アプリケーションの改良
  - 検索効率の更なる向上
- ユーザの嗜好性を考慮したP2Pネットワーク
  - セマンティックP2Pネットワーク

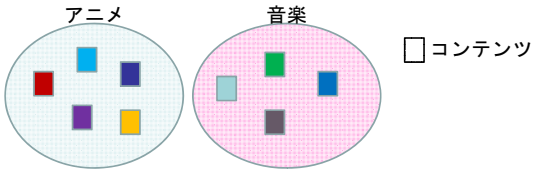
セマンティックP2Pネットワーク

- 類似した嗜好をもつピア同士がオーバレイネットワーク上で隣接関係をもつ
  - 嗜好・・・ユーザがどういったコンテンツに興味があるか
- 既存手法†
  - ピアの保持コンテンツのジャンル保持比による嗜好性の決定
  - 嗜好性の近いピアを検索クエリ送出先とする
  - ピア間の嗜好性の近さはジャンル比率の重複度（後述）により決定
  - 重複度の高いピアと隣接関係を持つことによる検索効率の向上

† ファイルダウンロード時間を考慮したセマンティックP2Pネットワークのトポロジ構築法（大林　功実ら　2008）

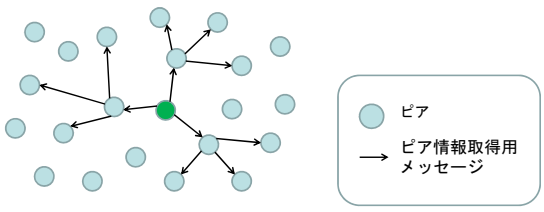
セマンティックP2Pネットワーク

- ジャンル
  - 例…音楽/アニメ/映画/ドラマ etc
  - コンテンツは一意にジャンル分けされている
- コンテンツ
  - 音楽タイトル/アニメタイトル/映画タイトル/ドラマタイトル



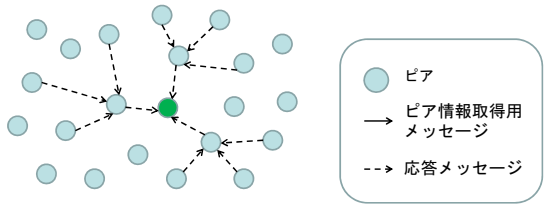
既存手法手順

- STEP1　ピア情報取得
  - ピア情報…IPアドレス・ポート番号・ジャンルの比率情報
  - ピア情報取得用メッセージを伝搬し周辺のピア情報を取得



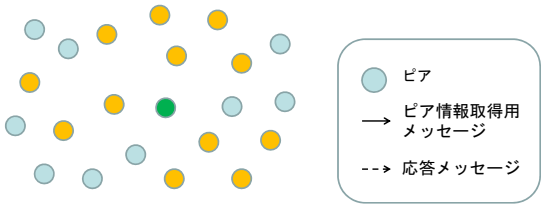
既存手法手順

- STEP1　ピア情報取得
  - ピア情報…IPアドレス・ポート番号・ジャンルの比率情報
  - ピア情報取得用メッセージを伝搬し周辺のピア情報を取得



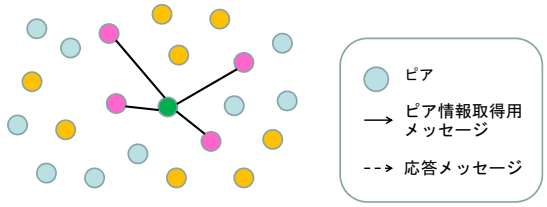
既存手法手順

- STEP2　隣接ピア決定
  - 取得できたピア情報をピアリストに追加
    - ピアリスト‥ピア番号・ジャンルの比率情報のリスト
  - ピアリストのピアから隣接ピアを重複度（後述）により決定
    - 重複度の高いピア上位　n ピアを検索クエリ転送先を決定



既存手法手順

- STEP2　隣接ピア決定
  - 取得できたピア情報をピアリストに追加
    - ピアリスト‥ピア番号・ジャンルの比率情報のリスト
  - ピアリストのピアから隣接ピアを重複度（後述）により決定
    - 重複度の高いピア上位　n ピアを検索クエリ転送先を決定



既存手法

- ピア間の嗜好性の近さ
  - 重複度 $S(i,j)$ により決定

$$S(i,j)=\sum_{r\in G}\min\{c_i(r),c_j(r)\}$$

G：　ジャンルの集合

$c_i(r)$：あるジャンルrに属するコンテンツの全保有コンテンツに対する割合

- 重複度の大きい上位 n 個のピアを選択し検索クエリの転送先とする

既存手法

- 重複度S(A, B)（計算例）

$S(\mathrm{A},\mathrm{B})$ ＝

ジャンル	比率
音楽	0.4
アニメ	0.2
映画	0.1
ドラマ	0.3
学問	

ピアA

ジャンル	比率
音楽	0.2
アニメ	0.3
映画	0.4
ドラマ	
学問	0.1

ピアB

既存手法

- 重複度S(A, B)（計算例）

$S(\mathrm{A},\mathrm{B})$ ＝ 0.2

ジャンル	比率
音楽	0.4
アニメ	0.2
映画	0.1
ドラマ	0.3
学問	

ピアA

ジャンル	比率
音楽	0.2
アニメ	0.3
映画	0.4
ドラマ	
学問	0.1

ピアB

既存手法

- 重複度S(A, B)（計算例）

$$S(A,B) = 0.2 + 0.2$$

ジャンル	比率
音楽	0.4
アニメ	0.2
映画	0.1
ドラマ	0.3
学問	

ピアA

ジャンル	比率
音楽	0.2
アニメ	0.3
映画	0.4
ドラマ	
学問	0.1

ピアB

既存手法

- 重複度S(A, B)（計算例）

$$S(A,B) = 0.2 + 0.2 + 0.1$$

ジャンル	比率
音楽	0.4
アニメ	0.2
映画	0.1
ドラマ	0.3
学問	

ピアA

ジャンル	比率
音楽	0.2
アニメ	0.3
映画	0.4
ドラマ	
学問	0.1

ピアB

既存手法

- 重複度S(A, B)（計算例）

$$S(A,B) = 0.2 + 0.2 + 0.1 = 0.5$$

ジャンル	比率
音楽	0.4
アニメ	0.2
映画	0.1
ドラマ	0.3
学問	

ピアA

ジャンル	比率
音楽	0.2
アニメ	0.3
映画	0.4
ドラマ	
学問	0.1

ピアB

既存手法の問題点

- ジャンル比率の低いジャンルが接続先決定の際に優先されない
  - コンテンツ保持比率の低いジャンルのコンテンツの検索が向上する接続先とは限らない
- ジャンル比率の低いジャンルのコンテンツの検索効率比較的悪い

既存手法の問題点

ジャンル	比率
音楽	0.1
ゲーム	0.4
アニメ	0.1
映画	0.4
ドラマ	
学問	
スポーツ	

ピアC

ジャンル	比率
音楽	0.5
ゲーム	
アニメ	0.2
映画	
ドラマ	
学問	0.2
スポーツ	0.1

ピアB

ジャンル	比率
音楽	0.5
ゲーム	0.2
アニメ	
映画	0.1
ドラマ	0.2
学問	
スポーツ	

ピアA

既存手法の問題点

ジャンル	比率
音楽	0.1
ゲーム	0.4
アニメ	0.1
映画	0.4
ドラマ	
学問	
スポーツ	

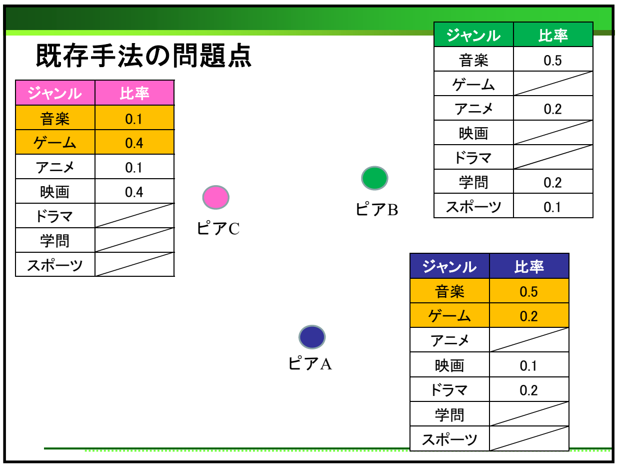
ピアC

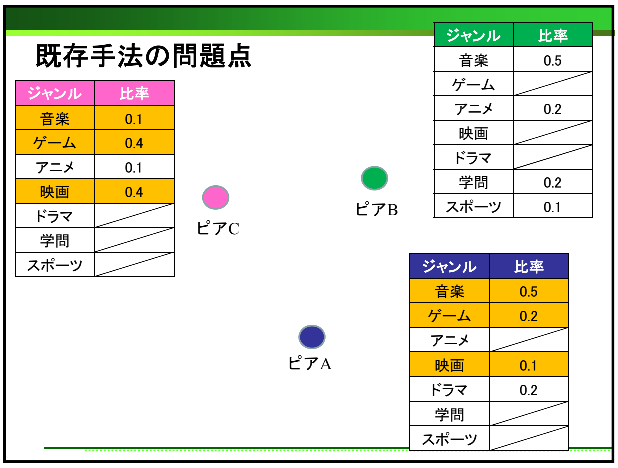
ジャンル	比率
音楽	0.5
ゲーム	
アニメ	0.2
映画	
ドラマ	
学問	0.2
スポーツ	0.1

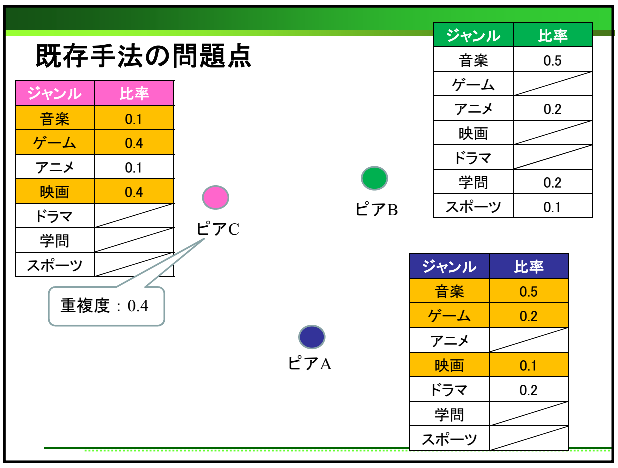
ピアB

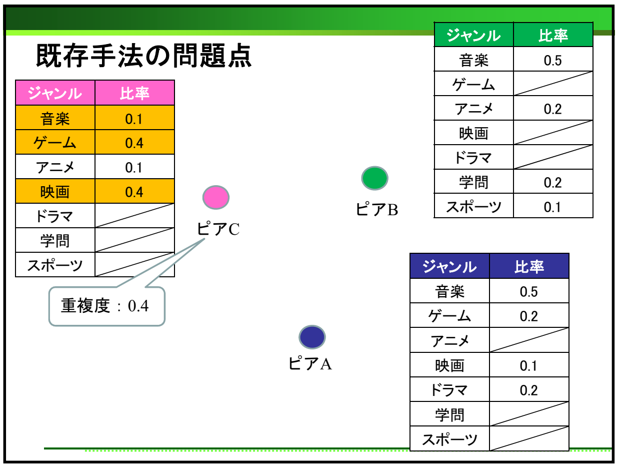
ジャンル	比率
音楽	0.5
ゲーム	0.2
アニメ	
映画	0.1
ドラマ	0.2
学問	
スポーツ	

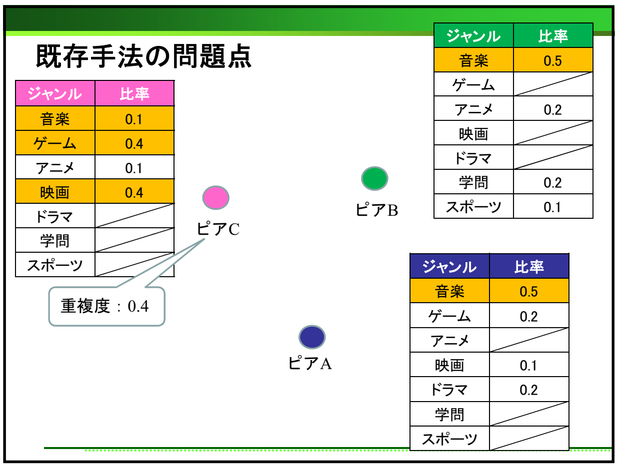
ピアA

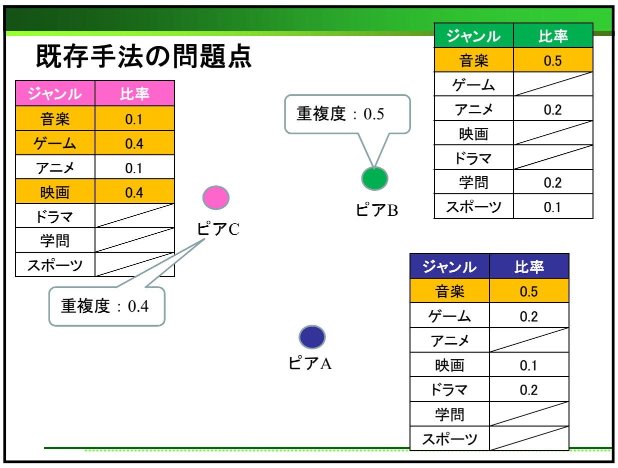

既存手法の問題点
ジャンル 比率
音楽 0.1
ゲーム 0.4
アニメ 0.1
映画 0.4
ドラマ
学問
スポーツ
ピアC
ピアB
ジャンル 比率
音楽 0.5
ゲーム
アニメ 0.2
映画
ドラマ
学問 0.2
スポーツ 0.1
ピアA
ジャンル 比率
音楽 0.5
ゲーム 0.2
アニメ
映画 0.1
ドラマ 0.2
学問
スポーツ


既存手法の問題点
ジャンル 比率
音楽 0.1
ゲーム 0.4
アニメ 0.1
映画 0.4
ドラマ
学問
スポーツ
ピアC
ピアB
ジャンル 比率
音楽 0.5
ゲーム
アニメ 0.2
映画
ドラマ
学問 0.2
スポーツ 0.1
ピアA
ジャンル 比率
音楽 0.5
ゲーム 0.2
アニメ
映画 0.1
ドラマ 0.2
学問
スポーツ


既存手法の問題点
ジャンル 比率
音楽 0.1
ゲーム 0.4
アニメ 0.1
映画 0.4
ドラマ
学問
スポーツ
ピアC
重複度：0.4
ピアB
ジャンル 比率
音楽 0.5
ゲーム
アニメ 0.2
映画
ドラマ
学問 0.2
スポーツ 0.1
ピアA
ジャンル 比率
音楽 0.5
ゲーム 0.2
アニメ
映画 0.1
ドラマ 0.2
学問
スポーツ

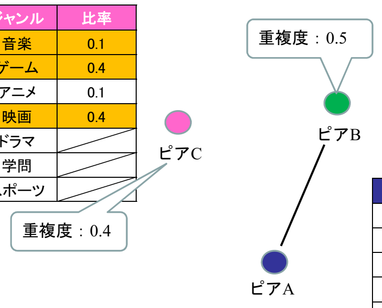

既存手法の問題点
ジャンル 比率
音楽 0.1
ゲーム 0.4
アニメ 0.1
映画 0.4
ドラマ
学問
スポーツ
ピアC
重複度：0.4
ピアB
ジャンル 比率
音楽 0.5
ゲーム
アニメ 0.2
映画
ドラマ
学問 0.2
スポーツ 0.1
ピアA
ジャンル 比率
音楽 0.5
ゲーム 0.2
アニメ
映画 0.1
ドラマ 0.2
学問
スポーツ


既存手法の問題点
ジャンル 比率
音楽 0.1
ゲーム 0.4
アニメ 0.1
映画 0.4
ドラマ
学問
スポーツ
ピアC
重複度：0.4
ピアB
ジャンル 比率
音楽 0.5
ゲーム
アニメ 0.2
映画
ドラマ
学問 0.2
スポーツ 0.1
ピアA
ジャンル 比率
音楽 0.5
ゲーム 0.2
アニメ
映画 0.1
ドラマ 0.2
学問
スポーツ


既存手法の問題点
ジャンル 比率
音楽 0.1
ゲーム 0.4
アニメ 0.1
映画 0.4
ドラマ
学問
スポーツ
ピアC
重複度：0.4
重複度：0.5
ピアB
ジャンル 比率
音楽 0.5
ゲーム
アニメ 0.2
映画
ドラマ
学問 0.2
スポーツ 0.1
ピアA
ジャンル 比率
音楽 0.5
ゲーム 0.2
アニメ
映画 0.1
ドラマ 0.2
学問
スポーツ

既存手法の問題点

ジャンル	比率
音楽	0.1
ゲーム	0.4
アニメ	0.1
映画	0.4
ドラマ	
学問	
スポーツ	

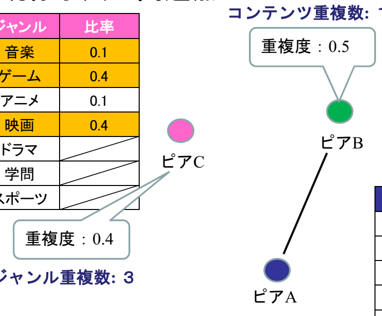


ジャンル	比率
音楽	0.5
ゲーム	
アニメ	0.2
映画	
ドラマ	
学問	0.2
スポーツ	0.1

ジャンル	比率
音楽	0.5
ゲーム	0.2
アニメ	
映画	0.1
ドラマ	0.2
学問	
スポーツ	

既存手法の問題点

ジャンル	比率
音楽	0.1
ゲーム	0.4
アニメ	0.1
映画	0.4
ドラマ	
学問	
スポーツ	



ジャンル	比率
音楽	0.5
ゲーム	
アニメ	0.2
映画	
ドラマ	
学問	0.2
スポーツ	0.1

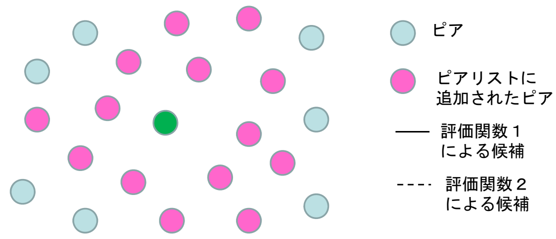
ジャンル	比率
音楽	0.5
ゲーム	0.2
アニメ	
映画	0.1
ドラマ	0.2
学問	
スポーツ	

提案手法

- 検索時ホップ数に応じて検索クエリ転送先のポリシーを変えていく
  - 少ないホップ数で見つからない場合同様の嗜好のピアに検索クエリを転送しても見つからない
  - 転送先の方針を変えることにより検索効率を向上させる
  - 2種類の方針により各方針ごとに2種類の検索クエリ転送先候補を作る
- ジャンル比率が低いジャンルのコンテンツに関しても検索効率を向上させる

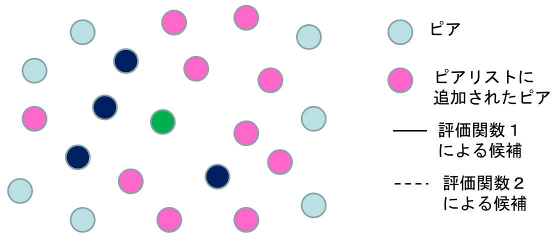
提案手法詳細

- 転送先候補ピアの決定
  - 2つの評価関数を用いてスコアを算出
    - 各評価関数で上位ピアを候補先とする
  - 2種類の方針で決定した検索クエリ転送先候補をもつ



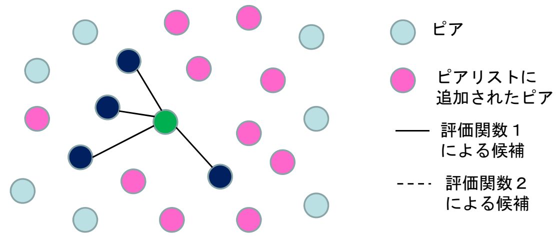
提案手法詳細

- 転送先候補ピアの決定
  - 2つの評価関数を用いてスコアを算出
    - 各評価関数で上位ピアを候補先とする
  - 2種類の方針で決定した検索クエリ転送先候補をもつ



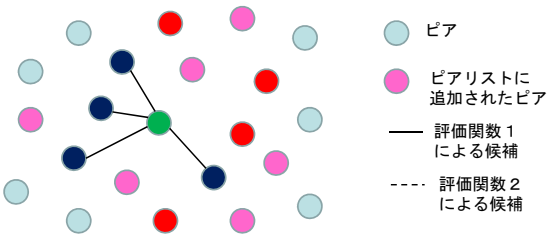
提案手法詳細

- 転送先候補ピアの決定
  - 2つの評価関数を用いてスコアを算出
    - 各評価関数で上位ピアを候補先とする
  - 2種類の方針で決定した検索クエリ転送先候補をもつ



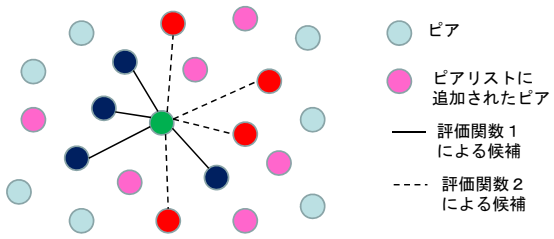
提案手法詳細

- 転送先候補ピアの決定
  - 2つの評価関数を用いてスコアを算出
    - 各評価関数で上位ピアを候補先とする
  - 2種類の方針で決定した検索クエリ転送先候補をもつ



提案手法詳細

- 転送先候補ピアの決定
  - 2つの評価関数を用いてスコアを算出
    - 各評価関数で上位ピアを候補先とする
  - 2種類の方針で決定した検索クエリ転送先候補をもつ



提案手法詳細

- 評価関数1（以下方針1）
  - 既存手法と同様 $S(i,j)=\sum_{r\in G}\min\{c_i(r),c_j(r)\}$
- 評価関数2（以下方針2）

$$T(i,j)=\sum_{r\in G}|\,c_i(r)-c_j(r)\,|$$

G： ジャンルの集合

$c_i(r)$ ：あるジャンルｒに属するコンテンツの全保有コンテンツに対する割合

提案手法詳細

- 検索
  - 検索クエリのホップ数が1増えるごとに方針1による転送先ピアを減らし方針2による転送先を増やす

□１ホップ目

提案手法詳細

- 検索
  - 検索クエリのホップ数が1増えるごとに方針1による転送先ピアを減らし方針2による転送先を増やす

□１ホップ目

提案手法詳細

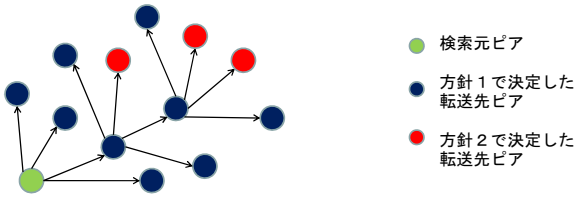
- 検索
  - 検索クエリのホップ数が1増えるごとに方針1による転送先ピアを減らし方針2による転送先を増やす

□２ホップ目

提案手法詳細

- 検索
  - 検索クエリのホップ数が1増えるごとに方針1による転送先ピアを減らし方針2による転送先を増やす

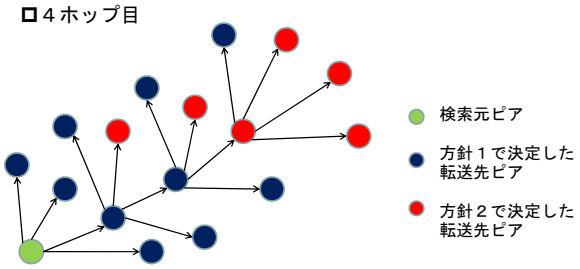
□３ホップ目



提案手法詳細

- 検索
  - 検索クエリのホップ数が1増えるごとに方針1による転送先ピアを減らし方針2による転送先を増やす

□４ホップ目



シミュレーション評価

- 各ジャンル比率の順位ごとのヒット率
- システム全体のヒット率
- ホップ数ごとのヒット数

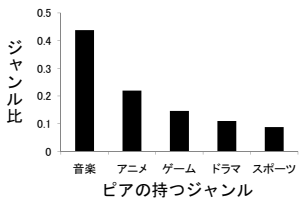
シミュレーション評価

- シミュレーション条件

初期トポロジ	BAモデル
ピア数	10000
ジャンル数	20
コンテンツ数	100000
ピアが保持するコンテンツ数	20～100　Zipf則に従う偏りで重みつけられる
ピアが保持するジャンル数	5
情報取得クエリのTTL	5
ピアリスト容量	ピア数の10分の1
検索クエリ転送先ピア数	4
検索時のTTL	4
ジャンル比率分布	$f(x) \propto cx^{\alpha}$ に従い $\alpha = -1, -\frac{1}{2}, -\frac{1}{3}$
ジャンル存在分布	$h(x) \propto dx^{\beta}$ に従い $\beta = -1, -\frac{1}{2}, -\frac{1}{3}$,均一

シミュレーション評価（パラメータ変化）

- ジャンル比分布
  - ピアがジャンル毎に持つコンテンツ数の割合
- $f(x) \propto cx^{\alpha}$ に従い $\alpha = -1, -\frac{1}{3}$ に変化
  - 偏りがある場合・比較的一様な場合

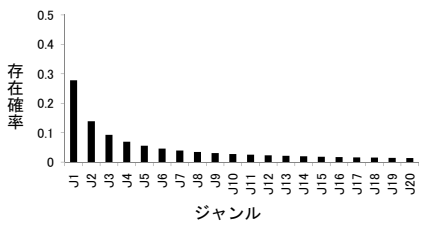


例）ピアの保持するコンテンツのジャンル比

ジャンル	比率
音楽	0.44
アニメ	0.22
ゲーム	0.15
ドラマ	0.11
スポーツ	0.08

シミュレーション評価（パラメータ変化）

- ジャンル存在分布
  - ピアがどのジャンルを保持しやすいのか
- $h(x) \propto dx^{\beta}$ に従い $\beta = -1, -\frac{1}{3}$ または均一に変化
  - 偏りがある場合・比較的一様な場合



（評価1）ジャンル比順位別のヒット率

- ジャンル比順位別のヒット率
  - ジャンル比率順に順位（以下Rank）をつける
  - 各Rankに属するジャンルのコンテンツを10000回検索
  - 各Rankごとのヒット率を求める
- Rankごとに検索効率の違いを評価

例） ピアAのコンテンツ情報

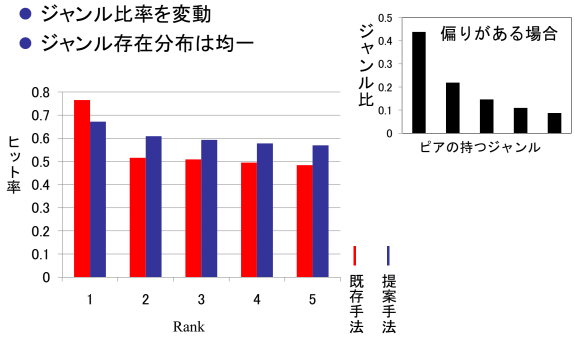
ジャンル	ジャンル比率	Rank
アニメ	0.44	1
音楽	0.22	2
ドラマ	0.15	3
映画	0.11	4
学問	0.08	5

ヒット率の計算

$$\text{Rank Nのヒット率} = \frac{\text{Rank Nのジャンルのコンテンツの発見数}}{\text{Rank N のジャンルのコンテンツの検索数}}$$

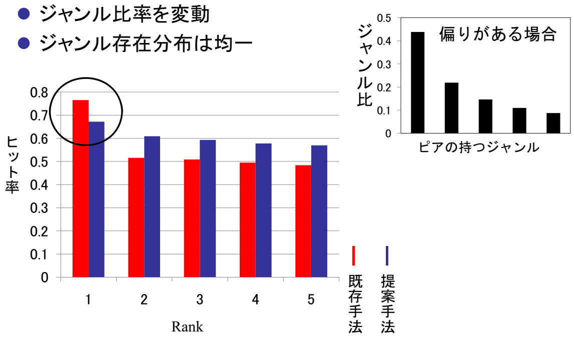
（評価1）ジャンル比順位別のヒット率

- ジャンル比率を変動
- ジャンル存在分布は均一



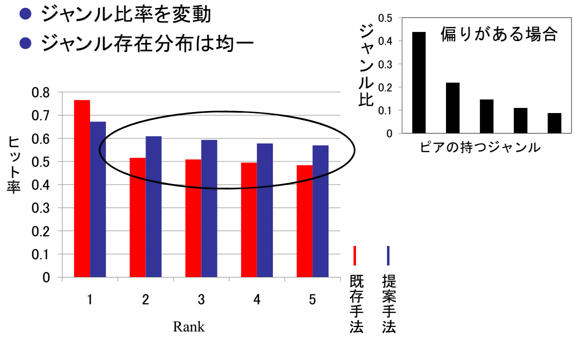
（評価1）ジャンル比順位別のヒット率

- ジャンル比率を変動
- ジャンル存在分布は均一



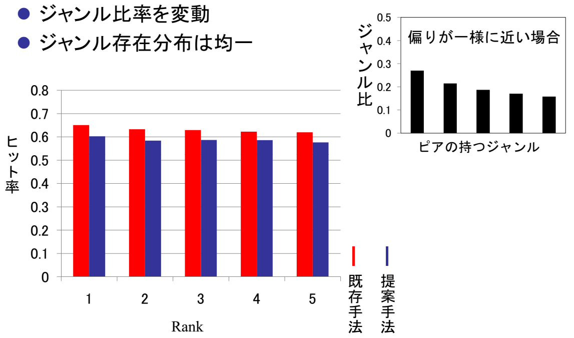
（評価1）ジャンル比順位別のヒット率

- ジャンル比率を変動
- ジャンル存在分布は均一



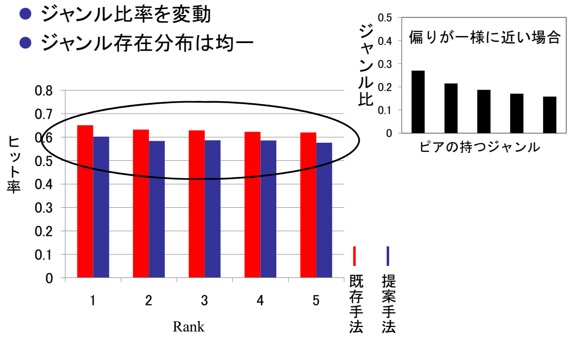
（評価1）コンテンツ保持比順位別のヒット率

- ジャンル比率を変動
- ジャンル存在分布は均一



（評価1）コンテンツ保持比順位別のヒット率

- ジャンル比率を変動
- ジャンル存在分布は均一



(評価1)ジャンル比順位別のヒット率

考察

- ジャンル比率の分布の偏りが強い場合
  - 下位Rankのジャンルにおいて提案手法が有効
- ジャンル比率の分布が一様に近づくに場合
  - 全てのRankにおいて既存手法が有効

- ジャンル比率が大きいジャンルが検索効率に大きく影響を与える

(評価2)システム全体のヒット率

- システム全体のヒット率
  - 各ピアは保持する各ジャンル比率に従い検索するコンテンツを決定
  - 10000回コンテンツを検索しヒット率を求める
- 実環境に近い状況でのヒット率を求める

例) ピアAのコンテンツ情報

ジャンル	ジャンル比率	検索確率
アニメ	0.44	0.44
音楽	0.22	0.22
ドラマ	0.15	0.15
映画	0.11	0.11
学問	0.08	0.08

ヒット率の計算

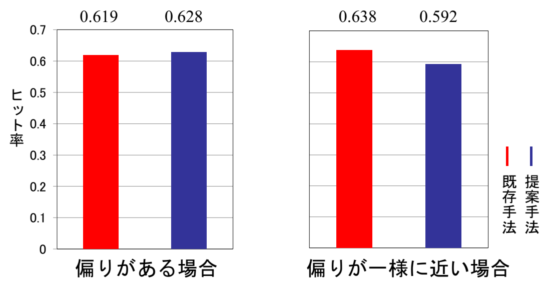
$$\text{ヒット率} = \frac{\text{コンテンツの発見数}}{\text{コンテンツの検索数}}$$

(評価2)システム全体のヒット率

- パラメータ変化
  - ジャンル比変化
    - ジャンル比を変化させた場合の既存手法・提案手法の有効性の確認
    - ジャンル比に偏りがある場合・一様に近い場合
  - ジャンル存在分布
    - ジャンル存在分布が偏っている場合提案手法の有効性が変化する場合がある?
    - ジャンル存在分布に偏りがある場合・一様な場合

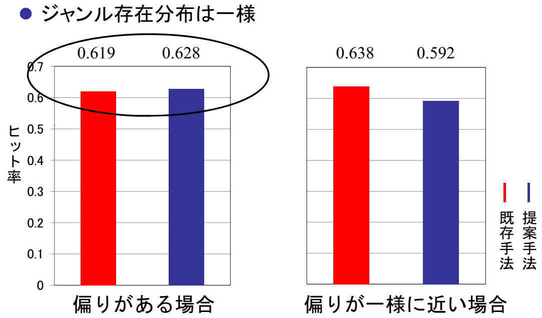
(評価2)システム全体のヒット率

- ジャンル比率の分布を変動
- ジャンル存在分布は一様



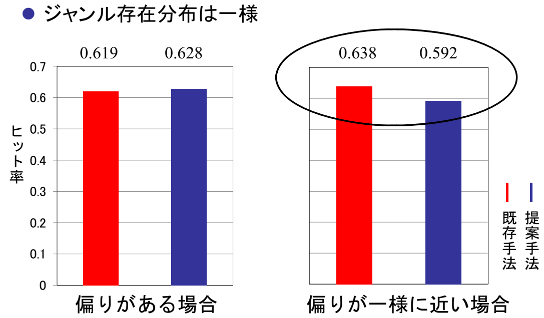
(評価2)システム全体のヒット率

- ジャンル比率の分布を変動
- ジャンル存在分布は一様



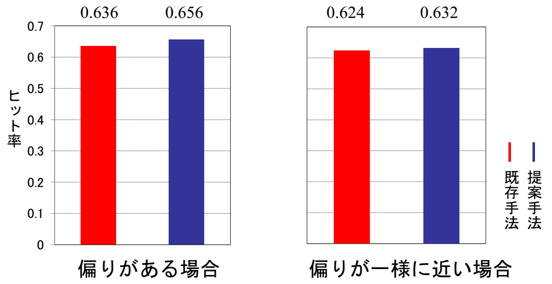
(評価2)システム全体のヒット率

- ジャンル比率の分布を変動
- ジャンル存在分布は一様



（評価2）システム全体のヒット率

- ジャンル存在分布を変動
- ジャンル比率は偏りがある場合



（評価2）システム全体のヒット率

結果

- ジャンル比分布変化
  - ジャンル比分布の偏りがある場合
    - 既存手法と提案手法はほぼ同程度のヒット率
  - ジャンル比分布が一様に近づく場合
    - 提案手法は既存手法よりヒット率が低くなる
- ジャンル存在分布
  - ジャンル存在分布に偏りがあった場合でも提案手法の有効性は下がらない

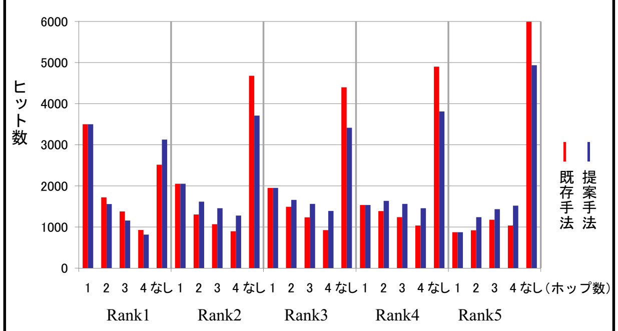
（評価3）ホップ数毎のヒット数

- ジャンル比順位別・ホップ数毎のヒット数
- ジャンル比率順に順位（以下Rank）をつける
  - 各Rankに属するジャンルのコンテンツを10000回検索
  - 各Rankごと・ホップ数ごとのヒット数
- パラメータ
  - ジャンル比に偏りがある場合
  - ジャンル存在分布は一様

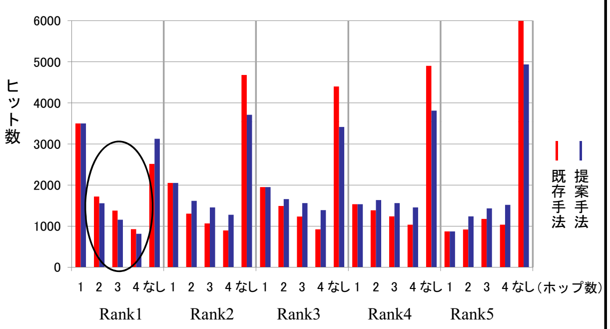
例） ピアAのコンテンツ情報

ジャンル	ジャンル比率	Rank
B	0.44	1
D	0.22	2
E	0.15	3
A	0.11	4
C	0.08	5

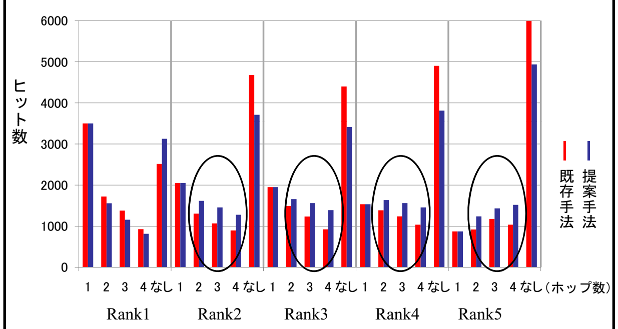
（評価3）ホップ数毎のヒット数



（評価3）ホップ数毎のヒット数



（評価3）ホップ数毎のヒット数



（評価3）ホップ数毎のヒット数

考察

- 上位Rank
  - 既存手法のほうがヒット数が多い
  - ホップ数が増すごと既存手法のヒット数と提案手法のヒット数との差が開いている
  - ホップ数を増すごとに違うポリシーの転送先が増加
- 下位Rank
  - 提案手法のほうがヒット数が多い
  - ホップ数が増すごと既存手法のヒット数と提案手法のヒット数との差が開いている
  - ホップ数を増すごとに違うポリシーの転送先が増加

まとめ・今後の課題

- ホップ数による検索クエリ転送先を変更する検索手法を提案
  - 低Rankにおける提案手法の有効性を確認
  - ホップ数を増すごとに提案手法の有効性を確認
  - ジャンル比率の偏りが大きいほど提案手法の有効性は大きい
- 今後の課題
  - より効率の良いルーティング手法の提案
  - ピア間のリンクのメンテナンスコストなど総パケット数の改善

編集後記

今回、プログラム編集委員をさせていただきました。発表して頂いた学生の方、お忙しい中ご出席くださった先生方、および JSASS の各委員担当の皆様に改めてお礼申し上げます。私自身、学生時代に第 1 回から第 5 回まで参加していました。卒業後は ATR に就職し、今年度より立命館大学へ戻ってきましたので、今回が 5 年ぶりの参加でした。以前同様、専門・専門外に関わらず多くの発表を聴くことができ、勉強になると同時に学生に負けないよう頑張る活力とすることができました。ただ、なんとなくですが雰囲気が少し硬くなったように感じます。JSASS だからこその活発な議論ができる雰囲気づくりが必要だと感じた次第です（無論、私のような若手が頑張らねばならないのですが)。

とはいえ、このようなプライベートな研究発表の場は他でもあるのでしょうが、JSASS のように 5 大学を超える規模で 10 年以上にわたって行われるものは稀ではないでしょうか。これもひとえに諸先生方のご尽力だと思います。私も微力ではありますが、こらからの JSASS をよりよい発表の場、交流の場となるようお手伝いできればと思います。

立命館大学　瀧本 栄二

今年の JSASS は，名古屋工業大学で行われた．今年の夏は非常に暑いと言われているが，JSASS 実施日も夜まで暑い日であった．そのようななかでも，多くの研究室から参加をしてもらうことができた．発表も，長い時間をかけて進めた研究であることを想像できるものが多く，たいへん良かったと思う．

今回の JSASS では，昨年に比べ多くの発表があった．発表件数が多くなったため，1 件あたりの発表時間を短縮することになったが，それでも多くの発表を興味深く聞くことができた．また，多くの研究分野からの発表があり，JSASS の広がりを感じることができた．参加した多くの学生は，自身の所属する研究室では扱っていないテーマの発表を見付けることができたのではないだろうか．興味をもった発表をもとに研究の視野を広げてもらえれば嬉しく思う．

龍谷大学　芝 公仁

今回の名古屋開催は，私や齋藤先生が事前に打ち合わせて決めたが，その理由はいくつかある．一つは，名古屋が地理的に便利であり，関西・中国圏からも，関東圏からもアクセスが良いという点である．もう一つは，名古屋の食べ物のおいしさもある．どの地域にもそこに根ざした食文化があるが，名古屋の食文化は特に魅力的である．これまで，きしめんや味噌カツといった有名なものを口にする機会はあったが，「ひつまぶし」に出会ったことがなかった．今回は，ひつまぶしにトライすべく，名工大へ行く途中にお昼ご飯としていただこうと計画した．午前 11 時過ぎ，名古屋駅に降り立って早速店を探し始めた．しかし，近くのレストラン街を歩けど歩けど見当たらない．そういえば，ひつまぶしはどのジャンルの食べ物だろうかと，それさえわかっていないことに気づいた．定食屋，鰻専門店，そば屋，…どこに行けばいいのか？名物だから，すぐに見つかるだろうと考えていたのが甘かった．歩くこと 30 分以上．やっと寿司屋でひつまぶしを発見．しかし，驚くべきことに，高い．2500 円．確かに鰻重のことを思えば，その値段も不思議ではないのかもしれない．ただ，歩き疲れたところで見たこの値段は致命的であった．さらに，「20 分程度かかります」の文字．これでは遅刻してしまう．幹事がひつまぶし食べたさに遅刻ということは避けねばなるまい．今回は泣く泣く撤退を決めた…．ばかばかしい話であるが，名古屋以外から参加した諸君も多かったと思う．せっかく来たその土地を楽しんだであろうか？研究も大切だけれども，視野を広く持ち，楽しんでほしい．

少しはまじめな話も書こう．JSASS でここ最近悩んでいることがある．それは，JSASS が，学会開催の研究会とさほど変わらないように感じているところである．ただし，研究会の善し悪しを論じているのではない．JSASS のあるべき姿を論じたい．私は，何か，もっと，ワクワク・ドキドキするような，そんな JSASS にできないだろうかと思っている．その答えははっきりしないものの，ひとつは「発表者自身がその研究を楽しいと感じている」のが重要だと思う．その楽しさが伝わってきたとき，聴衆もワクワク・ドキドキしそうである．もう一つは「研究の中で行った試行錯誤の過程」がキーではないかと思う．発表者は何を悩んだのか．どんな解法を検討してみたのか．解法の検討の過程での失敗談・成功談，そこで得られた知見やエピソードなど．これらは，普段の研究会では紹介されないことであるが，意義高くておもしろい．そして最後に，その試行錯誤の中で，「ある一定の結果を得られた」ことだと思う．これは，研究の完了や品質の高い評価だけを意味しているのではない．研究の一部でいいので，今回チャレンジしたことについて「ここはわかった！」と自信を持って言える部分があるということである．おそらく，この 3 つが揃えば，ワクワク・ドキドキできるのではないかと期待している．この 3 つの中に「おもしろい研究テーマ」というのが含まれていないが，この 3 つを満たしていれば，それはおもしろい研究だろうと想像している．このようなワクワク・ドキドキを実現するために，JSASS としてできることはあるか？最近，こんなことを考えている．ぜひ皆さんからもアイディアを！

立命館大学　毛利 公一

先進的基盤ソフトウェア
Joint Symposium for Advanced System Software 2010 (JSASS2010)

4 巻 1 号（通号 4 号）オンライン版　2010 年 10 月 22 日発行

編　　集　毛利 公一，芝 公仁，瀧本 栄二，並木 美太郎
委 員 長　大久保 英嗣
発 行 者　JSASS 実行委員会
〒525-8577 滋賀県草津市野路東１－１－１
立命館大学情報理工学部 毛利研究室内
電話 077-561-5061
発 行 所　〒184-8588 東京都小金井市中町２－２４－１６
東京農工大学工学部 並木研究室
電話 042-388-7139