先進的基盤ソフトウェア 5巻1号（通号5号）オンライン版 2011年10月28日発行
ISSN 1882-4196

先進的基盤ソフトウェア

Joint Symposium for Advanced System Software 2011
(JSASS2011)

2011年8月24日(水)・25日(木)

於 龍谷大学 (滋賀県大津市)

JSASS実行委員会

巻頭言

龍谷大学　芝 公仁

2011 年の Joint Symposium for Advanced System Software (JSASS) は，龍谷大学瀬田学舎で行われました．各地から参加いただきありがとうございました．JSASS はこれまで大学など様々な所で開催されてきましたが，龍谷大学での開催は 2 回目となります．開催期間は授業期間外であり，キャンパス内はそれほど騒がしくなく，議論に集中できたのではないでしょうか．

今回の JSASS での発表は，オペレーティングシステム，ネットワーク，セキュリティなどといったものでした．これらは，JSASS が始まったころから変わらないものです．いずれも，様々な環境において基盤となる技術であり，その重要性は当時から変わっていません．また，今後も重要な基盤技術でありつづけると思います．一方，個々の研究内容は複雑になってきています．ハードウェアなどのアーキテクチャの高度化や，アプリケーションからの要求が複雑になった結果であると思います．どのような要求があるか，それを実現するためにはどのようにすればよいか，これらを検討することは研究の面白さのひとつです．また，その検討結果に基づき実際にシステムを構築し動作させることができ，それが何かの役に立ったとき，非常に嬉しいものです．研究を進めるにあたっては，有用性，独自性を確立していくことが必要です．これらが間違いなくあると他人からの信頼を得るには，研究途中に客観的な意見を聞くことが重要になります．JSASS では，普段の研究室内での意見交換ではなく，異なる立場の人と率直に意見を言い合うことで，研究の楽しさを感じることができるでしょう．受けた指摘を自分なりによく考え，研究に役立ててほしいと思います．

さて，JSASS は今回で 12 回目になります．ひとまわりということになりますし，継続して行われてきたということからも，ポストプロシーディングに記録としての価値が出てきたかと思います．本ポストプロシーディングに掲載されている研究がどのような時期に行われていたのか，後で読み返したときに思

い出せるよう，2011 年に起こったことを書いておきたいと思います．一番大きな出来事といえば，3 月に発生した東北地方太平洋沖地震でしょう．この巻頭言を書いているのは 2011 年の途中ですが，この地震とこれによって生じた東日本大震災が今年の最大の出来事になることはたしかでしょう．現時点でもこの出来事が日本人のものの考え方に変化をもたらしたという人が現れています．来年度以降も日本人の記憶に残るものとなるでしょう．一方，世界的には，日々金融危機が叫ばれ，財務危機に陥る国が現れています．世界の秩序を支えている通貨への信頼性が揺らいでいるということです．想定していなかったことが次々と起こっています．このような環境で，システムソフトウェアの研究者はどのようにあるべきでしょうか．幸い基盤技術に取り組んでおり，各々が自身の拠り所になる技術を持っているでしょう．どのような環境でも，自らを燈明とすることができるはずです．しかし，我々を取り巻く環境を考えると，自身が軸としているものが揺らぐことも十分に考えられます．このような場合も，確固としたものを持ち，変化を理解し興味深いと感じることで，研究も楽しく行うことができると思います．

最後になりましたが，JSASS の開催において実行委員としてご尽力いただきました名古屋工業大学の齋藤彰一先生，立命館大学の毛利公一先生，横田祐介先生，瀧本栄二先生にこの場を借りて感謝の気持ちを表します．また，JSASS 2011 をスケジュール通りに進行できましたのは，参加者の皆様のご協力によるものです．ここにお礼を申し上げるとともに，今後のご活躍をお祈りいたします．

2011 年 10 月 1 日記

Joint Symposium for Advanced System Software 2011 (JSASS2011)

2011 年 8 月 24 日(水)・25 日(木)

龍谷大学 (滋賀県大津市)

プログラム

■ 8 月 24 日(水)

○ オープニング: 13:30～13:40

○ セッション 1: 13:40～15:40 セキュリティ

○ セッション 2: 15:55～17:55 プロセス・I/O 制御

○ 懇親会: 18:00～20:00

■ 8 月 25 日(木)

Taint 解析を応用したアクセス制御の提案

大石 達也†
立命館大学大学院理工学研究科

1 はじめに

近年，プライバシ情報は電子化され，その情報を計算機で管理がされている．それに伴い，この電子化されたプライバシ情報が所有者の意図に反して情報漏洩をする事件が多発していることが社会問題となっている．文献 [1] では，情報漏洩を引き起こす要因は，正当なアクセス権限を持つ者の管理ミスによる紛失や流出，アプリケーションの誤操作，置き忘れ，盗難が挙げられている．特に，管理ミスや誤操作は，正当な権限を持つ者によって引き起こした人為的なミスであり，それらが高い割合で占めている．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しているため，このような人為的なミスによる情報漏洩を防止することが困難である．

以上の背景により，これまで我々は，正当なアクセス権限を持つ者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 DF-Salvia[2] の開発を行っている．DF-Salvia では, 保護すべきデータを含むファイル（以下, 保護ファイル) に対して, そのデータの保護方針を定義したデータ保護ポリシ（以下，ポリシ）を設定する．ファイルやソケットなどの情報漏洩の可能性がある計算機資源に対する操作を監視し，その操作が行われた時にポリシの内容に応じたアクセス制御を課すことで情報漏洩を防止する．

本稿では，Taint 解析を応用したデータフローに基づくアクセス制御について提案する．提案手法では，プログラム実行時に発生するデータフローを動的に解析し，その解析結果に基づいてアクセス制御を行う．

以降，本稿では，2 章で Taint 解析を応用したアクセス制御の提案手法について述べ，第 3 章では実装方法について述べる．第 4 章で今後の課題を述べ，本稿をまとめる．

2 提案手法

本稿では，Taint 解析を行う機構を備えたハードウェア用いて，動的に解析されるデータフローに基づいてアクセス制御を行う手法について提案をする．提案手法のハードウェアと DF-Salvia が連携する際の特徴を以下に示す．

- ハードウェアは DF-Salvia が保護対象のデータとそのデータを識別するためのタグ（情報）を命令実行時に伝播させることでデータフローの追跡を行う．
- DF-Salvia は情報漏洩のアクセス制御を提供し，保護対象のデータに対するタグをハードウェアに渡しデータフローを追跡をさせる．情報漏洩の可能性がある操作に対して，ハードウェアが追跡したデータフローに基づいてアクセス制御を行う．

2.1 Taint 解析

DF-Salvia がデータフローを基づいてアクセス制御を実現するには，情報漏洩の可能性がある操作が行なわれた際，データがどの保護ファイルから流れてきたかを把握するためにデータフローの情報が必要である．そのため，動的なデータフロー解析である Taint 解析を用いる．

Taint 解析とは，追跡するデータを Taint（汚染）とみなし，そのデータフローを追うことで，そのデータがどのような影響を与えたかを解析する技術である．Taint 解析は，マルウェアの感染による攻撃や脆弱性を持ったプログラムに対する外部からの攻撃に対して検出および制御を行う手法として多く用いられている．Taint 解析を実現する方法としては，追跡対象のデータに対してタグやラベルといった Taint 情報を関連付け，その Taint 情報をもとにプログラム実行時に発生するデータフローを追跡する方法がある．本稿は，このような Taint 解析を行う機構を備えたハードウェアを用いる．

2.2 アクセス制御方式

DF-Salvia では，情報漏洩の可能性がある操作に対してアクセス制御を課すために，特定のシステムコールを監視を行う．Taint 解析を行う機構を備えたハードウェアを用いた DF-Salvia によるアクセス制御の手順を図 1 に示す．図 1 中の番号は，アクセス制御の手順番号を示し，以下にその処理内容を示す．

1. DF-Salvia で保護ファイルのデータを読み込む read システムコールが呼び出されたとき，その保護ファイルに設定されいるポリシをとタグを管理する．
2. 保護対象のデータフローを追跡するため，DF-Salvia がタグをセットする．

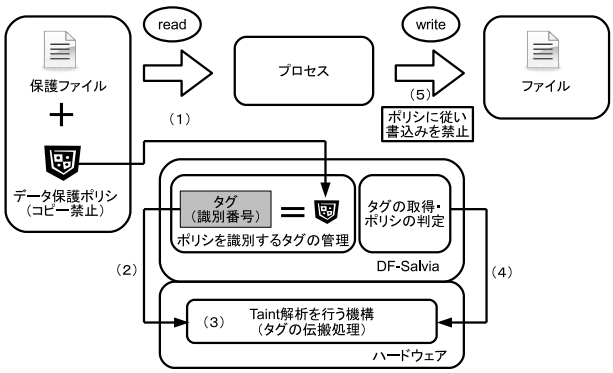


図 1: DF-Salvia によるアクセス制御の手順

3. セットされたタグは OS の演算を行うたびに，ハードウェア上の Taint 解析を行う機構にてタグの有無を確認し，データの伝播規則に従って他のデータに伝播される．

4. 情報漏洩の可能性がある write システムコールが実行された場合，読み込むデータに対して適用されたタグの有無を確認する．

5. タグが登録されている場合は，保護対象のデータが含まれている可能性があるため，タグに関連するポリシに従ってシステムコールの実行の可否を判定しアクセス制御を行う．

DF-Salvia のアクセス制御では，手順１，5 において，プログラムに入力された保護対象のデータとプログラムが出力しようとするデータとの依存関係を把握する必要がある．そのため，DF-Salvia は保護ファイルに設定されいるポリシとタグを関連付けて管理する．タグは，DF-Salvia にてポリシを識別するための任意の番号を割り当てる．保護対象外のデータであればゼロの値とし，ハードウェアがデータフローの追跡を行わないようにする．

3 実装方法

ポリシを識別するタグを伝搬させるために，メモリやレジスタを拡張する必要がある．しかし，それらを拡張したハードウェアを入手することは困難であるため，Taint 解析を用いたソフトエアベースのエミュレータである Argos[3] を用いる．

Argos は，Taint 解析を用いて外部からのゼロデイ攻撃を検知することを目的としたシステムである．Argos では，Taint 解析を実現するのために，メモリやレジスタに対応する Taint 解析用のメモリ領域が用意されている．Argos にはゲスト OS の仮想ネットワークデバイスが受け取ったデータをメモリに書き込む際に，そのデータに対応する Taint 解析用のメモリ領域に Taint 情報を付加する．Taint 情報は仮想プロセッサが命令を実行するたびに，Taint 情報が存在するか確認をし伝播規則に基づいて伝播される．

DF-Salvia において Arogs の Taint 情報の伝播機能をポリシを識別するためのタグの伝搬機構として利用する．なお，Arogs の仮想ネットワークデバイスから受け取るデータに対して Taint 情報を付加する機能や，Taint 情報をもとに攻撃を検出し制御する機能は無効にする．また，メモリやレジスタに対して 1 対 1 に対応する Taint 解析用のメモリ領域を Arogs 内部に用意されているため，ゲスト OS である DF-Salvia からではそのメモリ領域にタグをセットすることや参照することができない．そのため，そのメモリ領域をゲスト OS から参照できるようにするために Arogs はそのメモリ領域を提供する機構を用意し，DF-Salvia がデータに対して適用された Taint 解析用のメモリ領域を特定するための機能を設計する必要がある．

4 おわりに

本稿は，Taint 解析を行う機構を備えたハードウェアを用いて Taint 解析を応用したデータフローに基づくアクセス制御の手法について述べた．今後は，ゲスト OS に対して Taint 用のメモリ領域を提供する機構を Arogs に用意し，DF-Salvia はデータに対して適用された Taint 解析用のメモリ領域を特定する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 2010 年度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/incident/data/2010incident_survey_PIL_v1.4.pdf, 2010.

[2] 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.

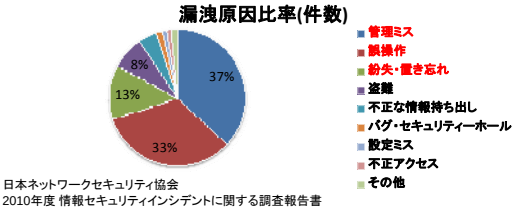
[3] Georgios Porgokalidis, Asia Slowinska, and Herbert Bos: “Argos: An emulator for fingerprinting zero-day attacks,” InProc. ACM SIGOPS EUROSYS’2006, pages 15–27, Leuven, Belgium, April 2006.

Taint解析を応用したアクセス制御の提案

立命館大学院
毛利研究室
M1
大石達也

はじめに

- 近年，機密情報が漏洩する事件が多発
- 情報漏洩の多くが正当な権限を持つ者による人為的ミス
  - 管理ミス：（例）機密情報を含まれた記憶媒体の誤廃棄
  - 誤操作　：（例）電子メールの誤送信
  - 紛失・置き忘れ



日本ネットワークセキュリティ協会
2010年度 情報セキュリティインシデントに関する調査報告書

はじめに（2/2）

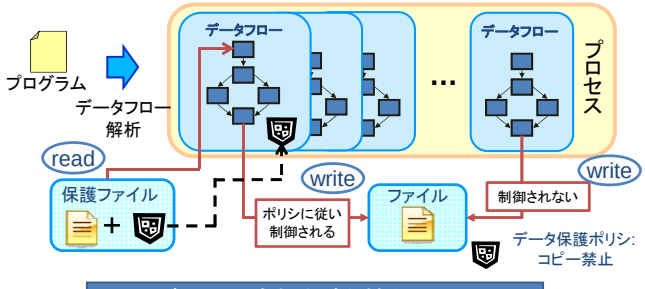
- 情報漏洩の対策
  - 暗号化や認証など
  - 外部からの攻撃を防止することを目的としている
- 正当な権限を持つ者が管理ミスや誤操作などによって引き起す情報漏洩を防止することは困難

人為的ミスを起因とする情報漏洩防止を目的としたDF-Salviaの開発

DF-Salviaの概要

- 人為的ミスによる情報漏洩の防止を目的としたアクセス制御機構を備えたOS
- データフローを対象にアクセス制御を課す
  - 動的にデータの流れを追跡を行う
  - データ入力時から追跡を始めてデータが出力する際にアクセス制御を行う
    - 機密情報の場合は制御を行う
    - そうでないものに対しては制御を行わない

DF-Salviaの動作



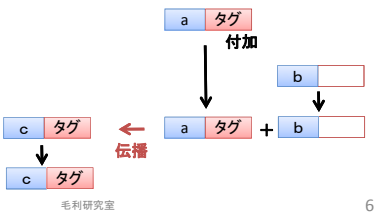
データの流れを追う技術として Taint解析に着目

動的なデータフロー解析

- Taint解析
  - データにタグ付けることで追跡したいデータであるとみなしそのタグを伝播させてデータの流れを追跡する手法

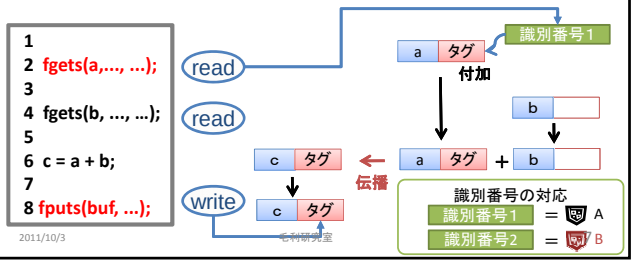
追跡したいデータが変数aに格納される場合

```
1
2 fgets(a,..., ...);
3
4 int b;
5
6 c = a + b;
7
8 fputs(c, ...);
9
```



Taint解析を応用したDF-Salvia（1/2）

- データの流れから書き込まれるデータがどのファイルから来たデータかを特定する必要がある
  - Readシステムコール発行時に，任意の識別番号を発行し，データ保護ポリシと関連付けて対応表を作成し管理を行う
  - writeシステムコールが発行時に，識別番号からデータ保護ポリシを特定しアクセス制御を行う



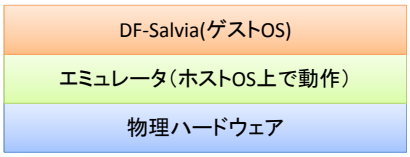
Taint解析を応用したDF-Salvia（2/2）

- 追跡したいデータに対して付加したタグを伝播させる伝播機構が必要
- ハードウェア拡張を行う
  - タグを格納するメモリやレジスタの領域の拡張
  - 命令実行時によるタグの伝播機能の拡張
- しかし，ハードウェアを拡張することは困難

ハードウェアエミュレータを用いて
Taint解析を行う仕組みが必要である

Taint解析を行う ハードウェアエミュレータ

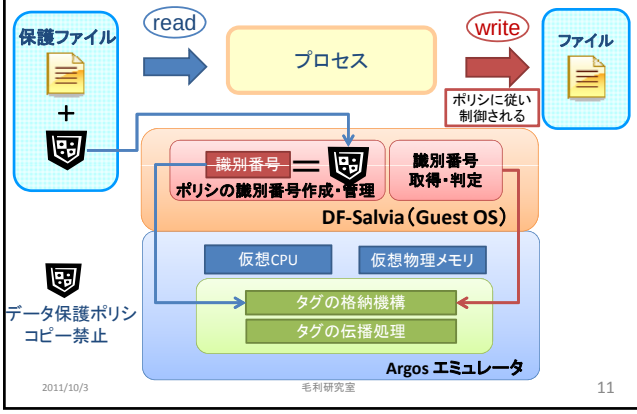
- Argos
  - QEMUを改良したハードウェアエミュレータ
    - 仮想CPUや仮想物理メモリなどを実現
  - 0-day Attack を検知することを目的としたシステム
  - メモリやレジスタの記憶領域に対してタグを付加する機構がある
  - 命令実行時に伝搬規則に従ってタグの伝搬を行う



ArgosとDF-Salviaの連携

- DF-Salvia
  - 保護ポリシとデータフローの管理
  - データフローに基づくアクセス制御を行う
- Argos(ハードウェアエミュレータ)
  - プログラム実行時に発生するデータの伝播を行う
  - データの流れを追跡する

Taint解析を応用したアクセス制御



おわりに

- Taint解析を応用したDF－Salvia
  - 保護データの追跡をタグ（識別番号）を用いる
  - データ保護ポリシと識別番号を関連付けて管理
    - 流れた保護データのポリシを特定できるようにする
    - データ保護ポリシに基づくアクセス制御が可能
- 今後
  - ポリシを識別するためタグの挿入機構の考案

LLVM を用いたアクセス制御のためのデータフロー解析機構

樫山　武浩 †

† 立命館グローバル・イノベーション研究機構

1　はじめに

近年，社会問題化している情報漏洩の多くは，管理ミス，誤操作，盗難・紛失といった正当なアクセス権限を持つユーザの人為的ミスを要因としている．そこで，我々は，これらの情報漏洩を防止するアクセス制御機構を備えたオペレーティングシステム *DF-Salvia*[1] を開発してきた．*DF-Salvia* では，保護対象のファイル (以下，保護ファイル) ごとに，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) として設定することを可能とする．そして，プロセスの動作を監視し，保護データの格納元である保護ファイルに設定されたポリシに従って，プロセスによる保護データの外部への書き出しを制御することを目指している．

DF-Salvia のアクセス制御では，OS 内において，プロセスが保護データを読み込んだのち，その保護データをどのように使用するか，つまりプロセス内のデータフローを把握する仕組みが必要である．これまで，静的解析したデータフロー情報に基づくことで，上記を実現するアクセス制御機構を開発してきた．本アクセス制御機構では，プロセスの実行前にデータフロー解析が完了しているため，アクセス制御におけるオーバヘッドを低く抑えることができる．しかし，変数単位に生成した局所的なデータフロー情報のみを対象としており，プロセス内のデータフローを完全に把握できていない．つまり，保護データの外部への書き出しを制御できない場合がある．

そこで，*DF-Salvia* への適用を目的として，コンパイラと OS を協調させることで，広域なデータフローの解析を実現するデータフロー解析機構について述べる．

2　データフロー解析機構の構想

DF-Salvia がデータフロー情報として使用する定義-使用連鎖は，変数ごとに表現されるデータフローである．そのため，*DF-Salvia* では変数や関数間を跨ぐような広域なデータフローを把握できない．そこで，プログラム中の代入文や関数コールを契機として，個々の定義-使用連鎖を静的に結合することで，広域なデータフローを解析できる．しかし，if 文や for 文等の制御文によりプログラム中に複数の実行経路が存在する場合，アクセス制御に適用すべきポリシを適切に決定できない場合がある．この問題については，文献 [2] において詳細を述べている．

そこで，本稿では，局所的なデータフローとして定義-使用連鎖を静的解析し，データフロー間のデータ伝播を動的解析することで，大局的なデータフローを解析する機構について述べる．本機構により，静的解析によるオーバヘッドの抑制と動的解析によるアクセス制御におけるポリシ決定精度の向上を実現する．

3　コンパイラ側における静的解析機構

3.1　定義-使用連鎖解析

データフロー情報として，定義-使用連鎖を解析する．一般的な最適化では，変数の代入文を定義点とし，その変数を使用する文を使用点とする定義-使用連鎖が使用される．一方，本機構では，代入文に加え，ライブラリ関数による変数への値の代入を考慮して定義-使用連鎖を解析する．なお，OS には，定義点と使用点のうち，read・write システムコールの発行元となるライブラリ関数コールのみを提供する．また，それぞれのライブラリ関数コールについては，リンク時に生成される命令アドレスの情報を含める．本機構の詳細については，文献 [1] を参照されたい．

3.2　コード挿入

実行時において，OS 側に動的解析すべきタイミングを通知するために，プログラム中に以下の 2 つのシステムコールを挿入する．システムコールの挿入例を図 1 に示す．

（1）代入通知システムコール

データフロー間のデータ伝播の発生を OS に通知するシステムコールである．図 1 では，関数 main 内のライブラリ関数 memcpy によって変数 bufA から変数 bufB へのデータ伝播が，関数 main 内の関数 sub の関数コールによって変数 bufB から変数 bufC へのデータ伝播が発生する．前者では，関数 memcpy の直前に代入通知システムコールを挿入し，その命令文を bufA のデータフローの使用点として，bufB のデータフローの

Data flow analysis for the access control based on LLVM

†Takehiro KASHIYAMA

†Ritsumeikan Global Innovation Research Organization, Ritsumeikan University

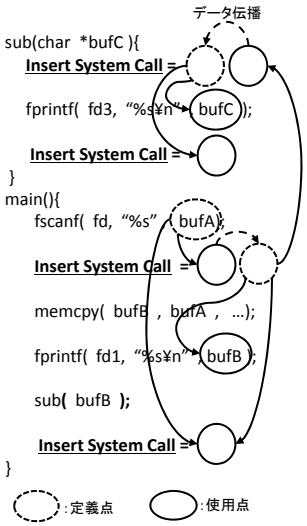


図 1: 代入通知システムコール

定義点として含める．後者では，関数 sub の先頭に代入通知システムコールを挿入し，その命令文を変数 bufB のデータフローの使用点として，変数 bufC のデータフローの定義点として含める．

（2）最終使用点システムコール

データフロー内において，実行経路上の最終の使用点（以降，最終使用点）が実行されたことを OS に通知するシステムコールである．図 1 では，関数の最終行に最終使用点システムコールを挿入し，それを関数内の全てのデータフローの使用点として含めている．

4 OS 側における動的解析機構

静的解析したデータフロー情報とシステムコールの発行に基づいて，広域なデータフローを考慮したアクセス制御の手順を以下に示す．

1. read システムコールが発行されたとき，読み込み対象が保護ファイルなら，その発行元を定義点として含むデータフローに対して，対象保護ファイルのポリシを適用する．
2. 伝播通知システムコールが発行されたとき，その発行元を使用点として含むデータフローに適用されるポリシを，発行元を定義点として含むデータフローに適用する．
3. 最終使用点システムコールが発行されたとき，その発行元を使用点として含むデータフローに適用されるポリシを削除する．

上記の手順により，データフローとポリシの関連づけて管理する．これにより，write システムコールが発行されたとき，その発行元を使用点として含むデータフローを特定することで，アクセス制御に適用すべきポリシを一意に決定することが可能となる．

なお，システムコールの発行元が属するデータフローは，定義点・使用点に含めた命令アドレスと，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たライブラリ関数コールの命令アドレスを比較することで特定する．

5 実装

コンパイラインフラストラクチャとして開発される LLVM（Low Level Virtual Machine）を使用して，本稿で述べたデータフロー解析機構のうち，定義-使用連鎖解析・コード挿入機構を備えたセキュアコンパイラを実装した．

まず，プログラム言語として C 言語を対象とすることから，コンパイラのフロントエンドとして提供される LLVM-GCC を使用して，対象プログラムから LLVM 中間コードを生成する．そして，LLVM 中間コードにおいて，データフロー情報として定義-使用連鎖の解析とプログラム中へのコード挿入を行う．最後に，LLVM のバックエンドを使用して実行ファイルの生成を行う．

なお，コンパイラで静的解析したデータフロー情報は，実行ファイル中のセクションに格納して OS に伝達する．

6 おわりに

本稿では，*DF-Salvia* のためのデータフロー解析機構について述べた．本機構では，静的解析したデータフローのみに依存した場合に比べ，プロセス内に広域なデータフローの監視を実現するとともに，実行時のアクセス制御時におけるポリシの決定精度を向上する．

今後は，データフロー解析機構の実装を行い，既存のアプリケーションに適用することで，アクセス制御におけるポリシ決定の精度とオーバヘッド量について検証する．また，構造体や配列の要素，ポインタを考慮したデータフロー解析について検討する．

参考文献

[1] 井田他：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，情報処理学会全国大会講演論文集，Vol. 3，pp. 511-512，2011.

[2] 樫山他：データフロー情報に基づくアクセス制御のためのプログラム解析，情報処理学会研究報告，Vol. 2011-CSEC-52，No. 45，pp. 1-6，2011.

LLVMを用いたアクセス制御のためのデータフロー解析機構

立命館大学
グローバル・イノベーション研究機構
樫山　武浩

1

はじめに

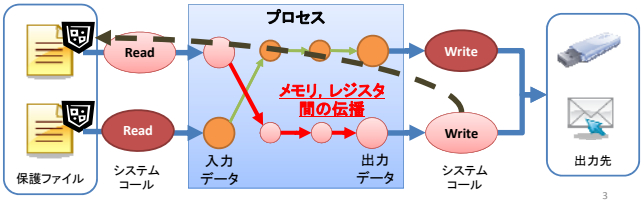
- 情報漏洩の防止を目的としたアクセス制御機構を備えるOS「DF-Salvia」の開発
  - 保護データの管理
    - ファイル単位に，データの使用方針（ポリシ）を設定すること可能とする
  - アクセス制御
    - プロセスによるポリシに違反するデータの使用を禁止する

「だれが」「どこに」だけではなく，
「なにを」に応じたアクセス制御を実現する

2

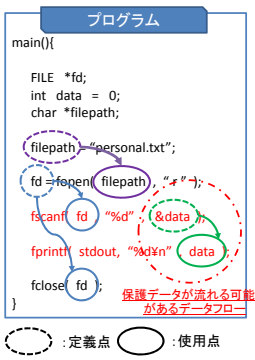
DF-Salviaの実現のために

- 書き出される保護データの格納元ファイル（ポリシ）を特定できる必要がある
- つまり，プロセス内のデータフローを把握することが必要である



3

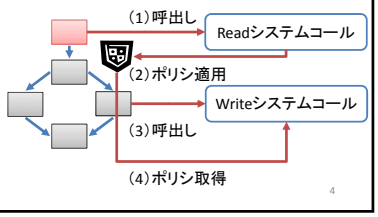
これまでのアプローチ



➢ **セキュアコンパイラ側**

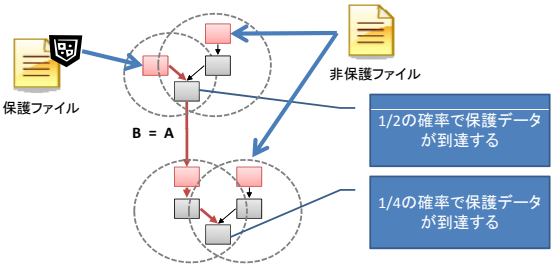
プログラムからデータフローを解析し，実行時にOSに伝達する

➢ **セキュアOS側**



4

静的解析における課題

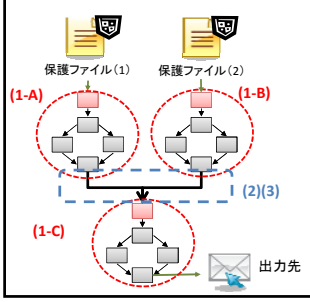


到達確立の低下に伴い，アクセス制御の精度が低下する
⇒アクセス制御に課すべきポリシを適切に決定できない

5

提案するデータフロー解析手法

- 静的解析と動的解析の協調によるアクセス制御のためのデータフロー解析手法



コンパイラ部

(1)ヒント情報解析機構
ポリシの一意に決定可能なデータフローを解析する

(2)コード挿入機構
実行コード中に，データフロー間のデータ伝播をOSに通知するコードを挿入する

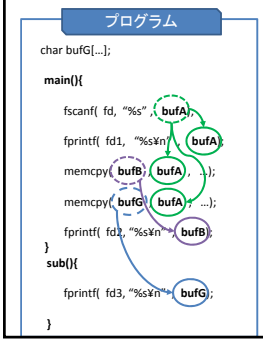
OS部

(3)データ追跡機構
コンパイラ(2)において，実行時情報をもとにデータフローを動的解析する

6

ヒント情報解析機構（セキュアコンパイラ）

- 変数単位の命令文集合を静的解析する



➢ **ヒント情報**
各変数の定義点と使用点の集合

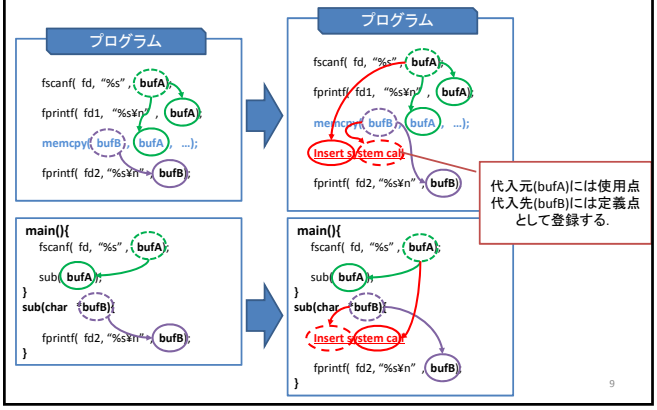
➢定義点
- Readシステムコール発行元となるライブラリ関数
- 変数に値が格納される（代入文，memcpy等のライブラリ関数）

➢使用点
- Writeシステムコール発行元となるライブラリ関数
- ユーザ定義関数コール

コード挿入機構（セキュアコンパイラ）

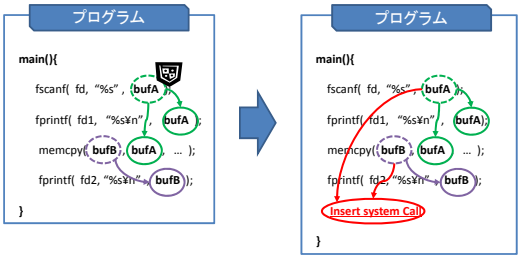
- 静的解析したヒント情報をもとに，データフローを動的解析するためのコードを挿入する
- データ伝播通知コード
  - 各変数間のデータ伝播を動的解析するためのシステムコール
- ポリシ削除通知コード
  - データフローに適用したポリシを削除するためのシステムコール

データ伝播通知コード

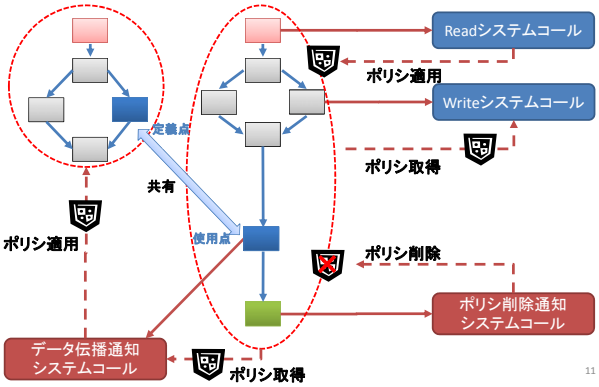


ポリシ削除通知コード

- ローカル変数を対象に，各変数の最終使用点の後にシステムコールを挿入する

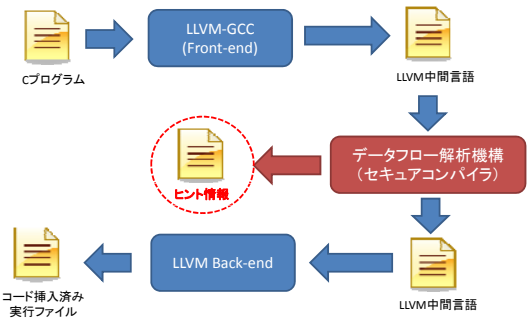


データ追跡機構（OS）



セキュアコンパイラの実装

- LLVM（Low Level Virtual Machine）コンパイラを使用



解析結果

Cプログラム

```
int main(){

  char buf1[MAX_LENGTH];
  char buf2[MAX_LENGTH];

  FILE *fp1 = fopen("input.txt","r");
  FILE *fp2 = fopen("output.txt","a");

  fgets(buf1,MAX_LENGTH,fp1);

  fputs(buf1,fp2);

  memcpy(buf2,buf1,sizeof(char) *
      MAX_LENGTH);
  fputs(buf2,fp2);

  fclose(fp1);
  fclose(fp2);

  return 0;
}
```

LLVM中間言語（処理後）

```
define i32 @main() nounwind {
entry:
  %"alloca point" = bitcast i32 0 to i32
  %1 = call %struct.FILE* @fopen(i8* noalias getelementptr inbounds
  %2 = call i32 asm sideeffect "int $$0x80", "=(ax),0,~{dirflag},~{fpsr
  store %struct.FILE* %1, %struct.FILE** %fp1, align 4
  %3 = call %struct.FILE* @fopen(i8* noalias getelementptr
  %4 = call i32 asm sideeffect "int $$0x80", "=(ax),0,~{dirflag},~{fpsr
  store %struct.FILE* %3, %struct.FILE** %fp2, align 4
  %buf11 = bitcast [100 x i8]* %buf1 to i8*
  %5 = load %struct.FILE** %fp1, align 4
  %6 = call i8* @fgets(i8* noalias %buf11, i32 100, %struct.FILE*
  %buf12 = bitcast [100 x i8]* %buf1 to i8*
  %7 = load %struct.FILE** %fp2, align 4
  %8 = call i32 @fputs(i8* noalias %buf12, %struct.FILE* noalias %7
  %buf23 = bitcast [100 x i8]* %buf2 to i8*
  %buf14 = bitcast [100 x i8]* %buf1 to i8*
  %9 = call i32 asm sideeffect "int $$0x80", "=(ax),0,~{dirflag},~{fpsr
  call void @llvm.memcpy.p0i8.p0i8.i32(i8* %buf23, i8* %buf14, i32
  %10 = load %struct.FILE** %fp1, align 4
  %11 = call i32 @fclose(%struct.FILE* %10) nounwind
  %12 = load %struct.FILE** %fp2, align 4
  %13 = call i32 @fclose(%struct.FILE* %12) nounwind
  %14 = call i32 asm sideeffect "int $$0x80", "=(ax),0,~{dirflag},~{fpsr
%16 = call i32 asm sideeffect "int $$0x80", "=(ax),0,~{dirflag},~{fpsr
return:                                           ; preds = %entry
  %17 = call i32 asm sideeffect "int $$0x80", "=(ax),0,~{dirflag},~{fpsr
  %retval5 = load i32* %retval
  ret i32 %retval5
}
```

ヒント情報

```
Flow :  %buf1 = alloca [100 x i8]
  [2][LST]  %17 = call i32 asm sideeffect "
  [0][DEF]  %6 = call i8* @fgets(i8* noalias
  [1][USE]  %8 = call i32 @fputs(i8* noalias
  [1][USE]  %9 = call i32 asm sideeffect "int
  [1] [USE]call void @llvm.memcpy.p0i8.i3

Flow :  %buf2 = alloca [100 x i8]
  [2][LST]  %17 = call i32 asm sideeffect "int
  [1][USE]  %8 = call i32 @fputs(i8* noalias
  [1][USE]  %9 = call i32 asm sideeffect "int
  [0] [USE]call void @llvm.memcpy.p0i8.i3
```

13

おわりに

- データフロー解析手法の提案
  - 最小単位のデータフローを静的に解析する
    - アクセス制御時にポリシの決定に問題とならない領域
  - 広域なデータフローを動的に解析する
    - データフロー間を跨いだ保護データの流れを追跡
- 今後の予定
  - ポインタにより発生する精度低下への対策
  - データフローの最適化によるオーバヘッドの軽減

14

不変式を利用したセルフヒーリングシステムの提案

白井 宏憲† 齋藤 彰一† 松尾 啓志†
†名古屋工業大学大学院工学研究科

1 はじめに

近年、ゼロデイ攻撃による被害が増加しており未知の攻撃から計算機を守る必要性が増加している．これに対応する物として異常検知型の侵入防止システム (IPS) がある．しかし、IPS では侵入検知後は、管理者への侵入の通知や、サービス停止に止まる．そのため、修正パッチなどを適用し、脆弱性が無くなるまでは安定してサービスを提供できない．そのため、侵入を検知した後、対象プログラムを修復しサービスの実行を継続できるセルフヒーリングシステムが注目されている．本研究では、不変式を利用し、プログラム内で使用される変数等の規則化により、既存システムでは対応できないデータの異常検知・修復も可能なセルフヒーリングシステムを提案する．

2 既存システム

既存システム CrearView[1] は、MemoryFirewall[2] とヒープガードを使用して侵入の検知を行う．MemoryFirewall では不当制御フローを検知でき、本来と違う関数に遷移した場合に異常と検知できる．これらのシステムにより攻撃を検知した後は、不変式を使用してプログラムの修正を行う．

2.1 不変式

不変式とはプログラムの特定の範囲において常に成立する式である．例えば、“変数 x は定数 c1,c2,...cn のどれかを取る” や、“変数 y は変数 x より大きい” などがある．この不変式により、プログラム中で使用される変数の正常な値の範囲が分かり、その範囲内へ変数の値を修正することで、プログラムを正常な動作へ修復できる．

2.2 問題点

CrearView では修正は不変式を使用して行われるが、検知には使用していない．そのため、攻撃の結果としてフローが改竄されたり、ヒープがオーバーフローした場合には、攻撃を検知し、原因となった変数を不変式を使用して修復できる．しかしながら、フローが改竄されず、ヒープもオーバーフローしない場合には攻撃を検知できず、修復機能も働かない．

このような問題があるにも関わらず、不変式を侵入検知に使用しないのは、false positive を減らすためだと考えられる．CrearView では動的解析によって不変式を生成しているため、学習の強度によっては誤検知が多発する．

3 提案システム

本提案システムでは、動的解析だけでなく静的解析も組み合わせて不変式の信頼性を向上させる．これにより、侵入検知にも不変式を使用できるようになる．この結果、フローを変更しないデータのみに対する攻撃も検知し修復できる．

同時に、動的解析では観測しづらい、関数のエラー時の値を記録する．この値は修復時に使用し、修復の精度を高めるのに使用する．

3.1 動的解析部

動的解析部では、CrearView と同様に DR を使用して各命令の実行時の入力オペランド (レジスタやメモリ) の値を観測し記録する．記録した値を daikon を使用して規則性を解析し動的不変式の生成を行う．

3.2 侵入検知部

侵入検知部では、一個以上の静的不変式の違反、もしくは多量の動的不変式違反が検知された場合に攻撃が行われたと判定する．この時、違反する不変式のリストを保存した後に、対象プログラムの再起動を行い、プログラムを正常な状態に戻す．

3.3 修復部

上記の侵入検知部により攻撃が検知され再起動が行われた後に、対象プログラムの脆弱性を修正する．侵入検知部において記録した違反不変式リストの中から修正対象の不変式を選ぶ．そして、該当変数が不変式に違反した場合には不変式を満たすように値を修正するコードを対象プログラム中に挿入する．

修正対象の不変式の選択は、過去の修正結果、攻撃との関連度、違反が発生した時間の三つの基準によって決定する．

まず、以前に修復を行い成功した物を優先的に選ぶ．これは、正しい修正方法を候補の中から発見するため

の学習機能である．この時、過去に成功している修復が無い場合には一意に修正対象の不変式を定められないため、攻撃との関連度を評価する．攻撃を検知した時、常に違反している不変式は攻撃と直接関係がある可能性が高いと判断できる．さらに、最初の攻撃検知時や、関連度が等しい不変式が複数ある場合には、早い時間に違反した不変式を優先的に選ぶ．これは早い時間に発生した違反は後に発生する違反の原因となっている可能性が高いためである．

修復候補の不変式を見つけた後は、不変式を満たすように修正するコードを生成する．そして生成したコードを違反が検知された命令の直前に挿入する．これ以降、同じ攻撃を受けても変数の値は自動的に正常な値に修正され、正常に実行を継続できる．

4 評価

上記の提案システムを Linux 上に DR と daikon を使用して実装し評価を行った．ただし、静的不変式の実装は未完了のため、動的不変式のみで侵入検知・修復を行った．評価項目として検知・修復性能を確認する攻撃評価と、本提案システムの負荷を確かめるオーバーヘッド評価の 2 種類の評価を行った．

4.1 攻撃評価

tinyhttpd に対し擬似的にオーバーフロー脆弱性を発生させ評価を行った．この結果、フローに無関係な変数の一部が書き換えられ、クライアントはページを表示できなくなった．しかし、本提案システム上で実行した所、改竄を検知し正常な値に修復したため、再び正常にページが表示できた．

4.2 オーバーヘッド 評価

apache1.3 と apache bench を同一ＬＡＮ内の別 PC で実行し評価を行った．その結果は表 1 のようになった．ネイティブとあるのは、本提案システムを一切使用せず apache を単体で実行した場合の処理時間である．学習時は、本提案システムで不変式作成の為の変数値を収集している時の処理時間である．なおこの部分は既存システムと同様の実装を行っている．この学習時の処理時間はネイティブ実行に比べ約 2000 倍と大きな負荷がかかっている．しかし、この処理は事前に一度だけ行えばよいため問題は無いと考えている．しかしながら、提案システム (侵入検知処理時) でも 160 倍と比較的大きな負荷がかかっていると分かった．

この原因を解明するため追加評価を行った．この結果を表 2 に示す．DR のみは特別な動作を定義していない空の DR 上で apache を実行した時の実行時間である．本提案システムは DR 上での動作が前提であるため、この値が提案システムの理論上の最高値となる．また、本提案システムの内、検査用コードの挿入のみを省いた評価も行った．この結果、DR のみの実行結果とあまり差が無いと分かった．この事から、本提案システムでは挿入された検査用コードによって殆どのオーバーヘッドが発生していると考えられる．

現在の検査用コード作成部分は、 仮実装中であり、検査の度にコンテキスト切り替えを行っているのが原因と考えられる．

	Time per request[ms]	倍率
ネイティブ	1.183	1
学習時	2361.304	1996.0
提案システム	190.427	160.9

表 1: オーバーヘッド 評価

	Time per request[ms]	倍率
DR のみ	2.042	1.7
提案システム (検査用コードの挿入無し)	2.184	1.8

表 2: オーバーヘッド 追加評価

5 まとめ

不変式を使用し、プログラム内の変数異常を検知し修復できるシステムを提案した．従来のシステムと異なりフローに無関係な変数の異常も検知し修復できる．現在は一部の機能が未実装であり、またオーバーヘッドを考慮せずに実装してある部分がいくつか存在するため実装を進めていく予定である．また、実際の脆弱性を使用しての評価も行っていく．

参考文献

[1] Jeff H. Perkins et al. Automatically patching errors in deployed software. Master's thesis, 2009.

[2] V. et al KIRIANSKY. Secure execution via program shepherding. *In USENIX Security*, 2002.

[3] Derek L. Bruening. Efcient, transparent, and comprehensive runtime code manipulation, 2004.

[4] Michael D. Ernst et al. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.

不変式を利用したセルフヒーリングシステムの提案

名古屋工業大学
白井宏憲

研究背景

- ゼロデイ攻撃による被害が増加
  - 未知の攻撃から計算機を守る必要
- 侵入防止システム
  (IPS:Intrusion Prevention System)
  - 未知の攻撃を検知可能
  - 侵入検知後の動作
    - 管理者に侵入を通知
    - サービスを停止
  - 脆弱性が無くなるまでサービスを安定して提供不可

既存システム

- Self-Healingシステム
  - 異常を検知した後、システムを修復
  - 攻撃されても実行を継続可能
- ClearView※
  - 不当制御フローを攻撃として検知
  - 不変式を利用してシステムの修復が可能

※ J. H. Perkins et al. Automatically patching errors in deployed software. ACM SOSP, 2009.

不変式

- 特定の範囲において”常に成立する式”
- 様々な不変式(変数={x,y}, 定数={a,b,c1,…})
  - 定数＆範囲
    - x ∈ {c1, c2, c3, …., cn}
    - a < x < b
  - 関係(大小, 線形)
    - x < y
    - y == ax + b
  - etc

```
int func (int n){
  int x = 1;
  if( n == 0 )
    x += 2;
  return x;
}
```

x ∈{1, 3}

ClearView動作概要

- 学習時
  - 事前に安全な環境で不変式を作成
- 運用時
  1. 違反する不変式がないか検査
  2. 違反検知時、修復コード挿入
  3. 同じ攻撃は自動的に修復

```
int func(int x[], int size){
  int i;
  for(i=0; i < size; i++){
    if(i >= size) i=size-1;
    x[i] = 0;
  }
  ret(size);
}
```

0 < size < 1024

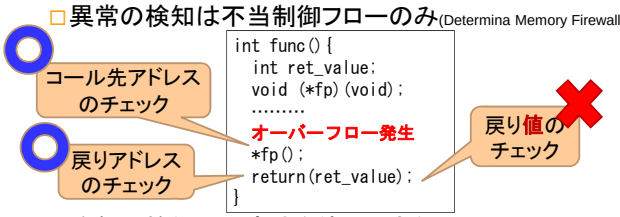
0 <= i < size

改竄を検知！

size == i

ClearViewの問題点

- フローを変更しない攻撃を検知できない
  - 異常の検知は不当制御フローのみ(Determina Memory Firewall※)



  - 攻撃の検知に不変式を使用できない
    - 動的解析による不変式は信頼性が低い
    - 学習不足の可能性を排除できない

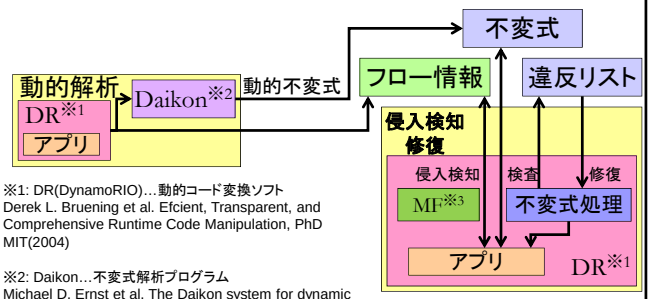
※ KIRIANSKY, V. et al. Secure Execution via Program Shepherding. In USENIX Security (2002).

提案

動的解析だけでなく静的解析も導入する

- フローを変更しない変数の異常も検知可能
  - 侵入検知にも不変式を使用できるようになるため
  - 不変式の信頼性が向上するため
    - 動的解析のみだと、100%不当だと断定できない
    - 静的解析ならば明らかに不当なデータを特定できる
- エラー時の返り値の推測が可能になる
  - 動的解析だとエラー時の値は観測しづらい
  - 修復時に利用し、精度の向上が期待できる

既存システム(ClearView):概要図

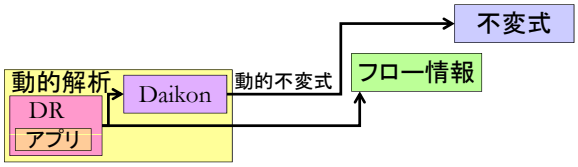


※1: DR(DynamoRIO)…動的コード変換ソフト
Derek L. Bruening et al. Efcient, Transparent, and Comprehensive Runtime Code Manipulation, PhD MIT(2004)

※2: Daikon…不変式解析プログラム
Michael D. Ernst et al. The Daikon system for dynamic detection of likely invariants, MIT, Science of Computer Programming(2007)

※3: Memory Firewall…不当制御フロー検出

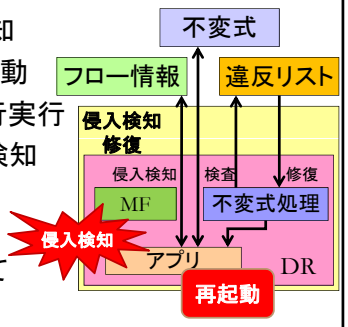
既存システム(ClearView):学習時



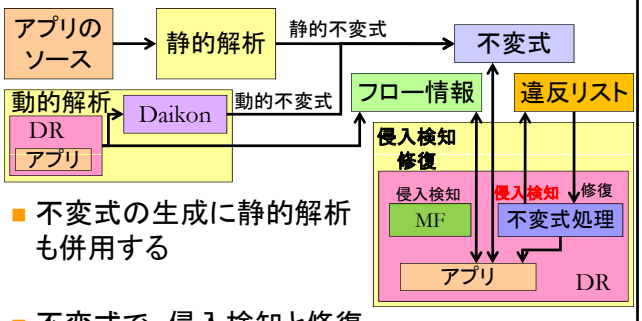
- アプリの動作を解析、規則を作成する
  - フロー情報
    - jump, callの呼び出し元、呼び出し先アドレスのセット等
  - 動的不変式
    - Daikonにレジスタやメモリの値を渡し、作成

既存システム(ClearView):運用時

1. MFを使用し侵入検知
2. 攻撃を検知後、再起動
3. 不変式の検査も並行実行
4. MFにより侵入を再検知
   1. 違反リストを記録
   2. 再起動
5. 違反リストを使用して脆弱性を修正
6. 同じ攻撃に耐性が出来る



提案システム:概要図



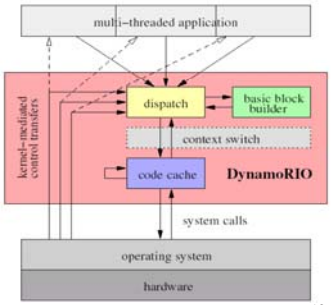
- 不変式の生成に静的解析も併用する
- 不変式で、侵入検知と修復の両方を行う

DR(DynamoRIO)1/2

- 動的コード変換プラットフォーム
- Linux,windows対応
- オープンソース

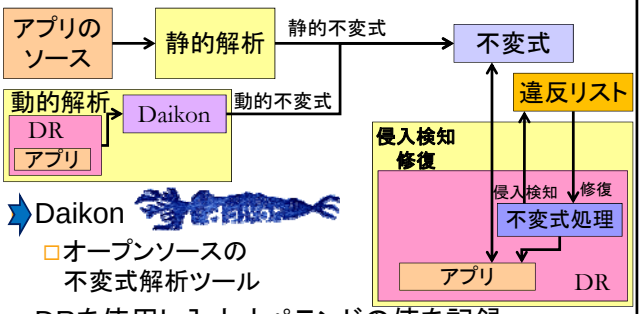
DR(DynamoRIO)2/2

- アプリケーションメモリのコピーを持つ
- プログラム実行中にcode cacheの内容を変更可
- code cache内はネイティブ実行される



13

提案システム:動的解析部



- Daikon
  - オープンソースの不変式解析ツール
- DRを使用し入力オペランドの値を記録
- 記録した値をdaikonで解析し、不変式を作成

14

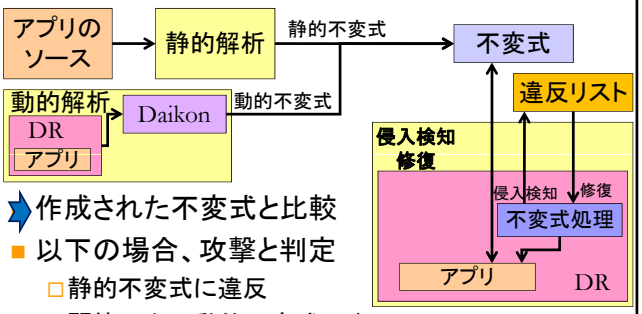
daikon出力結果(apache1.3)

addr　コード	結果1	…	結果98010	生成された不変式
8091f31:add %edi,%edi	src0 = 2 src1 = 2		src0 = 16 src1 = 16	src0 == src1 >= 2 src0 == power of 2
8091f33:mov -16(%ebp),%eax	src0 = 2		src0 = 5	src0 >= 1
8091f36:cmp 16(%ebp),%eax	src0 = 2 src1 = 5		src0 = 5 src1 = 5	src0 >= 1 src1 == 5 src0 <= src1
8091f39:jne 8091b1e	src0 = 8091b1e		src0 = 8091b1e	src0 == 08091b1e
8091f3f:mov %ebx,%eax	src0 = 2		src0 = 6	src0 ∈ { 2, 6 }
8091f41:add $0xc,%esp	src1 = bfdcee40		src1 = bfe8df10	なし

- 98010回の実行値から規則が作成された

15

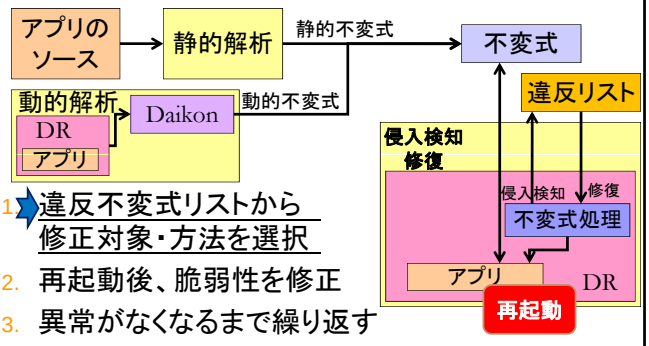
提案システム:侵入検知部



- 作成された不変式と比較
- 以下の場合、攻撃と判定
  - 静的不変式に違反
  - 閾値以上の動的不変式に違反
- 攻撃を検知した際、違反する不変式を保存

16

提案システム:修復部



1. 違反不変式リストから修正対象・方法を選択
2. 再起動後、脆弱性を修正
3. 異常がなくなるまで繰り返す

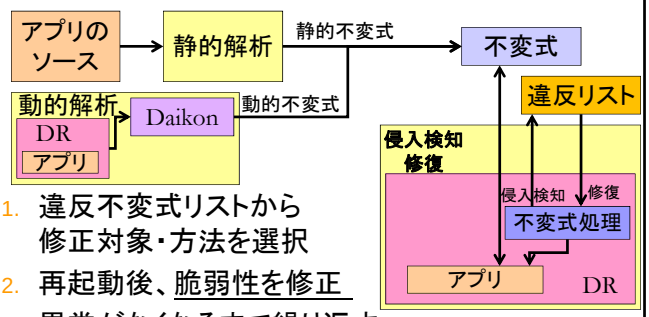
17

修正候補・方法の選定

- 不変式の選択
  - 攻撃との関連度(攻撃検知時の違反率)
  - 最初に違反した不変式
  - 過去に修正を行って成功しているもの
- 不変式が真となるように修正する
  - If(!(x ∈ {c1, c2, c3, …, cn}))　x = c1;
  - If(!(x <= c))　x = c;

18

提案システム：修復部



1. 違反不変式リストから
   修正対象・方法を選択
2. 再起動後、脆弱性を修正
3. 異常がなくなるまで繰り返す

19

修正コードの挿入

- 修正用のコードを違反命令の直前に挿入する
  - 不変式規則: %ecx <= Value
  - If(%ecx > Value) %ecx = Value;

アドレス	オリジナルコード	修正後コード
0x804a33c	sub $0xd8, %esp	sub $0xd8, %esp

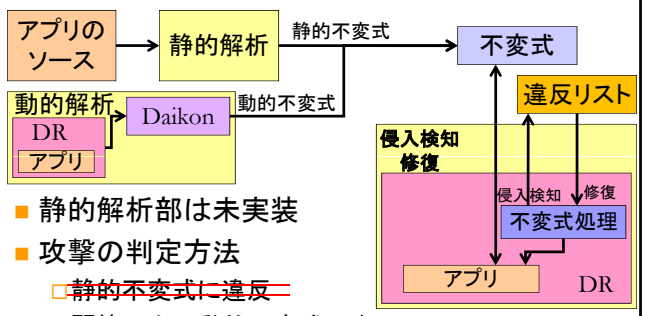
0x804a342	mov **%ecx**, %ebx	cmp %ecx, Value
		cmovg Value, %ecx
		mov **%ecx**, %ebx

0x804a344	mov 4(%ebx), %eax	mov 4(%ebx), %eax

%ecxが改竄されたとする

20

実装状況



- 静的解析部は未実装
- 攻撃の判定方法
  - ~~静的不変式に違反~~
  - 閾値以上の動的不変式に違反
- 動的不変式のみの検知・修復評価を行う

21

評価

- 評価環境
  - CPU: Core2 Duo E4500 2.2GHz
  - MEM: 2GB
  - kernel 2.6.32
  - DynamoRIO 2.2.0-2
- 評価項目
  - 攻撃評価(tinyhttpd-0.1.0)
  - オーバーヘッド評価(apache1.3.42)

22

攻撃評価1/3　脆弱性作成

- tinyhttpdに脆弱性を作り攻撃した

```
void accept_request(){
……
int cgi;
char path[512];
char url[255];
……
sprintf(path, "htdocs%s", url);
if(path[strlen(path) - 1] == '/')
  strcat(path,"index.html");
}
```

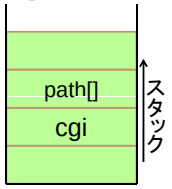
pathの長さを
strlen(url)+15以下に変更すれば
バッファオーバーフロー脆弱性

pathは最大で
"htdocs"+url+"index.html"
つまりstrlen(url)+16

23

攻撃評価2/3　改竄の詳細

- path[]がover flowするとcgiが上書きされる
- int型変数cgi
  - 「CGI」か「通常のページ」かを判別
  - 0:「通常のページの」時(htmlなど)
  - 1:「CGI」の時




```
int cgi = 0
if (method == POST || …)
    cgi = 1;
<path[]のオーバーフロー>
if (!cgi)   serve_file();
else        execute_cgi();
```

cgiを0以外の値に改竄した場合、ページに対しexecute_cgi()を実行させられる

24

攻撃評価3/3　攻撃・修復結果

- 4つの違反を検出

```
error: 8049070@src0: 54 one of 0(int), 1(int)
error: 8049076@src0: 54 one of 0(int), 1(int)
error: 8049086@src0: 54 one of 0(int), 1(int)
error: 804908c@src0: 54 one of 0(int), 1(int)
```

cgi ∈ {0,1}
に対する違反

- 最初の違反の8049070のsrc0を修正
  - 8049070@src(cgi)を0に修正
- 違反は無くなり、正常な動作を継続できた

25

オーバーヘッド評価

- apache1.3 + apache bench (LAN内別PC)

	Time per request [ms]	倍率
ネイティブ	1.183	1
学習時(不変式生成時)	2361.304	1996.0
提案システム	190.427	160.9

追加評価	Time per request [ms]	倍率
DRのみ	2.042	1.7
提案システム (検査用コードの挿入無)	2.184	1.8

26

オーバーヘッド評価考察

- 不変式はDR上のメモリにロードされている
- 検査毎にDRのメモリ空間にコンテキスト切り替えを行っている(ほぼ命令毎に)
- 違反時以外は切換え無しで検査する

現在のコード
0x804a342:
　コンテキスト切り替え
　call DR_func()
　コンテキスト切り替え
　mov %ecx, %ebx

改良予定案
0x804a342:
　cmp %ecx, Value
　jle 検査対象命令
　切替&DR_func()&切替
　mov %ecx, %ebx

27

まとめ

- 静的解析を組合せ、攻撃検知に不変式を用いるシステムを提案した
- フローを改変しない変数異常も検知・修復できる

今後の課題

- 未実装部分の実装を行う
  - 静的解析によって不変式を作成する
  - 静的と動的解析で作成した不変式を結合する
- オーバーヘッドの削減

28

クロスド メイン SSO 基板 Shibboleth におけるユーザ権限委譲方式の提案

†秋山 晋　　†齋藤 彰一
†名古屋工業大学大学院工学研究科

1　はじめに

現在, チケット予約から文章作成まで, さまざまなサービスがオンライン上で提供される中, 会社や企業といった組織間で互いに協調し合って作業に臨む機会が増えてきている.

しかし, こういった協調作業は参加者を増やすことで効率化が図れるものの, Web サービスの利用者側には, 他者へ簡単に利用権限を与える手段がない, という問題点が存在する. この要因として,Web サービスの利用権限の与奪がサーバ管理者のみが行なうという認証基板の基本構造が挙げられる. これにより, ユーザが他者へ利用権限を与える場合, サーバ管理者へいちいち問い合わせる手間が生じる. サーバ管理者への申請を省く手段として, 自身の ID やパスワードといったログイン情報を他者に教える方法があるが, これはセキュリティや個人情報の観点から好ましくないのは明らかである. よって, 協調作業の効率化を目標とした, 現行の認証機構への権限委譲機能の追加の必要性がでてくる.

そこで私は, 現在世界中で導入され, 主流となりつつあるシングルサインオン認証の技術基盤である Shibboleth 環境下で, 組織間における権限委譲方式を提案する.

以降, 第 2 章ではシングルサインオン認証基板 Shibboleth についての説明, 第 3 章で本提案手法, 第 4 章でセキュリティの考察を述べ, 第 5 章で本稿をまとめる.

2　関連研究

2.1　シングルサインオン：SSO

シングルサインオン（SSO）とは, ある認証機構に対してユーザが一度認証を受けるだけで, その認証を認可するサービスをすべて受けることが出来るようにある技術である.

2.2　クロスド メイン SSO 認証基盤 Shibboleth

本手法の技術基盤である Shibboleth について説明する.Shibboleth は以下の 3 つの要素が認証の要となる.

- ServiceProvider（SP）　: apache のモジュールとして動き,Web サービスを保護する.
- IdentityProvider（IdP）:認証を行なう. 認証後,SP へ認証アサーション, ユーザ属性を送付する.
- DiscoveryServer（DS　）:認証に参加する組織すべての IdP の一覧を保持している.

Shibboleth 認証では,Web リソースを保護する SP に認証要求されたユーザは,DS へリダイレクトされ, DS の提示する一覧から選択した IdP で認証を行なう. IdP で行なわれた認証情報は, 一緒に発行されるユーザのアサーション（表明）とユーザ属性と共に SP に渡され, SP はそれらを元にアクセス制御を行なう. これが Shibboleth による認証の概要である.

Shibboleth では, 各組織が互いの SP や IdP の情報をメタデータとして共有しあうことで, クロスド メイン SSO を実現している. これにより, 例えばある大学の Web サービスを, 別の大学の学生が自身の大学の IdP で認証を行なうことでサービスが利用可能とすることが出来る.

しかし, 前章で述べたように Shibboleth には権限委譲機能が実装されておらず, 例えば他大学の友人に無線 LAN を利用させてあげたい場合, アクセス権限を持たない友人は認証後,SP によってアクセス制御されてしまう. サーバ管理者への申請は, 即時に許可が降りないので, Web サービス利用の即時性が失われる可能性があるのは, 前章で述べた通りである. この問題を解消するために, 私は Shibboleth への権限委譲機能の実装を提案する.

3　提案

本章では既存の SSO 認証技術である Shibboleth に対する権限委譲機能付加の手法を述べる.

3.1　基本的なアイディア

クロスド メイン SSO 認証基板である Shibboleth にはユーザ間で権限委譲を行なう機能がない. そこで, 本手法では権限委譲行為に対する当事者の意思確認と本人確認を行なうことのできる機能を,Web サービス（Box）という形で提供する. これら 2 つの確認を行なうことにより,「ユーザが権限委譲を確かに行う」という本人性を確認することが可能となり, 権限委譲が実現する.

3.2 権限委譲サービス:Box

当事者同士の本人性を確認するWebサービス, つまり権限委譲サービスを提供するBoxは,2つの機能を備えている. 1つ目は上記の通り, 権限委譲元・先双方の本人確認と意思確認を行なう機能である. 2つ目は,1つ目の機能である本人性の確認を行なった後,IdPに対して権限委譲元の認証アサーションを権限委譲先へするためのアサーション発行要請機能である. これに併せて,IdPに追加実装を行なう必要がある. それは,Boxからの発行要請を認可した場合にのみ, 権限委譲元のアサーションを委譲先へ送付する機能である. これにより, 本手法は当事者双方の本人性の確認, アサーションの発行というプロセスを経て権限委譲を実現する.

3.3 権限委譲の流れ

説明のため権限委譲元をA, 権限委譲先をBとし,AがBへあるWebサービスへのアクセス権限を委譲する場合を考える. 本提案手法では, 権限委譲元と権限委譲先が以下の手順を踏むことで権限委譲が行なわれる.

3.3.1 権限委譲元Aの操作

- Aが Boxへアクセスする
- Boxは, ユーザに, 権限委譲先の情報（ID, 所属組織名等）を要求する
- Aは, 委譲先であるBの情報を入力する
- Boxは, 入力情報とAのユーザ属性をセットで保管する

以上で委譲元の操作は終了である. 次に委譲先ユーザの操作を説明する.

3.3.2 権限委譲先Bの操作

- Bは利用したWebサービスへアクセスする
- SPによってBはBoxへリダイレクトされる
- Boxが権限委譲元の情報を要求する
- Bは, 委譲元であるAの情報を入力する
- Boxは保管しておいた情報とBの入力した情報を照合し, 真であればIdPへアサーション発行要請を行なう
- 発行要請を受け取ったIdPは,BをAの認証アサーションと共にSPへリダイレクト
- SPはBをAと認識したままレスポンスを返す

Boxでは,Bのアサーションと A操作時に保管したBの情報を,Aのアサーションと Bの入力した Aの情報をそれぞれ照合することで,A,B双方の本人性を確認している. 双方の本人性の確認がとれた場合にのみIdPへアサーション発行要請を送ることで, 既存技術への追加という形で実装が可能となる.

4 考察

本章では提案手法による権限委譲機能のセキュリティについて考察する.

今回は権限委譲元Aが権限委譲先Bを誤入力した場合と,Bへの成り済ましによる委譲権限の奪取について考える.

前者については, 誤った指定先のユーザがAを権限委譲元であると主張しつづけない限り護送は有り得ないと言える. なぜなら, 本手法では権限委譲機能には委譲先へ権限委譲を通知しないので, 誤ってB以外を指定しまった場合にそのユーザが気づく可能性は低いからである.

後者については,Webサービスの利用＝ユーザの認証済みであるので成り済ましは不可能であると言える. Shibbolethでは保護されたWebサービスへのアクセスはすべて,IdPによる認証と,IdPから発行されるアサーションを認可するSPを通過する必要があるからである.

5 まとめ

本稿では, クロスドメインSSO認証基盤であるShibboleth環境下でユーザ間権限委譲を行なえる機能について述べた. 本手法は, 権限委譲元・先双方の本人性が確認できるWebサービスを提供することで権限委譲を可能とした. 今後は,Boxの機能に応じたIdP,SPの実装を進める予定である.

参考文献

[1] 金西計英 松浦健二 三好康夫, 大学間Webサービス連携のためのShibbolethを用いた認可管理機能の実現, 日本教育工学会 2008-12-20.

[2] 内藤久資 梶田将司 小尻智子, 大学における統一認証基盤としてのCASとその拡張, 情報処理学会 Apr.2006.

クロスドメインSSO基盤Shibbolethにおけるユーザ権限委譲方式

名古屋工業大学　齋藤彰一 研究室
秋山 晋

1

研究背景

- Web上で組織間の協調作業が増えてきている
  - ex1.ドキュメントの作成・編集
  - ex2.Web教材の利用
- Webサービス利用権限を共同作業者へ簡単に与える手段がない
  - ✓ 権限の与奪はWebサーバ管理者にしか行えない
  - ✓ ID・パスワードを他人に教えたくない

多数の組織に属するユーザ同士で利用権限を一時的に委譲できる仕組みが必要

2

研究背景 － 広域認証基盤 －

Shibboleth.

- Internet2(米)が2000年に発足したプロジェクト
- 組織間での認証基盤システム
  - 日本では大学間認証連携としてGakuNin（学認）が稼働中
- 各組織が提供するWebサービスを利用可能
  - Ex. 他大学の無線LAN、e-Learningの利用
- Single Sign-On(SSO)の実現
  - SSO:初回認証後、別サービスで認証工程を省略する技術

3

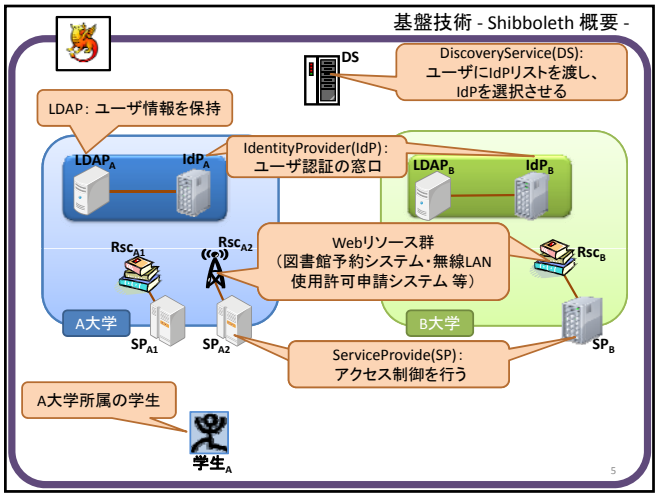
組織間認証基盤における権限委譲の提案

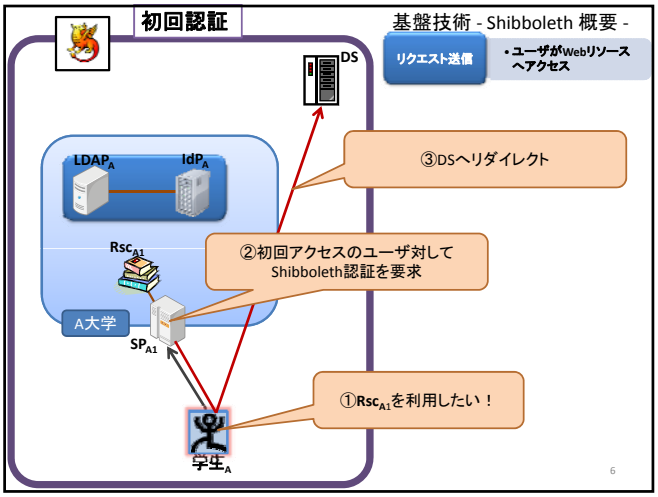
組織間認証基盤：Shibboleth
＋
権限委譲機能の必要性

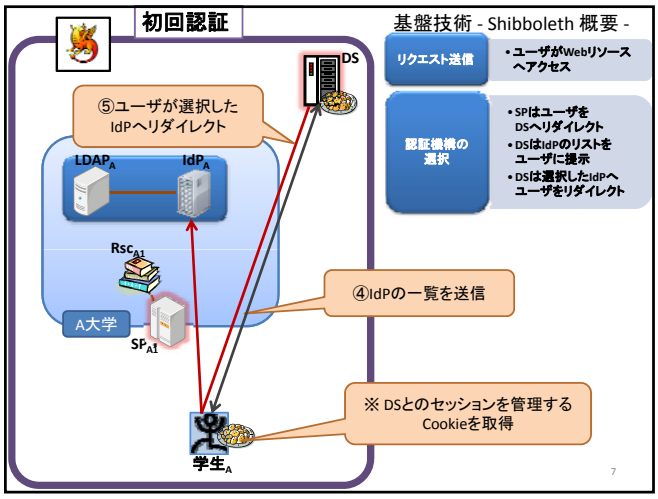
提案

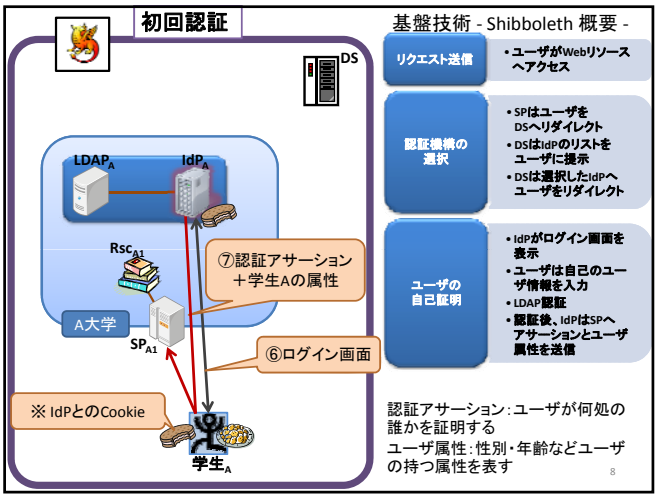
多数の組織に属するユーザ同士で、一時的にWebサービス利用権限を委譲できる認証システム

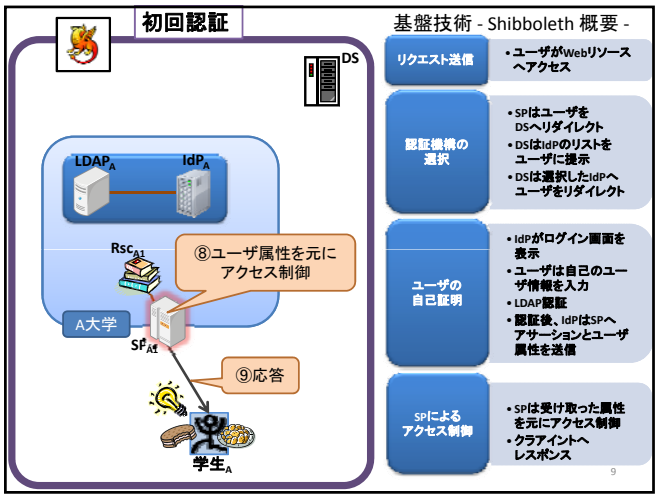
4

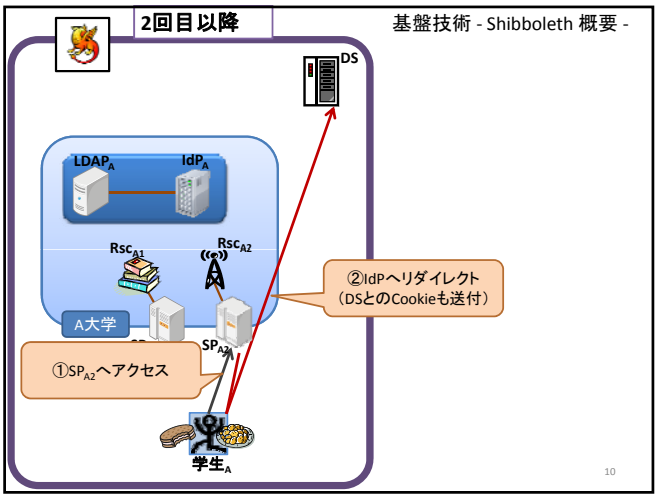


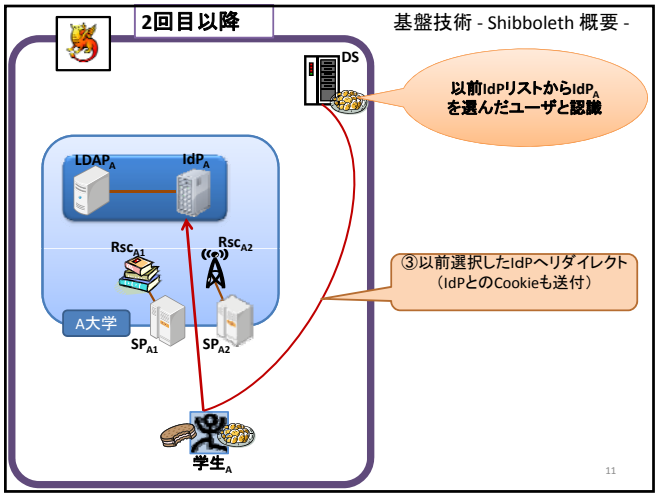


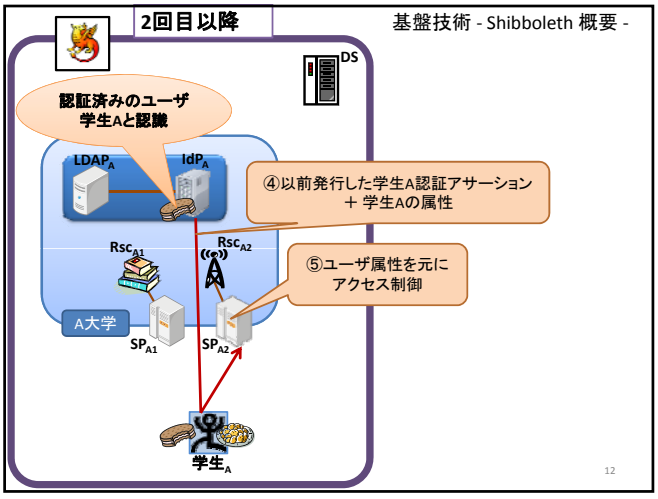

初回認証
基盤技術 - Shibboleth 概要 -
DS
⑤ユーザが選択した
IdPへリダイレクト
LDAP$_A$
IdP$_A$
Rsc$_{A1}$
A大学
SP$_{A1}$
④IdPの一覧を送信
※ DSとのセッションを管理する
Cookieを取得
学生$_A$
7
リクエスト送信
•ユーザがWebリソース
へアクセス
認証機構の
選択
•SPはユーザを
DSへリダイレクト
•DSはIdPのリストを
ユーザに提示
•DSは選択したIdPへ
ユーザをリダイレクト


初回認証
基盤技術 - Shibboleth 概要 -
DS
LDAP$_A$
IdP$_A$
Rsc$_{A1}$
A大学
SP$_{A1}$
⑦認証アサーション
＋学生Aの属性
⑥ログイン画面
※ IdPとのCookie
学生$_A$
認証アサーション：ユーザが何処の
誰かを証明する
ユーザ属性：性別・年齢などユーザ
の持つ属性を表す
8
リクエスト送信
•ユーザがWebリソース
へアクセス
認証機構の
選択
•SPはユーザを
DSへリダイレクト
•DSはIdPのリストを
ユーザに提示
•DSは選択したIdPへ
ユーザをリダイレクト
ユーザの
自己証明
•IdPがログイン画面を
表示
•ユーザは自己のユー
ザ情報を入力
•LDAP認証
•認証後、IdPはSPへ
アサーションとユーザ
属性を送信


初回認証
基盤技術 - Shibboleth 概要 -
DS
LDAP$_A$
IdP$_A$
Rsc$_{A1}$
⑧ユーザ属性を元に
アクセス制御
A大学
SP$_{A1}$
⑨応答
学生$_A$
9
リクエスト送信
•ユーザがWebリソース
へアクセス
認証機構の
選択
•SPはユーザを
DSへリダイレクト
•DSはIdPのリストを
ユーザに提示
•DSは選択したIdPへ
ユーザをリダイレクト
ユーザの
自己証明
•IdPがログイン画面を
表示
•ユーザは自己のユー
ザ情報を入力
•LDAP認証
•認証後、IdPはSPへ
アサーションとユーザ
属性を送信
SPによる
アクセス制御
•SPは受け取った属性
を元にアクセス制御
•クライアントへ
レスポンス


2回目以降
基盤技術 - Shibboleth 概要 -
DS
LDAP$_A$
IdP$_A$
Rsc$_{A1}$
Rsc$_{A2}$
②IdPへリダイレクト
（DSとのCookieも送付）
A大学
SP$_{A2}$
①SP$_{A2}$へアクセス
学生$_A$
10


2回目以降
基盤技術 - Shibboleth 概要 -
DS
以前IdPリストからIdP$_A$
を選んだユーザと認識
LDAP$_A$
IdP$_A$
Rsc$_{A1}$
Rsc$_{A2}$
③以前選択したIdPへリダイレクト
（IdPとのCookieも送付）
A大学
SP$_{A1}$
SP$_{A2}$
学生$_A$
11


2回目以降
基盤技術 - Shibboleth 概要 -
DS
認証済みのユーザ
学生Aと認識
LDAP$_A$
IdP$_A$
④以前発行した学生A認証アサーション
＋ 学生Aの属性
Rsc$_{A1}$
Rsc$_{A2}$
⑤ユーザ属性を元に
アクセス制御
A大学
SP$_{A1}$
SP$_{A2}$
学生$_A$
12

提案手法

- 権限委譲機能がない
  - Webサーバ(SP)管理者に申請が必要
  - 自身のログイン情報を他人に曝す
- 委譲行為に対する当事者の意思・本人確認が行える枠組を提供することで権限委譲を実現

組織間認証基盤であるShibboleth環境下で、「権限委譲」をサービスとして提供するWebサービスを提案

13

権限委譲サービス:Box

- Boxの機能:
  - 権限委譲元・先、双方の本人確認・意思確認を行う
  - 上記確認後に限り、IdPにアサーション発行を要請できる
- IdPの追加機能:
  - Boxからの要請に応じ、アサーションの発行を行う

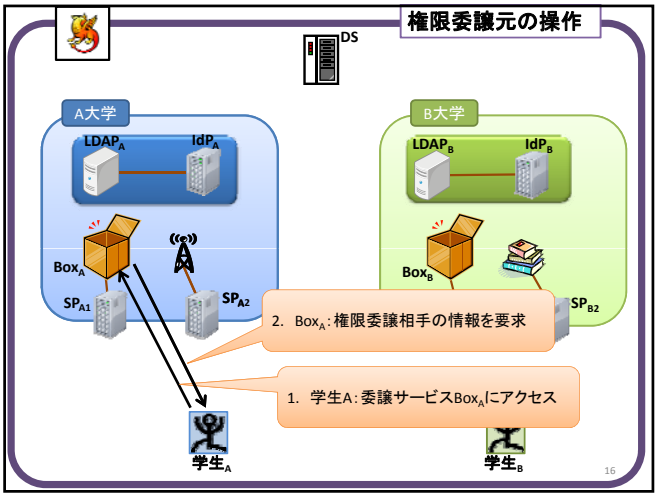
BoxはIdPへ発行要請し、IdPが委譲元のアサーションを委譲先に送ることで権限委譲が実現できる

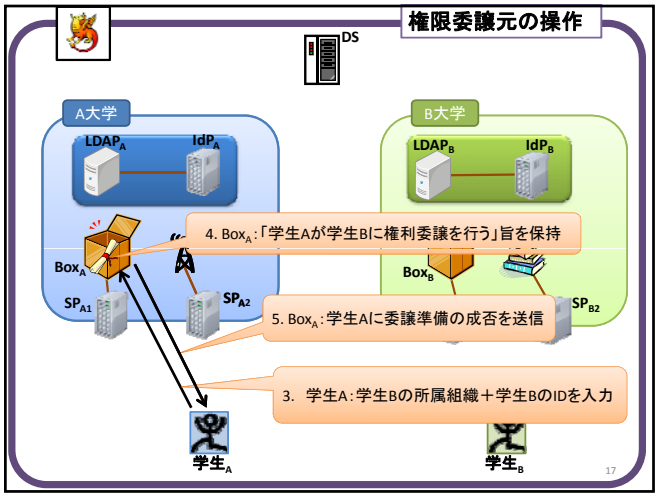
14

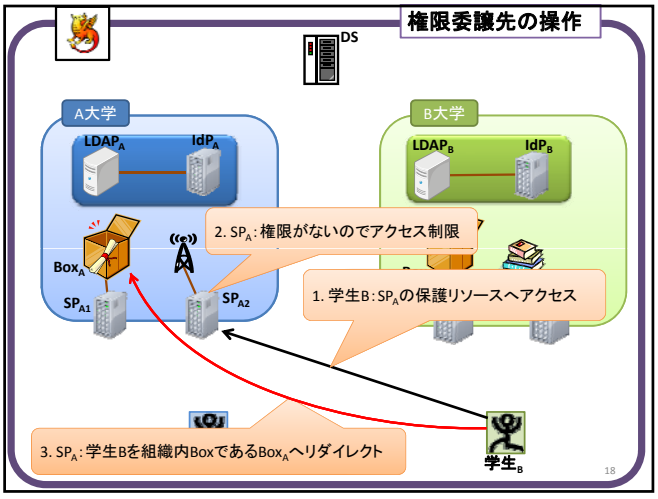
権限委譲操作 手順

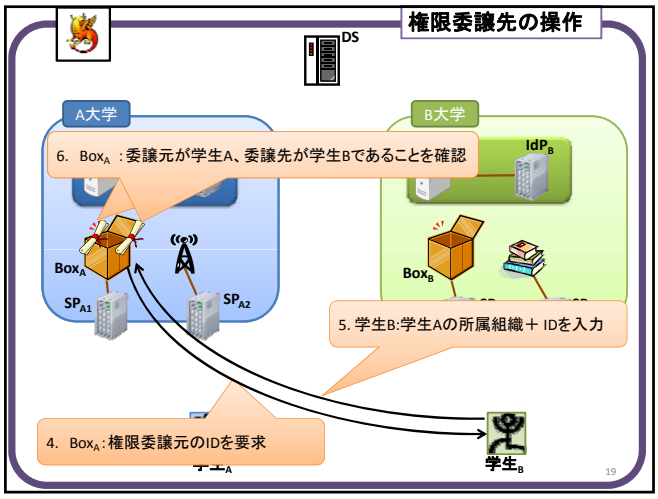
- 学生Aが学生Bに対して、A大学Webリソースへのアクセス権限を委譲する方法を考える
  - 学生BはA大学Webリソースへのアクセス権限がない
- 権限委譲操作
  - ① 権限委譲準備[学生A]:Box内で委譲元(学生A)が委譲先(学生B)を指名
  - ② 委譲権限利用[学生B]:Box内で委譲先(学生B)が委譲元(学生A)を指名

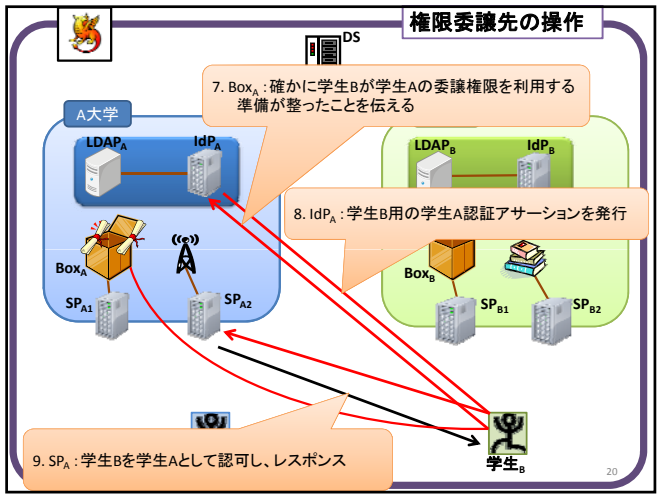
※以降、学生A・学生Bは認証済み、Boxとセッションを保持しているとする











考察：セキュリティ

- 学生A（権限委譲元）の誤入力
  - 本機能は委譲を通知しないので、誤って指定した相手が権限を誤送されても気づく可能性は低い
- 学生B（権限委譲先）へのなり済まし
  - 委譲操作時、学生BはIdPで認証を済ませているので成り済ますことはできない

21

考察：実装

- Boxの実装
  - 意思確認を行うための情報を保持する機能
  - 確認後、IdPへアサーション発行を要請する機能
- SP
  - アクセス制限後、ユーザをBoxへリダイレクトする機能
- IdP
  - 要請に応じてアサーションを発行する機能
- DS・Webサービス
  - 実装事項なし

22

まとめと今後の課題

- Shibboleth環境における権限委譲システムを提案
  - ログイン情報が漏洩するリスクの回避
  - 協調作業への参加の即時性を確保
  - SP単位で権限が委譲可能
- 今後の課題
  - BoxをShibbolethへ実装

23

関数のリターンアドレス保護によるスタック偽装攻撃検知

冨永 悠生 †　　樫山 武浩 ‡　　瀧本 栄二 ‡　　桑原 寛明 ‡　　毛利 公一 ‡　　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によるプログラムの脆弱性を突いた攻撃は，システムの乗っ取りや情報漏洩などを引き起こす．脆弱性を突いた攻撃には既知の攻撃と未知の攻撃が存在する．未知の攻撃を防ぐことを目的とした侵入検知システム [1][3] の研究は多くの成果を上げている．

多くの侵入検知システムは，プログラムの挙動を管理者が規制し，アクセス制御に基づいた挙動のみを許可する．許容範囲を超えた攻撃が行われた場合，異常を検知し攻撃を防ぐことができる．

従来の侵入検知システムを回避する攻撃として，バッファオーバフローから派生するスタック偽装攻撃（Mimicry Attack）[2] が存在する．スタック偽装攻撃は，ユーザスタックをプログラムの正常な挙動としてあり得る別状態に偽装する攻撃である．多くの侵入検知システムでは，正常なスタックか偽装されたスタックかを判断することができないため検知できない問題がある．

本研究では，スタック偽装攻撃を検知する手法を提案する．

2　スタック偽装攻撃

バッファオーバフローの脆弱性が存在するプログラムコードを図 1 に示す．この例の場合，バッファオーバフローは，プログラムが確保したメモリサイズを超えた書き込みによって発生する．図 1 のプログラムでは，char 型の配列 buf は 32 バイトのメモリ領域を持つ．関数 strcpy の引数に 32 バイト以上を与えた場合，配列 buf の領域を越えた書き込みが発生するためバッファオーバフローが発生する．

バッファオーバフロー攻撃から派生する攻撃として，コードインジェクション攻撃や Return-into-libc 攻撃がある．これらの攻撃はバッファオーバフローを利用し，関数ごとに積み上げられるスタックフレームのリターンアドレスを書き換える．リターンアドレスを書き換えることにより関数から復帰するさい任意のアドレスから実行を再開させ，攻撃コードの実行を可能とする．

```
1  int main(int argc,char **argv ){
2    char buf[32];
3    ...
4    strcpy(buf,argv[1]);
5    ...
6  }
```

図 1: バッファオーバフロー脆弱性の例

Mimicry Attack Detection By Protected Return Address Of Function
Yuuki Tominaga†,Takehiro Kashiyama‡and Eiji Takimoto‡and Hiroaki Kuwabara‡and Kouichi Mouri‡and Yoshitoshi Kunieda‡
†Graduate School of Science and Engineering, Ritsumeikan Univ
‡College of Information Science and Engineering, Ritsumeikan Univ

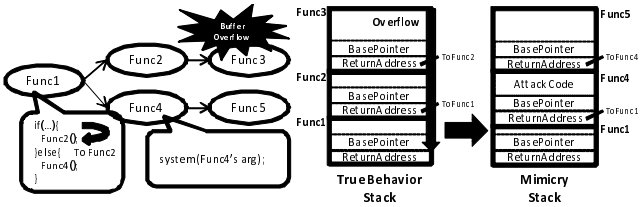


図 2: スタック偽装攻撃例

2.1　スタック偽装攻撃

スタック偽装攻撃は，バッファオーバフローにより現在のスタックの状態を別の状態に偽装する．スタック偽装攻撃の例を図 2 に示す．図 2 は，関数のコールグラフとスタック偽装攻撃を受ける前のスタックと受けた後のスタックを示している．Func1 → Func2 → Func3 と関数呼び出しされた際のスタックを図中の True Behavior Stack に示す．

関数 Func3 でバッファオーバフローを起こされ，True Behavior Stack が偽装された状態を Mimicry Stack に示す．Mimicry Stack では，Func1 → Func2 → Func3 と関数呼び出しされていた True Behavior Stack の状態を Func1 → Func4 → Func5 と関数呼び出しされた際のスタック状態に偽装している．この状態はスタックは偽装されているがプログラムの正常な挙動としてあり得るため，静的解析に基づき取り得る関数の戻りアドレスを検査する侵入検知システムでは検知できない．

3　スタック偽装攻撃の検知手法

「関数呼び出しが行われるとスタックにスタックフレームが積まれる」こと「リターンアドレスを動的に書き換える動作がない」ことの 2 つの条件が成り立つ一般のプログラムでは，関数の出口でリターンアドレスを検査することによりスタック偽装攻撃を検知できる．

提案手法の適応例を図 3 に示す．本手法では，各ユーザ関数の入り口と出口でリターンアドレスを保護とリターンアドレスの検査をすることによりスタック偽装攻撃の検知を行う．本手法は，ユーザ関数におけるスタック偽装攻撃の検知を対象としており，ライブラリ内では発生しないことを前提としている．

リターンアドレス保護の動作概要を図 5 に示す．関数入り口点のリターンアドレスの保護では，最上位のスタックフレームのリターンアドレスの値をコピーしカーネル空間に存在する保護スタックに積み上げる．関数出口点のリターンアドレスの検査では，全スタックフレームのリターンアドレス順を求め，保護スタックに積み上げられている値と全て比較する．バッファオーバフロー攻撃によりリターンアドレスが書き換えられた場合，全スタックフレームのリターンアドレスと保護スタックに積み上げられている値を比較したさいに値が一致しなくなる．この不一致を検出することにより，リターンアドレスが書き換えられたことが検知で

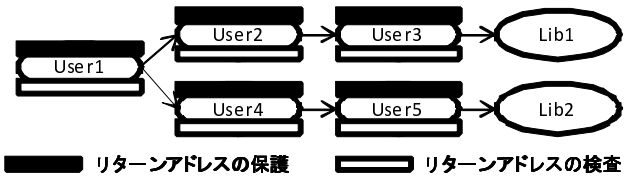


図 3: 提案手法の適応例

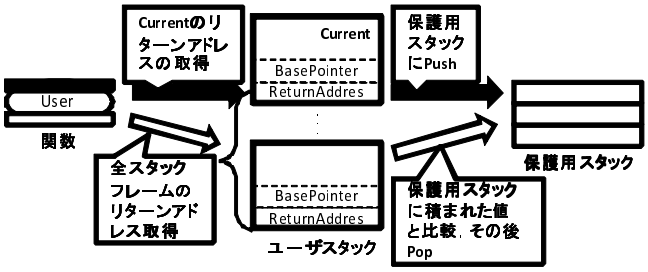


図 4: リターンアドレス保護の動作概要

きスタック偽装攻撃を防ぐことができる．

リターンアドレスの保護や検査は，コンパイラによるコード埋め込みにより実現する．図 5 はソースコードから実行ファイルができるまでの流れを表している．コンパイラを用いることで既存のソースコードに対しても検知コードの埋め込みが容易に行える．

4 実験

本提案手法を実装し，異常検知テストと正常動作の確認とオーバヘッド測定の 3 つの実験を行った．

異常検知テストでは，スタック偽装攻撃とリターンアドレスを書き換える攻撃を検知できるか確認した．図 2 のようなスタック偽装攻撃を行った結果，攻撃検知できることを確認した．また，リターンアドレスを書き換える種々の攻撃（表 1）に対しても検知の確認を行い，すべての攻撃の検知を確認した．

正常動作の確認では表 2 のプログラムを動作させ，誤検知なく動作するか確認した．

オーバヘッド測定では，gzip と tar に対するオーバヘッド測定（表 3）と関数呼び出しの深さとオーバヘッドの増加量（図 6）について調査した．表 3 の関数呼び出しの平均深度は，スタックフレームの深さ（関数呼び出しされた深さ）の平均値を表している．本手法を適応後の gzip におけるオーバヘッドは 3.83%と小さな値となっている．しかし，tar におけるオーバヘッドは 58.96%と大きくなった．図 6 は関数呼び出しの深度とオーバヘッドの関係を表している．tar におけるオーバヘッドが大きい理由としては，オーバヘッドが関数呼び出し数と関数呼び出しの深度に依存するためである．そのため，tar の方がオーバヘッドが大きくなっていることを図 6 から推測できる．

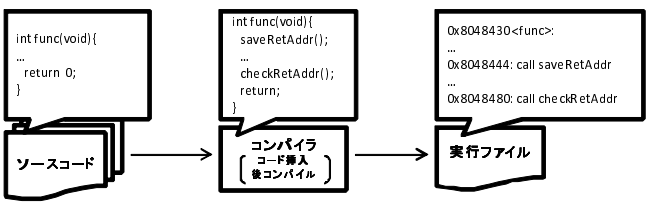


図 5: コード埋め込み

表 1: リターンアドレス書き換え攻撃の検知テスト

攻撃方法	検知
Strcpy overflow	○
Fork child and strcpy overflow	○
Strcat overflow	○
Format string exploit	○
Return-into-libc	○

表 2: 動作テスト

プログラム	動作	備考
gzip	○	linux-kernel.2.6.27.41 を解凍
tar	○	linux-kernel.2.6.27.41 を圧縮
httpd	○	wget によるファイル取得
bind	○	起動までを確認

表 3: gzip と tar に対するオーバヘッド測定

	gzip		tar	
動作	解凍		アーカイブ作成	
対象ファイル	linux-2.6.27.41.tar.gz		linux-2.6.27.41	
総関数呼び出し数	51843回		2624558回	
関数呼び出しの平均深度	8.421		22	
	実行時間 (sec)	オーバヘッド (/通常カーネル)	実行時間 (sec)	オーバヘッド (/通常カーネル)
0.通常カーネル	5.001	--	9.773	--
1.リターンアドレス検査	5.193	3.83%	15.54	58.96%

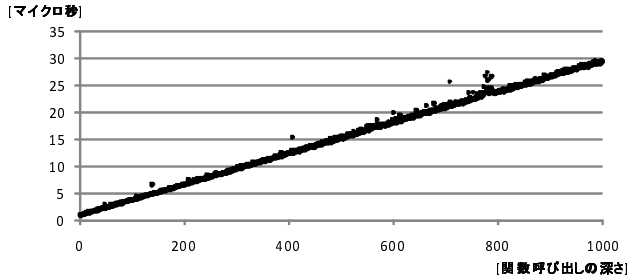


図 6: 関数呼び出しの深さとオーバヘッド

5 おわりに

本稿では，リターンアドレス検査を用いたスタック偽装攻撃検知手法を述べた．検査にかかるオーバヘッドは gzip に関しては 3.83%と低オーバヘッドである．また，リターンアドレス保護により他のバッファオーバフロー攻撃も検知することができる．今後の課題としては，sysenter 命令などを利用したオーバヘッド削減を行う予定である．

参考文献

[1] D. Wagner and D. Dean. Intrusion detection via static analysis. *IEEE Symposium on Security and Privacy*, pp. 156–169, 2001.

[2] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. *Proceedings of the 9th ACM Conference on Computer and Communications Security*, November 2002.

[3] 槙本裕司, 鶴田浩史, 齋藤彰一, 上原哲太郎, 松尾啓志. システムコールとライブラリ関数の監視による侵入防止システムの実現. 情報処理学会研究報告. [システムソフトウェアとオペレーティング・システム], pp. 3–10, June 2009.

関数のリターンアドレス保護による スタック偽装攻撃検知

立命館大学大学院　理工学研究科
冨永 悠生

立命館大学　情報理工学部
樫山 武浩　瀧本 栄二　桑原 寛明　毛利 公一 國枝 義敏

はじめに

- 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攻撃
    - **システムの乗っ取り**
    - **情報漏洩**
- 一般的なIDSで検知できない攻撃
  - スタック偽装攻撃（Mimicry Attack）
  - 文字列の書き換えによる攻撃
  - 変数の書き換えによる攻撃

目的

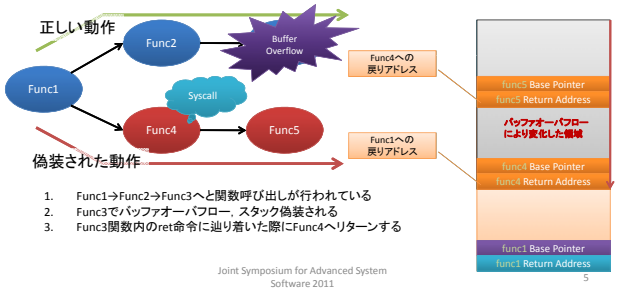
- 本研究の目的
  - スタック偽装攻撃を防ぐ
- 狙い
  - **脆弱性が存在するプログラムでも最低限のセキュリティ確保**
  - ゼロディ攻撃
    - 脆弱性の発見から修正までの期間に受ける攻撃
    - 即座に修正はできないため，一時無防備な状態となる

バッファオーバフロー攻撃

- リターンアドレス書き換えを用いた攻撃
  - 任意のアドレスから命令を再開できる
    - return-into-libc攻撃
    - コードインジェクション攻撃
- 攻撃手順
  1. 変数xがバッファオーバフロー
  2. BasePointerとReturnAddressが書き換え
  3. ret命令が発行されると書き換えられたアドレスに遷移することが可能

スタック偽装攻撃（MimicryAttack）

- バッファオーバフローから派生する攻撃
  - スタックを書き換えて異なるフロー状態にする
    - 静的解析情報のみを用いたIDSでは検知不可能



1. Func1→Func2→Func3へと関数呼び出しが行われている
2. Func3でバッファオーバフロー，スタック偽装される
3. Func3関数内のret命令に辿り着いた際にFunc4へリターンする

スタック偽装攻撃を防ぐには

- 全スタックのリターンアドレスが書き換えられているか検査する必要がある
  - 条件
    - リターンアドレスは，プログラム内で書き換えられない
    - スタックフレーム構造を必ず持っている
- リターンアドレスを関数の開始時に保存
- 関数の終了時にリターンアドレスが正しいか検査する

関数のリターンアドレス保護によるスタック偽装攻撃検知

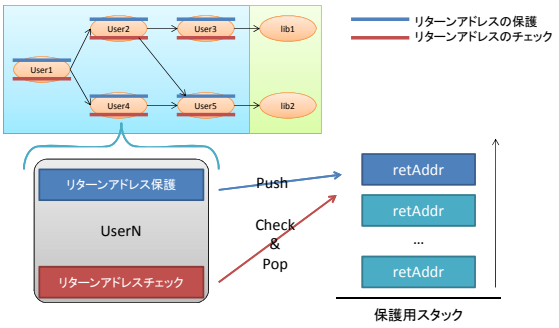
- 概要
  - 全スタックフレーム検査によるリターンアドレス書き換えの検知
- 動作
  - 関数の開始時でリターンアドレスを保護
  - 関数の終了時でリターンアドレスを検査

- 検知タイミング
  - バッファオーバフローが発生した関数を抜けるとき検知可能

関数内への検知コード埋め込み

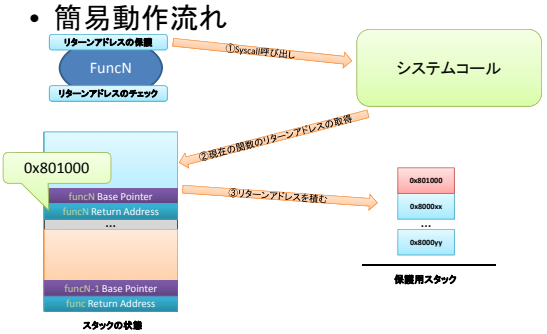
- リターンアドレスの保護とリターンアドレスのチェックはコード埋め込みによって実現
  - 関数の先頭
    - リターンアドレスを保護するコードを埋め込む
  - 関数の末端（return前）
    - リターンアドレスのチェックをするコードを埋め込む

コールグラフ図と動作概要



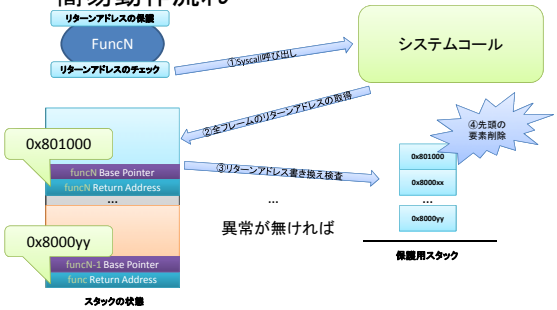
リターンアドレスの保護

- 簡易動作流れ



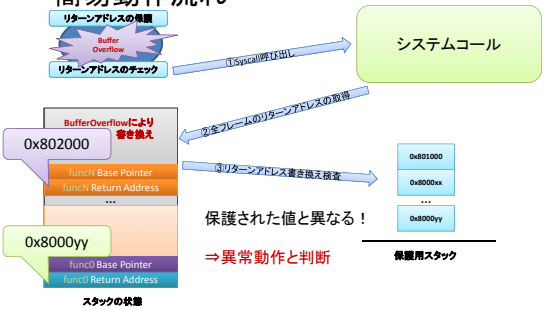
リターンアドレス書き換え検査

- 簡易動作流れ



バッファオーバフローが発生

- 簡易動作流れ



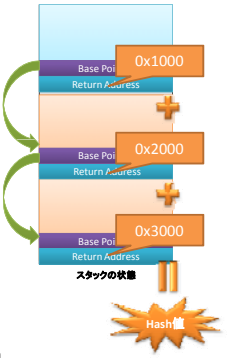
実装（コード埋め込み）

- 共有ライブラリに以下の関数を実装
  - __cyg_profile_func_enter
    - リターンアドレスの保護システムコールの呼び出し
  - __cyg_profile_func_exit
    - リターンアドレスの検査システムコールの呼び出し
- リターンアドレス保護と検査を行うためのコード挿入
  - 関数の入り口と出口をフックするGCCのオプション –finstrument-functionsを利用
    - 関数の入り口にcall __cyg_profile_func_enter
      関数の出口にcall __cyg_profile_func_exit
      が挿入される
- プログラムの実行時にLD_PRELOADで共有ライブラリを読み込ませる

実装（システムコール）

- 保護用スタックに積む値
  - 全てのリターンアドレスをxorした値
    - コールバック関数などを考慮した実装の都合

- sys_save_return_address
  - スタックのBPを辿り，RetAddrを求める
  - RetAddrはxorでHash化する
  - Hash化した値を保護用Stackに積み上げる
- sys_check_return_address
  - スタックのBPを辿り，RetAddrを求める
  - RetAddrはxorでHash化する
  - 保護用Stackの先頭を取り出し，値を比較する
    - 異なっていれば，プログラムを強制終了させる

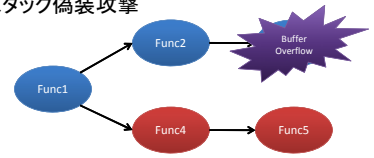


実験

- 3つの観点
  - 異常検知のテスト
    - スタック偽装攻撃の検知テスト
    - リターンアドレスを書き替える攻撃のテスト
  - 動作テスト
  - オーバヘッド
    - 一般プログラムに対するオーバヘッド測定
    - ベンチマークによるオーバヘッド測定

実験結果（異常検知テスト）

- スタック偽装攻撃の検知テスト
  - 事前準備：Func1→Func4→Func5と移動した際のスタックダンプを取得
  - Func3でバッファオーバフロー，スタックダンプを利用してスタック偽装攻撃



⇒バッファオーバフローを検知し，プログラムの停止を確認した

実験結果（異常検知テスト）

- リターンアドレス書き換える攻撃の検知テスト
  - exploitコードを用いてプログラムが正しく停止するか確認
    - Libsafeに付属されているexploitコードを利用

攻撃方法	本研究	StackGuard
Strcpy overflow	○	○
Fork child and strcpy overflow	○	○
Strcat overflow	○	○
Format string exploit	○	×
Return-into-libc	○	○

実験結果（動作テスト）

- 誤検知無く動作するか確認

プログラム名	動作	備考
gzip	○	カーネルソースコードを解凍
tar	○	カーネルソースコードのアーカイブ作成
httpd	○	httpd起動後，wgetによるファイル取得
bind	○	起動までを確認

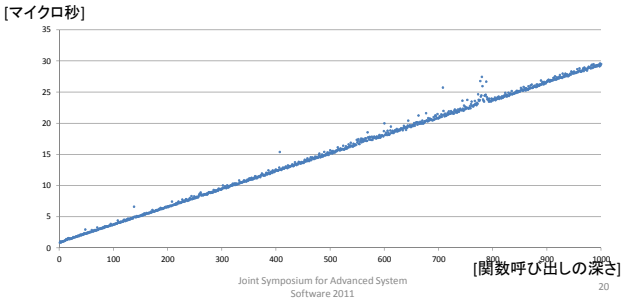
実験結果（オーバヘッド）

- 一般的なプログラムに対して動作

	gzip		tar	
動作	解凍		アーカイブ作成	
対象ファイル	linux-2.6.27.41.tar.gz		linux-2.6.27.41	
総関数呼び出し数	51843回		2624558回	
関数呼び出しの平均深度	8.421		22	
	実行時間 (sec)	オーバヘッド (/通常カーネル)	実行時間 (msec)	オーバヘッド (/通常カーネル)
0.通常カーネル	5.001	--	9.773	--
1.リターンアドレス検査	5.193	3.83%	15.536	58.96%

実験結果（ベンチマーク）

- システムコールのオーバヘッド測定
  - 1000回させた際の各関数における測定値



考察

- スタック偽装攻撃を検知でき，多くのプログラムで一緒に動作することを確認した
  - 検知精度は十分と考えられる
- オーバヘッドの増加差
  - 総関数呼び出し数と深度に依存する
    - gzipとtarの総関数呼び出し数と関数呼び出しの平均深度は以下の様になっている

	gzip	tar
総関数呼び出し数	51843回	2624558回
関数呼び出しの平均深度	8.421	22

    - 関数呼び出し毎にオーバヘッドがかかるため，関数呼び出し数が多いtarのオーバヘッドが大きくなっている

おわりに

- 本手法により，リターンアドレス書き換えを検知することができた
  - exploitコードを利用した検知精度のテストではすべて検知できた
    - StackGuardより検知精度が良い
- 一般的なプログラムに対して本手法を適応した
  - Httpd，Bindに対して容易に適応できた
    - 十分な汎用性がある
- 今後の課題
  - オーバヘッド削減
  - Httpdなどのプログラムに対する評価
  - 使いどころの考慮

組込みシステム向けプロセスロギング機構の開発

アモーンタマウット プラウィーン† 早川 栄一††

組込みシステムを対象とした省メモリの Linux プロセスのロギング機構の開発を行った．組込みシステムでシステム動作のロギングを行う場合，メモリ容量に限界があるため，ロギングデータ取得する際にリアルタイムでの割込み発生したコンテキストスイッチのログの消失が起こる可能性がある．この問題を解決するために Linux が提供している Trace をベースとしたコンテキストスイッチログを圧縮する機能を追加し，少ないオーバヘッドと，少ないメモリ容量の環境でログを生成・取得が可能な「圧縮機能付き Trace」の設計，プロトタイプの実装，評価を行った．

Development of process logging mechanism for embedded system

Praween Amontamavut, † Eiichi Hayakawa††

We have developed a process logging mechanism for Linux embedded OS which operate on a little memory embedded system's environment. In this case, if we take a context switch's log from the logging's system, by the part of log data including real-time processing log data may be lost because of the memory limited. By the problem-solving, we use the Trace which process logging system provide by Linux kernel to design the Trace possible to operate on a little memory embedded system's environment by add a little overhead compression function to it, Trace with compression function Prototype implementation, and evaluated it.

1 はじめに

組込みシステムはリアルタイム性に向かっている．これに対するリアルタイムOSのプロセススケジューリングの研究開発や学習などを対象とするプロセススケジューリングのデバッグや可視化などのニーズがある．

早川研究室も，カーネルプロセス及びユーザプロセスを学習するために，2009 年から Linux および Android を対象としたプロセスの可視化について研究を行ってきた．

プロセススケジューリングのデバッグは一般の Linux がトレーサによるコンテキストスイッチのログを扱う．プロセスの可視化も，コンテキストスイッチのログを取得し，保存してから解析を行う．リアルタイム性的なプロセススケジューリングを監視するため両者とも長時間のコンテキストスイッチが必要になる．しかし，生成されたコンテキストスイッチの容量が短時間であっても大きいことから，バッファ容量が少ない組込みシステムでは，問題が発生する．

その理由としては，コンテキストスイッチのログを生成する速度がコンテキストスイッチのログを取得する速度よりも速いことにある．一般にシステムでは，リングバッファを用いて速度差を吸収するが，データサイズが大きく，過去のデータが上書きされ，リアルタイム的な割込みが発生するそのプロセスログの情報を含む情報の一部がロストする可能性がある．また，データの外部への転送では，それ自体にオーバヘッドがあり，プロセス実行に影響を与える可能性がある．

2 従来のトレース

2.1 トレース機構

従来のトレース機構は図1のように「トレース内部機構」と「トレース外部機構」に分割できる．

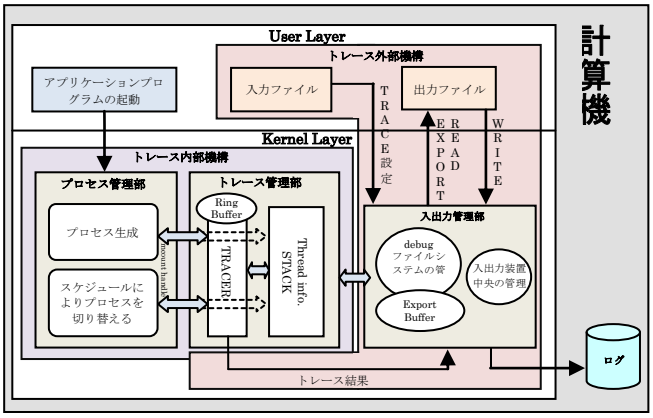


図１従来のトレース機構

前者はプロセススケジューリングを管理するプロセス管理部との関連性が高い，コンテキストスイッチのログを生成させる主な機能を担当する．これを「ログデータ生成部」と考えられる．ログを生成する機能は，

† 拓殖大学工学部情報工学科 4 年生
Faculty of Engineering, Department of Information Engineering, Takushoku University ,4th grade
†† 拓殖大学工学部情報工学科
Faculty of Engineering, Department of Information Engineering, Takushoku University

RingBuffer に，後者がユーザ設定によるログの要求を実行するまでにログデータを Non-Blocking 的に収集するのである．

後者は，ユーザ空間とカーネル空間を占め，カーネル空間内に所有する前者からユーザ空間に所有するユーザにログデータをエクスポートさせる機能を担当する．これをログデータエクスポート部と考えられる．ログデータをエクスポートする機能は，前者にログデータ要求信号を通知し，RingBuffer から ExportBuffer に Blocking 的にログデータを収集し Debug ファイルシステムを通じてログデータをユーザ空間にエクスポートする機能である．

2.2 コンテキストスイッチのログ

コンテキストスイッチのログはカーネルによるプロセススケジューリングの追跡を考えられ，プロセス生成やプロセス状態のスイッチなどの情報を表すログである．このログはトレース機構で述べたように RingBuffer に格納されるバイナリデータを，図 2 のように文字列化し ExportBuffer に収集しユーザ空間にエクスポートする形になる．

```
sh-42    [000] 4154504421.591616:    42:120:S      + [000]    42:120:S
sh-42    [000] 4154504421.591616:    42:120:S    ==> [000]     0:140:R
```

図 2 コンテキストスイッチのログの例

ログの意味は図3に表す．

```
{プロセス名}-{プロセス ID}  [現在の CPU] {エントリ時刻秒単位}.{エントリー時刻
マイクロ秒単位}: {現在のプロセス ID}:{現在の優先度}:{現在のプロセス状態}
{エントリの種類} [次の CPU] {次のプロセス ID}:{次の優先度}:{次のプロセス状
態}
```

図 3 コンテキストスイッチのログの意味

2.3 問題点とその影響への仮定

問題は主に次の 2 点を考えられる．

・ システムが提供する 1 ページ*のエクスポートバッファに格納されるコンテキストスイッチのログは無駄に文字列化しエクスポートするため，速度差を吸収できない問題により，エクスポートバッファがリングバッファを追えない可能性があり，リアルタイム的割込み発生プロセスのデータを含め，データロスト問題が発生する可能性がある．

・ ログデータのエクスポートはユーザ空間にデータを提供させる Debug ファイルシステムを一度通過しエクスポートする形になっているので，自分自体オーバヘッドがある．また，組込みシステムは機器内のフラッシュロム容量に制限があり，機器内に大容量なログデータを保存することはできない．

3 本研究のトレース

3.1 トレース機構の設計

本研究は機器内にログを保存することではなく，外部計算機の領域を用い保存することで，図4のように「ログ生成側という組込み機器」と「ログ取得側というログ収集計算機」に分別することになる．

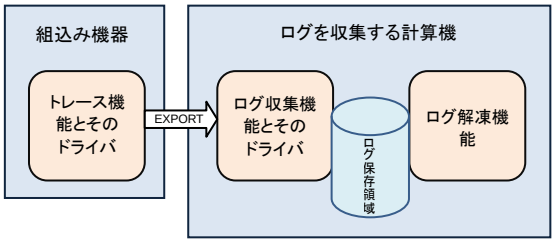


図 4 本研究の全体的なトレース機構

前者は従来トレースと同様にログを生成しエクスポートする．しかし，エクスポートする向きや追加機能などは異同で，次に詳しく述べる．

トレース内部機構はプロセス管理部との関連性が高いので，カーネル全体機構に影響が与えるため，トレース内部機構に構造体を再設計したりすることは困難である．このため，トレース外部機構を改良することにする．

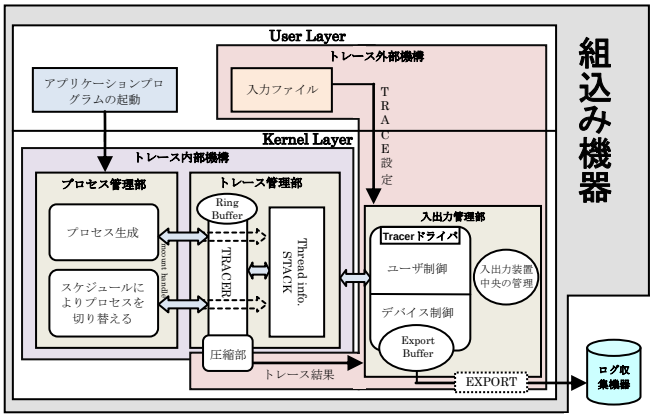


図 5 本研究のログ発生側のトレース機構

本研究の設計は従来のトレースと同様にトレース外部機構とトレース内部機構に分割するようにするが，トレース外部機構には図5のように圧縮部を追加し，直接データを装置にエクスポートできるようにするため，トレースドライバを追加することにする．

後者は前者からの全てのログを取得し保存する．そして，ログ解凍機能により圧縮されたログを解凍し提供させるようにする．

---

* 1ページのサイズは 4096 バイトであるが,システムによって 1 ページのサイズが変わる.[1]

3.2 圧縮部の機能の設計

圧縮概要としては，リングバッファから出力されたデータをできればエクスポートバッファに小領域で蓄えるようにする．

圧縮機能の設計は従来トレースの出力ログの特徴を分析し，適切な方法を選択する．コンテキストスイッチのログの特徴は三つに考えられる．

（1） データ表現の範囲が広く，同じデータの繰り返し頻度が高いデータ
（2） データ表現の範囲が狭いデータ
（3） データ表現の範囲がちょうど良いデータ

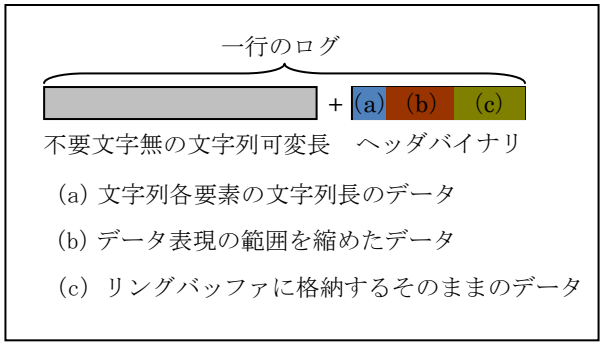


図 6 圧縮機能により ExportBuffer に格納する一行のログ

これで，三つの特徴によって，エクスポートバッファに蓄える一行のログを図 6 のように「文字列部」と「バイナリ部」に分割する．

前者は不要な文字を消した文字列データのグループにする．

後者はデータサイズが最も小さい固定サイズのバイナリ部をヘッダバイナリデータにし，解析するのにかかるオーバヘッドを削減するため，後末にする．

そして，解析したログの特徴に従ってデータを秩序する．(1)の特徴のデータはそのデータの文字列の長さを示す適切な範囲で表現するバイナリデータを持った文字列可変長データにする．文字列可変長の解析は次に述べる．

二回以上出続するデータであれば，その二回目以降のデータを前者に書き込まないようにし，長さを示す(a)に「0」を書き込む．その他は普段どおりに前者に書き込み，その長さを(a)に書き込む．

(2)の特徴のデータは表現する範囲を縮め，後者にバイナリデータを書き込む．

(3)の特徴のデータは何もせずにそのまま後者に書き込む．

3.3 解凍方法

後末から固定サイズのヘッダバイナリを読込み，図 6 による(a)～(c)の各データを図 7 のような中間ログに解析する．

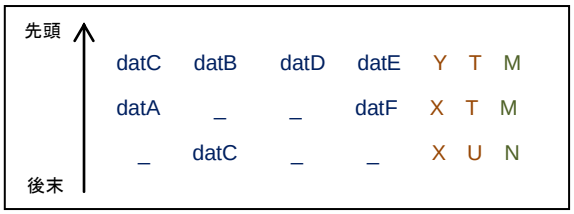


図 7 中間ログのイメージ（説明用）

(a)のデータ部分は各要素の文字列の長さをバイナリ部から読み込み，各要素の文字列を分離する．文字列の長さが「0」の場合は，文字列部にデータがエクスポートされていないので要素分離困難を回避するため「 __ 」に格納する．そうではない場合は，その要素の長さだけを各要素の文字列に格納する．(b)と(c)のデータ部分はそのまま文字列化し，格納する．

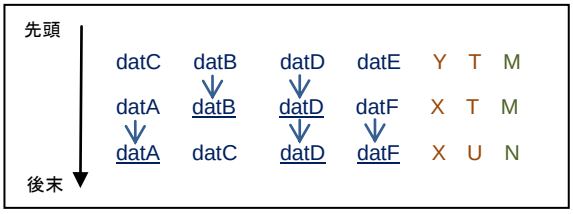


図 8 解凍した最終ログのイメージ（説明用）

解析された中間ログを図 8 のように先頭から読み込み，Implement 処理を行い，最終ログに解析する．

4 圧縮機能付きトレースの評価

4.1 評価システムの環境

本研究は KZM-A9 評価ボード上に動作するカーネルバージャン2．6．29をベースにシステムプロトタイプを合成し評価する．環境の詳細は下記に参照することである．

Hardware: KZM-A9 評価ボード

- CPU
  - ARM Cortex-A9MPCore™ Dual CPU
  - Operating Frequency 533MHz (MAX)
  - CPU Data cache 32KB/CPU
  - CPU L2 cache 256KB
- Memory
  - Main Memory 512MB (DDR2-533)
  - Internal SRAM 128KB
  - Internal ROM 64KB
  - eMMC NAND 4GB

Software: Linux kernel 2.6.29

4.2 評価解析

注目する評価点はデータ収集効果とデータ生成オーバヘッドの2点がある．

前者は圧縮機能を追加するトレースによるログデータを収集する効果の傾向を，圧縮機能を追加しない従来トレースによるログデータを収集する効果の傾向と比

較し，データ消失程度を監視する．データ収集効果は式①でまとめる．

$$\text{ログデータ収集効果} = \frac{\text{使用ログデータ容量}}{\text{CPU 時刻の間隔}} \quad \cdots\cdots\cdots ①$$

後者は前者に対するオーバヘッドを目指す．圧縮機能を追加するトレースによってログを生成し収集するまでのオーバヘッドを圧縮機能を追加しない従来のトレースの場合のそのオーバヘッドと比較することになる．データ生成オーバヘッドは式②でまとめる．

$$\text{データ生成オーバヘッド} = \frac{\text{CPU 時刻の間隔}}{\text{生成ログデータ容量}} \quad \cdots\cdots\cdots ②$$

生成ログデータ容量はリングバッファに格納され，エクスポートされる全体的なログデータの容量を考えられ，使用ログデータ容量はエクスポートされ，それを実際に使用することになる全体的なログデータの容量を考えられる．

4.3 評価

表 1 従来トレースの評価

CPU の時刻の間隔 (T)t	生成ログデータ容量 (Bytes)	使用ログデータ容量 (Bytes)
78	166296	166296
87	183066	183066
91	191334	191334
119	218868	218868
134	247728	247728
221	464334	464334

表 2 圧縮付きトレースの評価

CPU の時刻の間隔 (T)t	生成ログデータ容量 (Bytes)	使用ログデータ容量 (Bytes)
31	53485	140958
45	97899	251002
80	141582	347963
105	228228	554466
113	196416	473858
134	278084	684759

評価は従来のトレースと本研究で合成した圧縮付きトレースを行う．従来トレースの評価は表1に表す．本研究で合成した圧縮機能付きトレースは表２に表す．

両者も式①と式②に従って，ログデータ収集効果とデータ生成オーバヘッドをそれぞれに計算し，図9と図10のような比較グラフを描く．それぞれにグラフの特徴関数を書く．

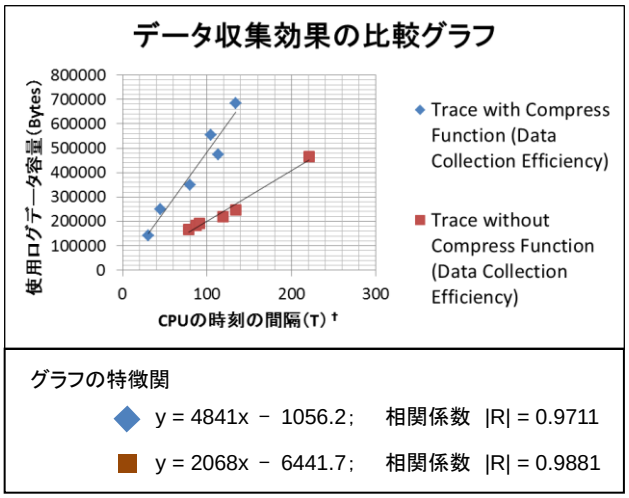


図 9 データ収集効果の比較グラフ

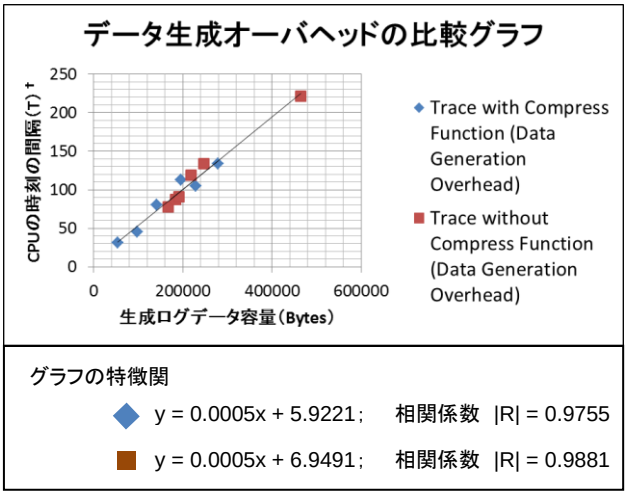


図 10 データ生成オーバヘッドの比較グラフ

グラフによると，圧縮機能付きトレースは圧縮機能無し従来トレースよりも二倍データ収集効果が高く，オーバヘッドも従来トレースとは変わらないことが判明した．

これで，圧縮付きトレースは小容量 ExportBuffer に低オーバヘッド圧縮機能により圧縮した小容量データを蓄えるので，ExportBuffer が RingBuffer を従来トレースよりも追えることができ，ExportBuffer と RingBuffer 間のデータ吸収性が高くなるため，データ収集効果が高くなるということを考えられる．

ということで，圧縮機能付きのトレースは省バッファ環境の下であっても，データ収集効果を上げ，データ消失率を下げ，総合コストも下げることができるのであ

る.

5　おわりに

5.1 まとめ

本研究はトレース内部機構設計ではなくトレース外部機構設計を行っているので，カーネル全体に影響を与えずに，問題を解決するためシステム合成が可能であった.

本研究のシステムは圧縮機能を追加し，ログ生成機器とログ収集計算機に分離した.

前者はログを生成する側にデータ収集効果を上げ，リアルタイム的割り込み発生プロセスデータを含めたデータ消失の問題解決が可能であった.

後者は大容量のログデータの保存が可能で，保存容量が少ない問題にも回避可能であった.

5.2 今後の課題

このドライバからプロセスが使用しているメモリ領域の情報を取得可能な機能を追加する検討を行う.

ログを解凍する側に解凍したログデータを提供するプログラムとその API を作成し，提供させる.

参 考 文 献

[1] ”LINUX DEVICE DRIVER”, Alessandro Rubini + Jonathan Corbet, O’REILLY

組込みシステム向け
プロセスロギング機構の開発

拓殖大学工学部情報工学4年生 †
拓殖大学工学部情報工学科 † †
Praween Amontamavut †、早川栄一 † †

背景

- プロセスの（可視化やデバッグ）のニーズ
  - 組込みシステムはリアルタイム性に向けている
  - これに対する
    - 組込みOSのスケジューリングの学習
    - 組込みOSのスケジューリングの開発
  - コンテキストスイッチで長時間ログを収集してから、可視化したりデバッグしたりするニーズ

背景

- 早川研究室
  - カーネルプロセス+ユーザプロセスを学習するために, 2009年からLinuxおよびAndroidを対象としたプロセスの可視化について研究を行ってきた
- プロセスの可視化
  - Linuxカーネルプロセスも含むため、Traceを用いる
  - Traceによって生成されたプロセスのログを取得し, 保存してから解析を行う

```
          sh-42  [000] 4154504421.591616:   42:120:S    + [000]   42:120:S
          sh-42  [000] 4154504421.591616:   42:120:S ==> [000]    0:140:R
```

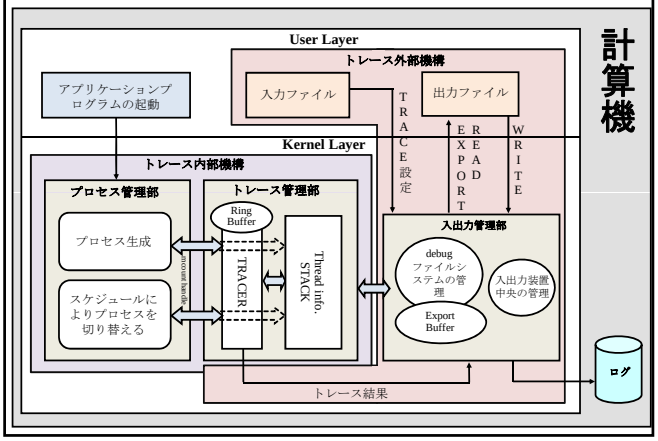
問題点

- コンテキストスイッチのログ
  - データを無駄な文字列化しエクスポートする形の欠点がある
    - エクスポートログ容量は、ユーザアプリケーション起動しない状態で平均一秒当たりの12833バイトで、容量が大きい
      - ユーザアプリケーション動作しながらログをとると、これの2～3倍に
- 組込みシステム
  - 機器内のフラッシュロム
    - 容量がeMMCの4GBで従来のような磁器ディスクより少ない
      - 従来のように機器内の磁器ディスクに保存することもできない
  - バッファ容量はシステムが4096バイトの1ページのエクスポートバッファを用意している
    - 1ページのサイズはシステムによって変わる

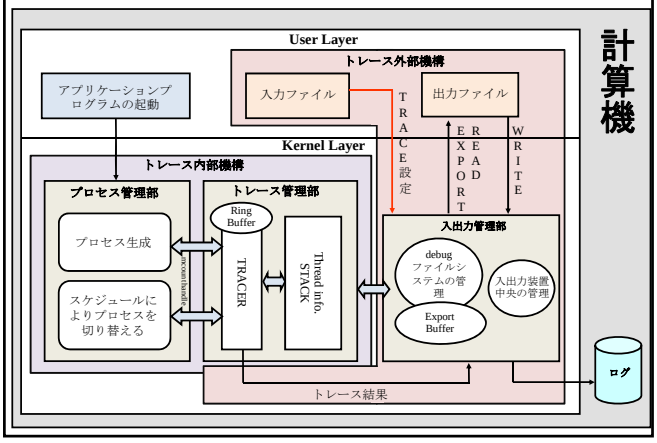
問題点

- Trace機構
  - プロセス管理部と関連するトレース内部機構 + ログを収集して提供するトレース外部機構に分離できる
  - トレース内部機構はプロセス管理部との関連性が高い
  - カーネル全体機構に影響が与える可能性があるため、トレース内部機構を再設計するのは好ましくない
    - トレース外部機構に触る

従来トレースの機構



従来トレースの機構



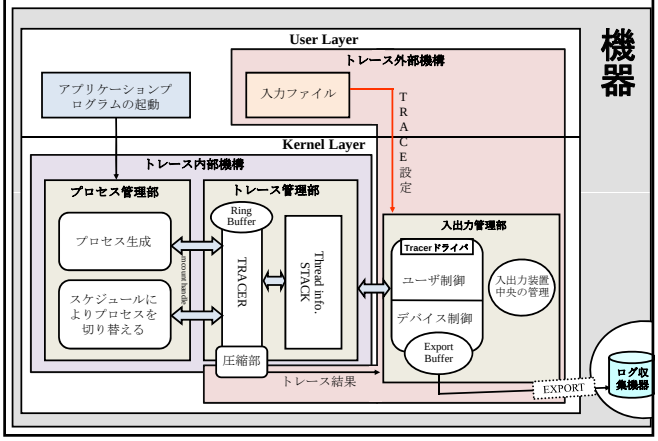
問題点とその影響の仮定

- 内部機構の一部（ログデータの生成）
  - 一般にシステムではリングバッファを用いる
    - 組込みシステムではバッファが少ないため速度差を吸収できない可能性がある
      - リアルタイム的割込み発生プロセスのデータを含め、データロスト問題発生可能性がある
  - ログを生成し収集するオーバヘッドがあるが、システム全体に影響を与えない
- 外部機構の一部（ログデータのエクスポート）
  - 自分自体にオーバヘッドがあり、ユーザ空間にエクスポートするため、プロセス実行に影響を与える可能性がある

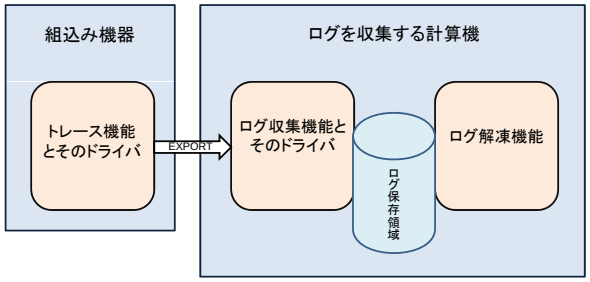
目的

- Export BufferをRing Bufferに追い付けるようにし、低オーバヘッドで、ログデータ収集効果を上げ、総コストを減少させる
  - Export Bufferにデータを多く保持できるように 、少オーバヘッド圧縮機能を追加する
- 機器内のフラッシュメモリ容量問題を回避する
  - 「ログを発生する側」＋「ログを取得する側」に分割する
    - ログを生成しドライバから送信させ、取得する側もドライバから受信し保存する

本研究のトレースの機構



本研究の全体的なトレース機構



圧縮機能

- 機能概要
  - 不要な文字は出力しない
    - 空白、区別字、改行
  - 二回以上同じデータが繰り返すなら、少ないデータ領域で表現する
  - 幅狭いデータは文字の１バイトではなく、その幅に相当する領域で表現する
- 機能設計
  - 出るログ形による特徴をベースに注目し、設計する

圧縮機能

- 機能設計
  - – 出力ログの形による特徴をベースに注目を払う

sched_switch（コンテキストスイッチ）によるトレースのログの一部

sh	42	000	4154504421	591616	42	120	S	+	000	42	120	S
sh	42	000	4154504421	591616	42	120	S	==>	000	0	140	R
<idle>	0	001	4154504421	691616	0	140	R	+	001	90	118	S
<idle>	0	001	4154504421	691616	0	140	R	==>	001	90	118	R
er.ServerThread	90	001	4154504421	691616	90	118	R	+	001	85	120	S

comm　cpu　usec_rem　prev_prio　entry_type　next_pid　next_state

entry_pid　secs　prev_pid　prev_state　next_cpu　next_prio

従来トレース（sched_switchの場合）

データ名	リングバッファに格納される領域	エクスポートバッファに格納される領域
comm	16バイト	固定16バイト
entry_pid	4バイト	固定7バイト
cpu	4バイト	固定5バイト
secs	8バイト	可変長2~11バイト
usec_rem	8バイト	固定7バイト
prev_pid	4バイト	固定7バイト
prev_prio	1バイト	固定4バイト
prev_state	1バイト	固定3バイト
entry_type	1バイト	固定3バイト
next_cpu	4バイト	固定5バイト
next_pid	4バイト	固定7バイト
next_prio	1バイト	固定4バイト
next_state	1バイト	固定3バイト
合計	57バイト	最小74～最大83バイト

従来トレース

トレースのログの特徴の分類：

- データ表現範囲が広くて、同じデータの繰り返し頻度が高いデータ
- データ表現範囲が狭いデータ
- データ表現範囲がちょうど良いデータ

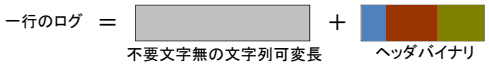
本研究のトレース（sched_switchの場合）

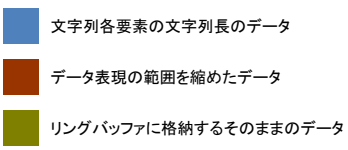
データ名	リングバッファに格納される領域	エクスポートバッファに格納される領域	
		バイナリ部	文字列部
comm	16バイト	■5ビット	0～16バイト
entry_pid	4バイト	■4バイト	無
cpu	4バイト	■2ビット	0～3バイト
secs	8バイト	■5ビット	0～10バイト
usec_rem	8バイト	■5ビット	0～6バイト
prev_pid	4バイト	■4バイト	無
prev_prio	1バイト	■1バイト	無
prev_state	1バイト	■3ビット	無
entry_type	1バイト	■2ビット	無
next_cpu	4バイト	■2ビット	0～3バイト
next_pid	4バイト	■4バイト	無
next_prio	1バイト	■1バイト	無
next_state	1バイト	■3ビット	無
合計	57バイト	18バイト	最小0～最大38バイト
		最小18～最大56バイト	

■文字列長
■そのままのバイナリデータ
■表現の範囲を縮めたバイナリデータ

Export Bufferに書き込むデータ

- 以下の図のように、データを「不要文字無しの文字列可変長」とヘッダバイナリにまとめ、ヘッダを後末にする





本研究のトレース（圧縮特徴）

- データ表現範囲が広くて、同じデータの繰り返し頻度が高いデータ：
  - 「文字列可変長」にする
- データ表現範囲が狭いデータ：
  - その範囲に相当する領域にすれば、繰り返す程度を無視しても良い
- データ表現範囲がちょうど良いデータ：
  - そのままバイナリデータを出すれば良い

圧縮の特徴

（1） データ表現範囲が広くて、同じデータの繰り返し頻度が高いデータ:
- 「文字列可変長」にする

この概要の特徴:

データ容量減少程度は同じデータ繰り返しの頻度とのDirect Variationになっている

従来トレースにある部分の特徴は
繰り返し頻度が高いため、圧縮性能が高い

＊ 文字列の長さをまとめるためのオーバヘッドや、文字列に変換するオーバヘッドなど

圧縮の特徴

（2） データ表現範囲が狭いデータ:
- その範囲に相当する領域にすれば、繰り返す程度を無視しても良い

この概要の特徴:

－ 文字列変換しなく、少オーバヘッドで通常より少領域でデータを把握できる

－ （1）よりオーバヘッドが少ないため、その分は吸収できる

圧縮の特徴

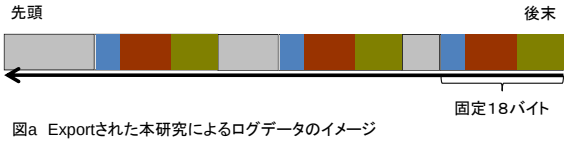
（3） データ表現範囲がちょうど良いデータ:
- そのままバイナリデータを出すれば良い

この概要の特徴:

－ 文字列変換しなく、（1）より少オーバヘッドでデータを把握できる

－ （1）よりオーバヘッドが少ないため、その分は吸収できる

解凍方法(取得側)



図a Exportされた本研究によるログデータのイメージ

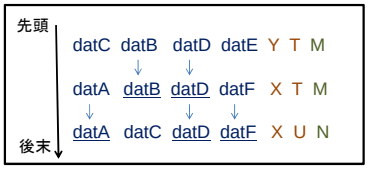
（1） 後末からの最後のヘッダから読み込み、後末から中間ログに解釈する
中間ログ:
第1族: 灰色にある各種の文字列のデータ長を表す青色部分を取り、各種の文字列データに分割する。データ長が0の場合、「_ 」に格納する
第2族+第3族: ヘッダにある各種バイナリデータをそのまま文字列化する

解凍方法(取得側)

図b 中間ログのイメージ例（説明用）

（1） 後末からの最後のヘッダから読み込み、後末から中間ログに解釈する
中間ログ:
第1族: 灰色にある各種の文字列のデータ長を表す青色部分を取り、各種の文字列データに分割する。データ長が0の場合、「_ 」に格納する
第2族+第3族: ヘッダにある各種バイナリデータをそのまま文字列化する

解凍方法(取得側)



図c 解凍したログのイメージ（説明用）

（2） 先頭からデータを一行ずつインプリメントし、意味が取れるログの文字列に直す

圧縮機能付きトレースの評価

- システムの評価の環境
  - KZM-A9評価ボード
    - CPU
      - ARM Cortex-A9MPCore™ Dual CPU
      - Operating Frequency 533MHz (MAX)
      - CPU Data cache 32KB/CPU
      - CPU L2 cache 256KB
    - Memory
      - Main Memory 512MB(DDR2-533)
      - Internal SRAM 128KB
      - Internal ROM 64KB
      - eMMC NAND 4GB

圧縮機能付きトレースの評価

従来トレース評価

CPUの時刻の間隔（T）†	生成ログデータ容量（Bytes）	使用ログデータ容量（Bytes）	使用ログデータ行数（行）	ログデータ収集効果（Bytes/T）	データ生成オーバヘッド（T/Bytes）
78	166296	166296	2132	2132.000000	0.000469043
87	183066	183066	2347	2104.206897	0.000475238
91	191334	191334	2453	2102.571429	0.000475608
119	218868	218868	2806	1839.226891	0.000543707
134	247728	247728	3176	1848.716418	0.000540916
221	464334	464334	5953	2101.058824	0.000475951

ログデータ収集効果＝使用ログデータ容量 / CPUの時刻の間隔

データ生成オーバヘッド＝CPUの時刻の間隔 / 生成ログデータ容量

† :current->stime から取り出した時刻データの単位を「T」として扱われている。

圧縮機能付きトレースの評価

圧縮機能付きトレース評価

CPUの時刻の間隔（T）†	生成ログデータ容量（Bytes）	使用ログデータ容量（Bytes）	使用ログデータ行数（行）	ログデータ収集効果（Bytes/T）	データ生成オーバヘッド（T/Bytes）	圧縮率
31	53485	140958	1719	4547.032258	0.000579602	0.620561
45	97899	251002	3061	5577.822222	0.000459657	0.609967
80	141582	347963	4519	4349.537500	0.000565044	0.593112
105	228228	554466	7133	5280.628571	0.000460066	0.588382
113	196416	473858	6154	4193.433628	0.000575310	0.585496
134	278084	684759	8822	5110.141791	0.000481869	0.593895

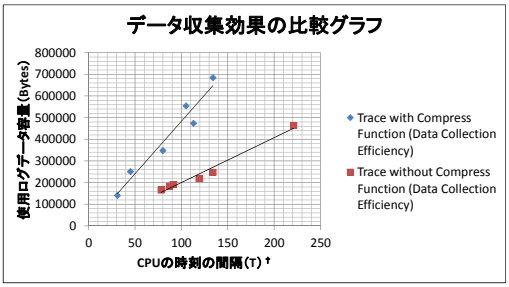
ログデータ収集効果＝使用ログデータ容量 / CPUの時刻の間隔

データ生成オーバヘッド＝CPUの時刻の間隔 / 生成ログデータ容量

圧縮率＝1－（生成ログデータ容量 / 使用ログデータ容量）

† :current->stime から取り出した時刻データの単位を「T」として扱われている。

圧縮機能付きトレースの評価

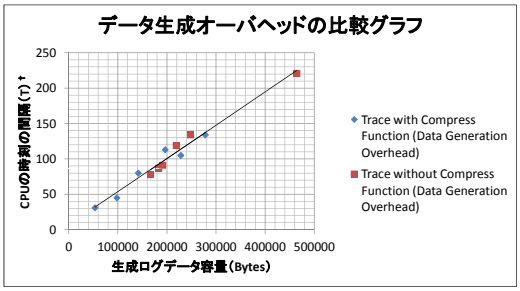


グラフの特徴関数

◆ y = 4841x – 1056.2;　相関係数 |R| = 0.9711

■ y = 2068x – 6441.7;　相関係数 |R| = 0.9881

圧縮機能付きトレースの評価



グラフの特徴関数

◆ y = 0.0005x + 5.9221;　相関係数 |R| = 0.9755

■ y = 0.0005x + 6.9491;　相関係数 |R| = 0.9881

比較結果

- 従来のトレース
  - 小容量ExportBufferに大容量データを蓄える
  - Exportする段階は遅いため、ExportBufferがRingBufferを追えない可能性がある
  - これによってデータ収集効果は低いためデータロスト問題もあった
- 圧縮機能付きのトレース
  - オーバヘッドは比較的に従来トレースとはかわらない
  - 小容量ExportBufferに小容量データを蓄える
  - Export Buffer がRing Bufferを一般トレースよりも追えられ、データ収集効果は高くなる
  - これによって小バッファ環境であってもデータロスト問題も解決できた

おわりに

- トレース内部機構設計ではなく外部機構設計になる
  - カーネル全体に影響を与えずに、問題解決できる
- 圧縮機能を追加した
  - ログ生成する側にデータ収集効果を上げ、リアルタイム的割込み発生プロセスデータを含めたデータロスト問題が解決できる
- ログ生成機器とログ収集計算機に分離した
  - 大容量のデータを蓄えることができる
  - 圧縮機能により収集データのサイズはより小さくなるため、よりデータを蓄えることができる

おわりに

- 今後の課題
  - 解凍する側に解凍したログデータを提供するプログラムとそのAPIを作成し、提供させる
- 関連研究
  - Measuring Function Duration with Ftrace
    - By Tim Bird
    - Sony Corperation of America

Android 向け OpenCL ライブラリの試作

望月 秋人†
東京農工大学大学院工学府情報工学専攻†

1 はじめに

近年，計算機に搭載されるプロセッサはその性能の向上のため，複数のコアを一つのプロセッサに集積する手法が一般的になってきている．加えて，このようなマルチコアプロセッサを利用する環境として，汎用処理を行うための CPU だけでなく，多数のコアを持つアクセラレータを備えるようなヘテロジニアス構成の並列演算システムも広まり始めている．このようなヘテロジニアス構成のシステムは，PC やサーバだけでなく，組込み分野においても用いられるようになっている．特に，高機能なサービスが要求され，従来よりも高い演算能力が必要とされている携帯端末ではその傾向が顕著である．しかし，組込みシステムの分野では PC と比較して，ヘテロジニアスシステム向けの並列演算環境の開発は遅れている．今回の研究ではその問題を解決するため，ヘテロジニアスな並列演算資源を用いた高い演算性能，及び同じコードを異なる実行環境で利用可能にする機能を提供する方式とそのライブラリを提案する．本ライブラリを携帯端末向けに採用が急増しているプラットフォームである Android 向けに開発することで，今後増加すると考えられる高性能な携帯端末にヘテロジニアス並列演算環境を提供できる．

2 背景

現在，PC やサーバではヘテロジニアス構成の並列演算システムは珍しいものではない．携帯端末でも CPU+GPU という構成は既に一般的であり，携帯端末で最も多く採用されている ARM アーキテクチャでは近い将来ヘテロジニアス構成のプロセッサが登場することが発表されており，今後より多くのデバイスでそのような並列演算資源を有効に利用するための環境が必要とされると考えられる．

ヘテロジニアス構成なシステム向けの既存の並列演算環境には，NVIDIA CUDA[1] や OpenCL[2]などがある．NVIDIA CUDA は基本的に自社のプロセッサを対象とし，利用範囲は限られている．それに対し OpenCL は，実装はベンダ依存であるが API が共通化されているのでソースコードのレベルであれば互換性がある．また，OpenCL では組込み機器向けプロファイルも規定されており，HPC 分野など高性能な環境を前提とされることの多い他の並列演算フレームワークと比較して組込み分野に適していると思われる．そのため，今回の研究では OpenCL を用いて，並列演算環境を構築するライブラリを試作する．

3 目的

本研究では OpenCL デバイスを実装するシステムにおいて，そのデバイスをアプリケーションから容易に利用することができ，異なるアーキテクチャにおいても同じコードが再利用可能なプログラムを作成，実行可能な並列演算環境を実現するシステムを開発する．この機能を携帯端末への導入が増加している Android 向けのライブラリとして提供することで，多くの携帯アプリケーション開発者が並列演算資源を利用できるようにする．

4 試作システム

図 1 に示すように，試作するシステムではシステムソフトウェアとして Android が動作する環境を対象とする．この Android の実行環境において VM 上で実行される Android の Java アプリケーションから，JNI（Java Native Interface）を通して OpenCL デバイスを利用するための環境を提供するライブラリの試作を行う．

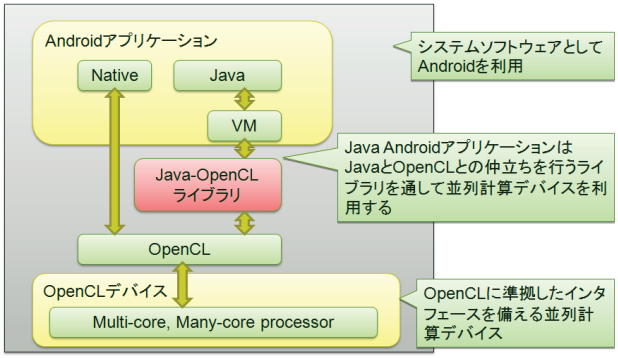


図 1. 試作システム概要

A Prototype of OpenCL Library for Android
† Akito MOCHIZUKI
Graduate school of Engineering, Tokyo University of Agriculture and Technology

4.1 Android 実行環境

現在, OpenCL を利用可能な Android の実行環境は非常に少ない．そのため，本研究では並木研究室で開発している OS 連携機構[3]を用いてマルチコア CPU の一部の CPU コアを OpenCL デバイスとして利用可能な Android の実行環境を構築し，試作ライブラリの実行環境とする．

本研究では 4.1.1 と 4.1.2 に示すように 2 通りの Android 実行環境を構築している．Linux のバージョンは 2.6.29, Android は 2.1 を用いる．

4.1.1 Android 実行環境 1

4 コアを備える SH-4A アーキテクチャのマルチコアプロセッサを対象とし，そのうちの 1 コア上で動作する Linux カーネルをベースとする Android を構築する．残りの 3 コア（コア 1～3）では Future+MULiTh[4]を動作させ, 並列演算デバイスとして利用する．この実行環境は現在主流であるシングル CPU の環境を模したものである．

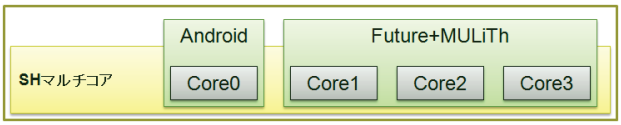


図 2. Android 実行環境 1

4.1.2 Android 実行環境 2

8 コアの SH-4A マルチコアプロセッサを対象として，Android の実行環境を構築する．Linux をコア 0～3 で実行，コア 4～7 を OpenCL デバイスとして利用する．この実行環境はこれから増加するであろう，マルチコア CPU＋メニーコアアクセラレータを模したものである．

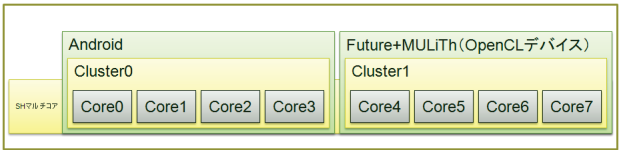


図 3. Android 実行環境 2

4.2 OS 連携機構

並木研究室で開発している OS 連携機構は汎用 OS（Linux）とマルチコアプロセッサを利用した演算に特化した専用 OS（Future+MULiTh）を連係動作させる機構である．Linux から Future を利用するインタフェースはデバイスドライバとして実装されており，OS 間のデータ通信には共有メモリを利用する．

Future とは東京農工大学で開発している CPU 資源管理，メモリ管理，I/O 管理を軽量にし，カーネル内処理のオーバヘッドを最小限にとどめる小型・軽量な OS であり，同じく東京農工大学で開発しているスレッドライブラリである MULiTh と組み合わせることで, 軽量なマルチスレッドプログラムの実行環境を提供する．

4.3 マルチコア SH-4A OpenCL

前述の Android 実行環境において, Linux カーネルが動作するコア以外を OpenCL デバイスとする OpenCL 実装である. デバイスの制御には並木研究室で開発している OS 連携機構を用いる．現在，この OpenCL 実装を対象として Android アプリケーションから利用可能な OpenCL ライブラリの試作を行っている．

本研究において試作するライブラリでは，限定された API のみを実装する. OpenCL では基本的にデバイス側のコードを実行時にコンパイルするが，今回は，事前にコンパイルしたものを使用する．デバイスコードのコンパイルには SH アーキテクチャに対応した統合開発環境である High-performance Embedded Workshop を利用し, デバイスコードの実行には OpenCL の仕様に定義されているネイティブカーネルを起動する API を用いる. 今回試作する OpenCL ライブラリではこのネイティブカーネルを起動する API の実行に関わるもののみ実装していく予定である．

5 まとめ

本研究の進捗状況は前述の Android 実行環境 1 を構築したところであり，実行環境 2 を構築している途中である．OpenCL 実装については OS 連携機構とどのように結びつけるか検討中である．

試作システム完成後の課題としては，OS 連携機構による OpenCL 実装やライブラリによって発生すると予想されるオーバヘッドを小さくすることである. また, 現在は試作段階であるため, デバイスで実行されるコードはコンパイル済みのネイティブカーネル関数のみの対応としているが，今後移植性を高めるためにはデバイスコードを実行時にコンパイルするための方法も考慮する必要がある

参考文献

[1] NVIDIA Developer Zone: http://developer.nvidia.com/category/zone/cuda-zone (2011)

[2] OpenCL http://www.khronos.org/opencl/ (2011)

[3] 磯部泰徳，佐藤未来子，並木美太郎，マルチコア CPU における OS の資源管理方式の研究, 情報処理学会第 72 回全国大会 (2010)

[4] 佐藤未来子，磯部泰徳，十山圭介，野尻徹，入江直彦，内山邦夫，並木美太郎，”汎用ホモジニアスマルチコアプロセッサにおける OS とスレッドライブラリ”，情報処理学会第 20 回コンピュータシステムシンポジウム (2008)

Android向けOpenCLライブラリの試作

2011 Summer Joint Symposium for Advanced System Software

東京農工大学大学院
工学府情報工学専攻
並木研究室　望月秋人
2011/08/24

目次

- 背景
  - OpenCL
- 目的
- 提案システム
  - プラットフォーム
  - 試作システムのOpenCL実装
- まとめ

背景

- マルチコア・メニーコアプロセッサの増加
  - 組込み機器、特に携帯端末向けの高機能なサービスが増加しているため、高性能なプロセッサが求められている
- ホモジニアス、ヘテロジニアスな並列演算環境
  - HPC分野で利用されてきたような、マルチコアCPU以外の、GPUなどのメニーコアプロセッサの組込み機器へ導入され始めている
- 携帯端末向けアプリケーションが並列演算資源を容易に活用できる環境の不足
  - システム上の制約、アプリケーション開発者の変化

代表的な既存のヘテロジニアスな並列演算環境

- NVIDIA CUDA
  - NVIDIAのGPUを対象としているものなので、現状では組込み向けには利用できない
- AMD Stream
  - CUDAと同じようにAMDの製品のみを対象としているため、利用範囲は限られる
- OpenCL
  - 実装自体はベンダ依存であるが、APIが共通化されているのでソースコードのレベルであれば互換性がある

OpenCL
ー概要ー

- Appleが提唱し、標準化団体Khronosグループが標準化を行っている、ヘテロジニアスプロセッサ環境向け並列コンピューティングのフレームワーク
- 対象はCPUやGPUなどのホモ・ヘテロジニアスプロセッサ
- プログラミングに用いる言語は拡張を加えたISO C99のサブセット
- 組み込み機器向けプロファイルも規定されている

既存のOpenCL実装

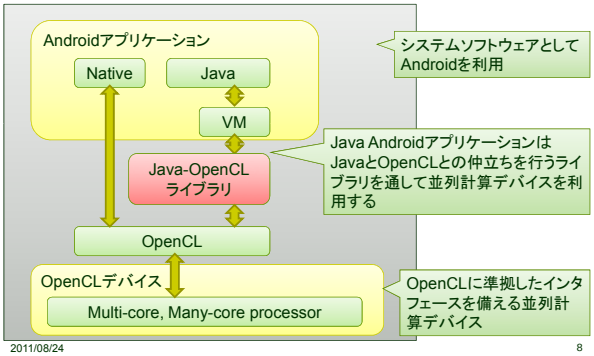
- NVIDIA OpenCL
  - NVIDIA社のCUDA対応GPUが対象
- ATI Stream OpenCL
  - AMD社のStream対応CPU、GPUが対象
- IBM OpenCL Development Kits for Linux on Power (α版)
  - IBM社のブレードサーバ（QS22/JS22/JS23/JS43）が対象、PS3などでも動作
- Intel OpenCL
  - Intel社のCPU（SSE 4.1以降の命令を備えるCore2以降）が対象
- その他
  - Apple OpenCL、Fixstars FOXCなど

目的

- 移植性の高いアプリケーションを並列処理環境で実現するためのライブラリを試作
- 標準化されているヘテロジニアスな並列演算環境であるOpenCLを用いることで、移植性のあるライブラリを実現
- 携帯端末への導入が増加しているAndroid向けに提供することで多くの携帯アプリケーション開発者が並列演算資源を利用できるようにする

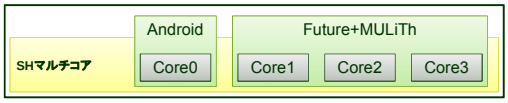
- このライブラリを用いることにより、異なるアーキテクチャにおいても同じコードが再利用可能なプログラムを作成、実行できる
- また、ライブラリとして提供するためVM等に変更を加える必要は無く、導入が容易である

提案システム概要



提案システム概要 ―Android実行環境　1―

- 4コアを備えるSH-4Aアーキテクチャのマルチコアプロセッサを対象とし、そのうちの1コア上で動作するLinuxカーネルをベースとするAndroidを構築
  - Linuxカーネル 2.6.29, Android 2.1
- 残りの3コア（コア1～3）ではFuture+MULiThを動作させ、並列演算デバイスとして利用
  - デバイスの制御には並木研究室で開発しているOS間連携機構を利用
  - AndroidアプリケーションからはJNIを通して利用



Future OS + MULiTh

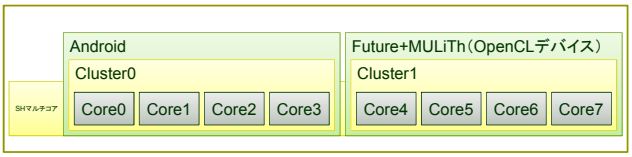
- Future OS
  - CPU資源管理、メモリ管理、I/O管理をシンプルにし、カーネル内処理のオーバーヘッドを最小限にとどめる小型・軽量なOS
- MULiTh（Userlevel Library of Thread for Multithreaded Architecture）
  - Pthreadインタフェースを持つスレッドライブラリ
  - マルチスレッドアーキテクチャを利用したスレッド制御

OS間連携機構

- 汎用OSとマルチコアプロセッサを利用した演算に特化した専用OSを連係動作させる機構
  - 汎用OSとしてLinux、専用OSとしてFuture OS（+MULiTh）を利用
  - LinuxからFutureを利用するインタフェースはデバイスドライバとして実装
  - OS間のデータ通信には共有メモリを利用

提案システム概要 ―Android実行環境　2―

- 8コアのSH-4Aマルチコアプロセッサを対象として、Androidの実行環境を構築
  - 基本的には前述の4コアのものと同じ
  - Linuxをコア0～3で実行、コア4～7をOpenCLデバイスとして利用



提案システム概要 ーマルチコアSH-4A OpenCLー

- デバイスとしてSH-4Aのマルチコアプロセッサを用いるOpenCL実装
  - 4コアのSH-4Aでは、ホストはAndroid（コア0）、デバイスはFuture+MULiTh（Core1～3）
  - 8コアのSH-4Aでは、ホストはAndroid（コア0～3）、デバイスはFuture+MULiTh（Core4～7）
  - デバイスの制御には並木研究室で開発しているOS連携機構を用いる
- このシステムを対象としてAndroidで利用可能なOpenCLライブラリを試作中である

提案システム概要 ー試作ライブラリー

- 実装するAPIは限定的
  - 基本的にOpenCLではデバイス側のコードを実行時にコンパイルするが、今回は事前にコンパイルしたものを利用する
  - デバイスコードのコンパイルにはHigh-performance Embedded Workshopを利用
  - デバイスコードの実行にはOpenCLの仕様に定義されているネイティブカーネルを起動するAPI（clEnqueueNativeKernel）を利用
  - 今回試作するOpenCLライブラリではこのAPIの実行に関わるもののみ実装予定

提案システム概要 ー現状ー

- 現在の進捗は前述のAndroid実行環境1を構築したところ
- 実行環境2は構築している途中である（SMPカーネルがまだ動作していない，1コアであれば動作する）
- OpenCLの実装についてはOS連携機構とどのように結びつけるか検討中
- 現在は試作段階なので、ネイティブカーネルのみの対応であるが、今後移植性を高めるためにはデバイスコードを実行時にコンパイルするための方法も考慮する必要がある

まとめ

- 発表内容
  - 組み込み機器（Android端末）において、ヘテロジニアスな並列演算環境を提供するOpenCLライブラリを提案
  - 試作中のシステムが対象とするOpenCLデバイスは並木研究室で開発しているOS間連携機構を利用したもの
- 今後の予定
  - 実行環境の構築
  - 実行性能の評価
  - 他の実行環境への移植性の評価

SSD ディスクキャッシュシステムの評価

仁科 圭介†

東京農工大学大学院工学府情報工学専攻†

1 はじめに

Flash SSD（Solid State Drive, 以下 SSD）は，HDD（Hard Disk Drive）に比べランダムアクセス性能が高く，省電力であるという特徴を持つ．しかし，SSD の容量単価は高く，HDD のストレージをそのまま SSD へ置き換えるには金銭的コストが大きい．したがって潜在的な要求として，SSD をコスト効率よく計算機システムの高速化や省電力化に利用する方式が求められている．

本研究では，HDD をストレージに用いた既存のシステムに対して，SSD を HDD と主記憶の間のキャッシュとして利用することでストレージ全体の高速化と省電力化を実現することを目的に，実際にシステムを設計，Linux デバイスドライバで実装を行った．本発表では，この SSD ディスクキャッシュシステムの概要とそのオーバヘッドの評価について述べる．

2 SSD ディスクキャッシュシステムの概要

2.1 システム構成

本システムの全体構成を次の図 1 に示す．本システムでは，ディスクキャッシュドライバが，各種ファイルシステムやバッファキャッシュから発行されるブロック I/O 要求を受け取り，SSD を HDD と主記憶間のキャッシュとして利用しながら要求を処理する．本デバイスドライバでの実装により任意のファイルシステムが利用可能である．

2.2 ディスクキャッシュドライバの構成

本ディスクキャッシュドライバでは，図 2 に示すとおり，Linux で提供される “device-mapper”と SSD のキャッシュ管理を行う dm-ssd モジュールで構成される．device-mapper は，システムコールインタフェース，HDD へのブロック I/O 要求を受け取るデバイスファイルの作成，実デバイスへの I/O 発行などを行う．dm-ssd モジュールでは，device-mapper から転送されたブロック I/O 要求を受け取り，device-mapper に実デバイスへの I/O を指示する．

2.3 dm-ssd におけるデータ管理機構

本ディスクキャッシュ管理では，HDD のデータブロックの SSD へのマッピングを管理している．dm-ssd では，マッピングエントリの数を抑えながら，SSD へのデータ割り当てのオーバヘッドを小さくするため，次に示す 2 種類の管理単位を用いている．

- ブロック: SSD へのキャッシュデータ割り当ての単位
- セット: HDD 空間を SSD 空間に対応付ける単位

ブロックは 2 の N 乗セクタ，セットは 2 の M 乗ブロック（N, M は自然数）で指定する．ブロックには，ファイルシステムなどから受け取るブロック I/O 要求の最小単位を指定することで，要求の大きさと同じ大きさでキャッシュ割り当てが可能になり，割り当てによる余分な I/O を削減できる．同時に，複数の連続したブロックで構成されたセット単位でのマッピングを導入することで，マッピングエントリの数を削減できる．

図 3 に本デバイスドライバの要求処理アルゴリズムを示す．HDD 上のセットに対応付けた SSD のセット（キャッシュセット）番号を管理するセット対応表と，キャッシュセット内の各ブロックの状態を管理するキャッシュセット構造体により，SSD 内のキャッシュを管理し，要

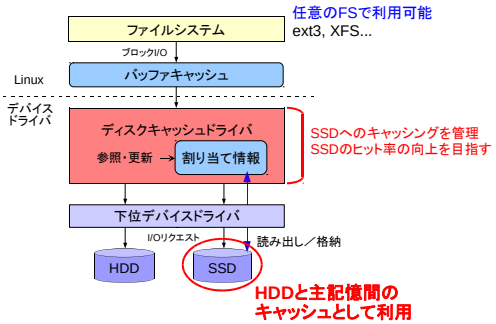


図 1 システムの全体構成

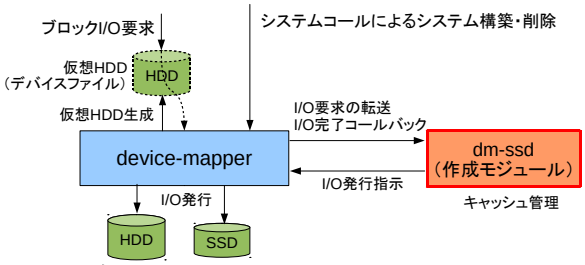


図 2 ディスクキャッシュドライバの全体構成

Evaluation of a SSD-based Disk-cache System

†Keisuke NISHINA

Graduate school of Engineering, Tokyo University of Agriculture and Technology

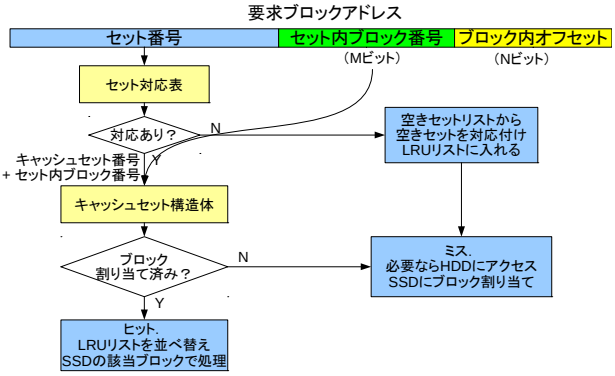


図 3　要求処理アルゴリズム

求がミスならば新たに割り当てを行う．ヒットの場合は，SSD で I/O を行う．

SSD 内の空きキャッシュセットが少なくなると割り当て済みキャッシュセットの中から LRU で選択されたセットが新しく空きセット化され，新たな割り当てに備える．

3 SSD ディスクキャッシュシステムの評価

本章では，SSD ディスクキャッシュシステムにおいて，SSD へキャッシュデータを割り当てる処理と，キャッシュデータを追い出す処理の管理オーバヘッドを評価する．なお，評価に使用したデバイスを表 1 に示す．

3.1 キャッシュ割り当て時の評価

3.1.1 評価方法

本ディスクキャッシュシステムを適用した場合と非適用の場合でストレージデバイスのランダムアクセス時とシーケンシャルアクセス時のスループットと電力を測定し，結果の差を比較する．測定には次の 4 つの I/O パターンを用いた．

r2g: 2GB シーケンシャル読み込み
w2g: 2GB シーケンシャル書き出し
rrand: 4KB ブロックのランダム読み込み
wrand: 4KB ブロックのランダム書き出し

これらの I/O パターンを各ブロックデバイスに実行し，実行時間と I/O 量からスループットを算出する．また，実行中の消費電力を測定し，平均消費電力を算出する．

3.1.2 評価結果および考察

図 4, 5, 6, 7 に，1 つの HDD タイプ（7.2K），3 つの SSD タイプ（SSD32, SSD80, SSD100）および，SSD32, SSD80, SSD100 を 7.2K HDD のディスクキャッシュとして適用した構成（cache32, cache80, cache100）において，r2g，w2g，rrand，wrand を実行した場合のスループットと平均消費電力の測定結果を示す．なお，キャッシュ適用デバイスの平均消費電力については HDD と SSD ディスクキャッシュが一つのストレージデバイスを構成しているとの考え方に基づき，I/O パターン実行時の HDD と SSD の平均消費電力の和を示している．

表 1 評価使用デバイス

15K	600GB 15000rpm 3.5″ HDD
7.2K	1TB 7200rpm 3.5″ HDD
SSD32	32GB SLC SSD
SSD80	80GB MLC SSD
SSD100	100GB MLC SSD

図 4，図 6 の読み出し処理の評価において，SSD をディスクキャッシュとする場合，HDD 上のデータを SSD へ割り当てる際，HDD 以上のスループットが出ることはなかった．また特に，r2g において，SSD の書き出し速度が HDD の読み込み速度よりも低速な cache32，80，のケースでは，HDD よりスループットが低下していた．

一方，HDD の読み込み速度よりも十分に高速に書き出しが行える cache100 のケースでは，ドライバによる速度低下は 10%ほどで，rrand では，50%の低下が見られた．新規 SSD キャッシュ割り当てに付随する管理オーバヘッドがセットごとに発生するため，これらのスループットの低下が起こったと考えられる．ランダムアクセスの場合，セットの割り当てがアクセスのたびに起こるので，このオーバヘッドが大きく影響したと考えられる．

図 5, 7 のキャッシュを適用したデバイスへの書き出しに関しては，SSD へ直接割り当てを行うため，HDD へのアクセスはない．複雑な状態管理も必要ないためオーバヘッドはほとんどなく SSD のスループットが出ている．ただし cache100 に関して図 7 の SSD100 で計測したスループットより明らかに低かった．これについては今後原因を調査する．

3.2 キャッシュ追い出し時の評価

SSD がキャッシュデータで埋まり，空きセットが少なくなると，新しいセットの割り当てに備えるため，LRU によって選択されたセットを空きセットにする．このとき，選択したセットにダーティブロックが含まれていた場合，このブロックを HDD に同期する．この追い出し処理により，I/O 要求のスループットがどれだけ低下するかを測定した．

3.2.1 評価方法

すぐに追い出しが起こるように SSD のキャッシュ領域の容量を 1GB に設定し，2GB 分のシーケンシャル書き出しを行い，スループットの変化を測定した．また，新しい空きセットを作り始める空きセット数の閾値が 10 と 100 の各場合で結果に違いが出るかを確認するため，それぞ

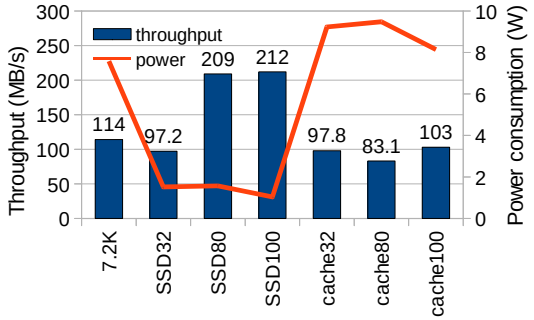


図 4　r2g のスループットと平均消費電力

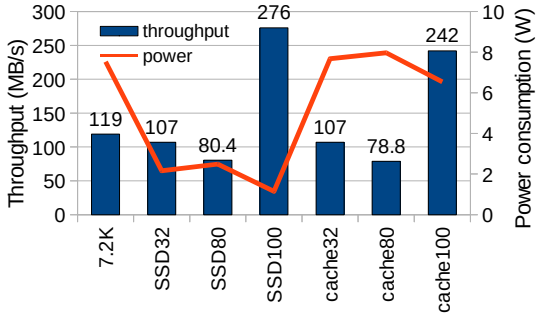


図 5　w2g のスループットと平均消費電力

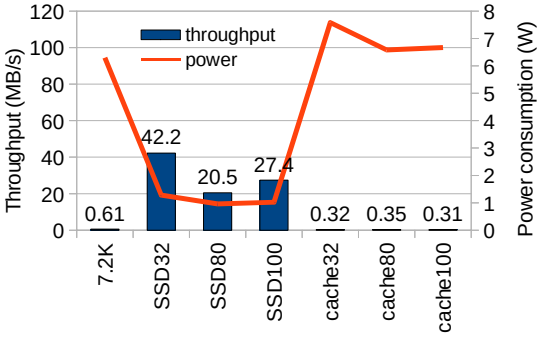


図 6　rrand のスループットと平均消費電力

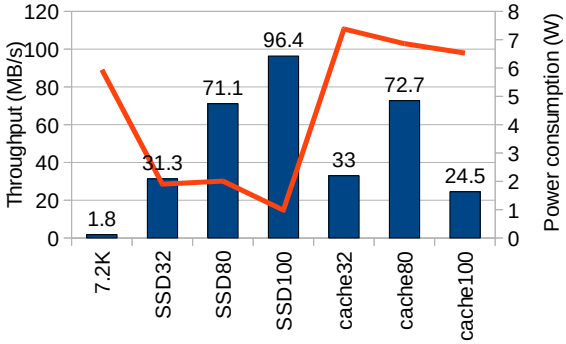


図 7　wrand のスループットと平均消費電力

れの閾値で測定を行った．使用したデバイスは HDD に 15K，SSD に SSD100 を用いた．

3.2.2 評価結果と考察

空きセット数の閾値が 10，100 のときの評価結果をそれぞれ図 8，9 に示す．追い出しが発生してからのスループットを比較すると，閾値 100 は閾値 10 よりも約 7%スループットが向上したが，僅かな性能向上に留まった．

現在の実装では，I/O 要求到着時に空きセット数が閾値以下ならその場で非同期の追い出し処理を行っているため，要求が集中すると追い出し処理と要求処理が混在し，性能が悪化すると考えられる．この改善策としては，I/O 負荷が高いときはあえて追い出しを行わず，負荷が低くなってから追い出しを開始することが考えられる．

4 おわりに

デバイスドライバにより SSD ディスクキャッシュシステムを構築し，write アクセスによる新規キャッシュ割り当て時では SSD の基本性能と同等のアクセス性能を実現できることを確認した．read アクセスではドライバによるオーバヘッドが生じるため，選択的な割り当てによる割り当ての削減が今後の課題である．また，キャッシュ追い出し時の評価では，大きな性能低下が見られたが，閾値調整による性能改善が確認できた．更なる性能改善のため，追い出しアルゴリズムの改良が今後の課題である．

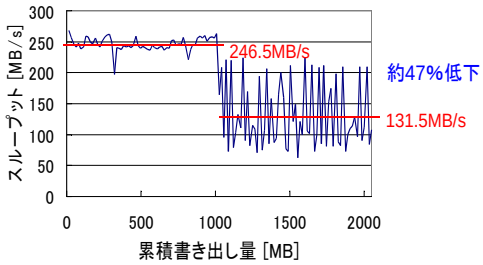


図 8　閾値 10 でのスループットの変化

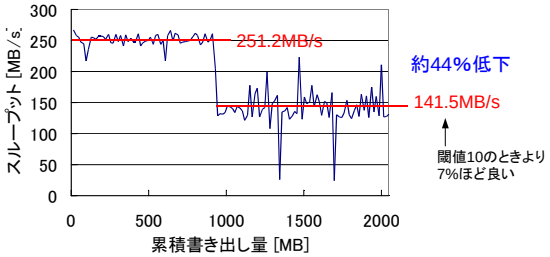


図 9　閾値 100 でのスループットの変化

参　考　文　献

1) J. Matthews, et al.: Intel Turbo Memory: Nonvolatile disk caches in the storage hierarchy of main stream computer systems, ACM Transactions on Storage, Vol.4, No.2, pp.1-24 (2008).
2) Sun Microsystems, Solaris ZFS Administration Guide, Part No: 819-5461 (2009).
3) T. Makatos, et al.: Using Transparent Compression to Improve SSD-based I/O Caches, Proceedings of the 5th European conference on Computer systems (2010).

SSDディスクキャッシュシステムの評価

東京農工大学 大学院 工学府
情報工学専攻 並木研究室
仁科 圭介

本発表の概要

- 本研究について
  - 背景
  - 既存技術・研究の問題点
  - 研究の目標とシステム設計方針
  - SSDディスクキャッシュシステムの概要
  - 実装
- 評価
  - 評価方法
  - 評価結果
  - 得られた知見と改善案

本研究の背景

- SSDストレージデバイスの登場
  - 小さいアクセスレイテンシ（0.1ms << HDD: 4〜12ms）
  - 高スループット
  - 省電力（最大2W程度）
- しかし、一方で…
  - 現在でも容量単価はHDDの数十倍以上
  - 書き換え回数に寿命が存在
- そのままHDDと置き換えるには金銭的なコストが大きい
  →SSDのコスト効率の良い利用法が課題

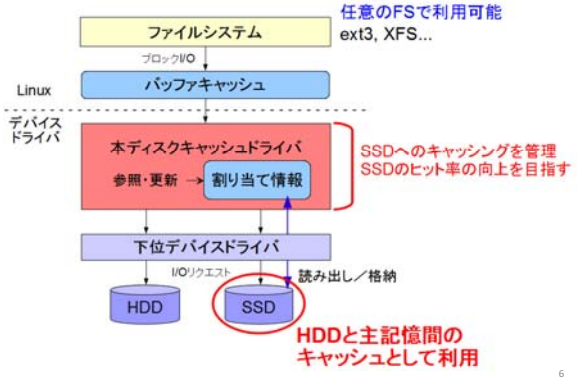
関連技術・研究

- Intel Turbo Memory (ITM) / Intel Smart Response Technology (ISRT)
  - 専用のSSDデバイスをディスクキャッシュとして用いてアクセス高速化と省電力化を図る
  - 利用可能なハードウェアやOSが限定的
- ZFSファイルシステム
  - ディスクブロックのキャッシュ用の領域にSSDを指定可能
  - 他のファイルシステムでは利用できない
- Flaz (‘10, T. Makatos, et al.)
  - ディスクブロックを圧縮してSSDにキャッシュすることでSSDの容量効率を上げる
  - オンラインで圧縮・解凍するためある程度のCPUリソースが必要

本研究の目標とシステム設計方針

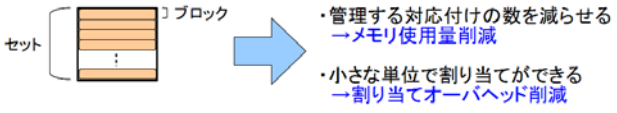
- 目標 – SSDを効率的に用いてディスクアクセスの高速化と消費電力削減を実現
- 方針1 – SSDをHDDのディスクキャッシュとして利用
  - 多くのアクセスをSSDで処理することで アクセス高速化と消費電力削減を目指す
- 方針2 – 既存のシステムに柔軟に適用できるようにファイルシステム／デバイス独立な実装にする
  - Linuxのブロックデバイスドライバにより実現
  - デバイスドライバなので他のOSでも本方式を適用可能

本ディスクキャッシュシステム構成図



キャッシュ管理の単位 ブロック，セット

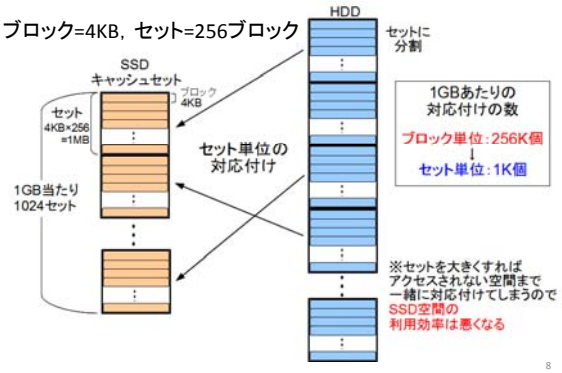
- ブロック･･･実際のキャッシュ割り当て単位
  - 固定サイズ（　セクタ）
  - ページサイズやファイルシステムのブロックサイズが適当
- セット･･･HDD-SSD空間の対応付けの単位
  - 固定サイズ（　ブロック）
  - 「キャッシュセット」: SSD内のセット
- N, M はディスクキャッシュシステム構築時に指定



7

「セット」による対応付けの例

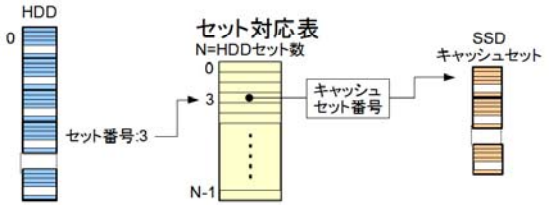
- ブロック=4KB，セット=256ブロック



8

セット対応付け管理

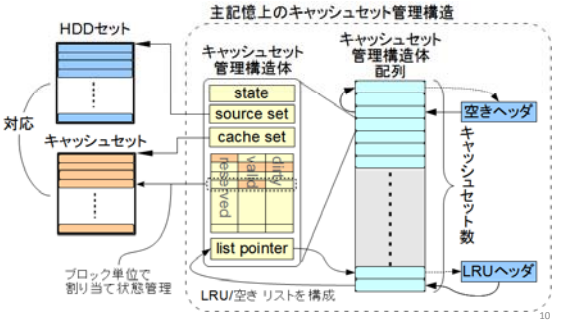
- 「セット対応表」: HDDのセットとキャッシュセットの対応をディスクキャッシュドライバで管理



9

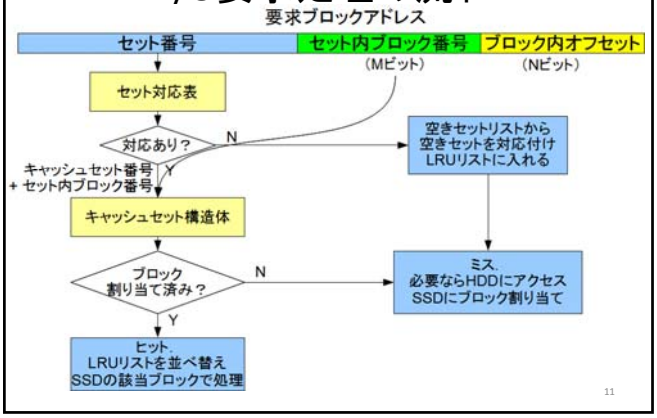
キャッシュセット管理

- セット内の各ブロックの割り当て状態を管理



10

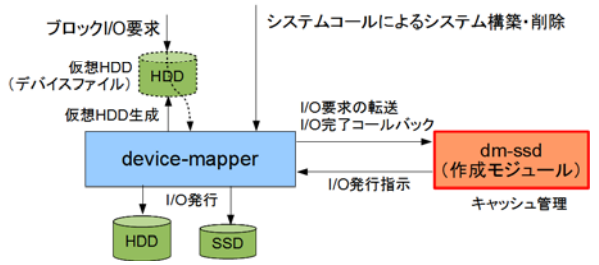
I/O要求処理の流れ



11

実装

Linuxの論理ボリューム管理機能を提供する device-mapperを用いて実装



12

評価

- これまでの評価でわかっていること
  - 要求がSSDにヒットすれば，アクセス速度はSSDの性能にほぼ一致する
- 今回の評価
  - SSDへのキャッシュ割り当てのオーバヘッドを測定
  - キャッシュ置換時のダーティデータ追い出しのオーバヘッドを測定

13

評価方法

- SSDディスクキャッシュのキャッシュ割り当て時の性能と消費電力を測定し、各ストレージデバイスと比較する
  - Linuxのダイレクト I/Oインタフェースを用いて次の4種類のI/Oパターンを実行し，スループットと消費電力を測定

r2g	2GBの連続領域に読み込み
w2g	2GBの連続領域に書き出し
rrand	4KB単位のランダム読み込み
wrand	4KB単位のランダム書き出し

使用SSD

デバイス	概要
SSD32	32GB SLC
SSD80	80GB MLC
SSD100	100GB MLC

使用HDD

デバイス	概要
15K	600GB 15000rpm
10K	300GB 10000rpm
7.2K	1TB 7200rpm
5.4K	160GB 5400rpm

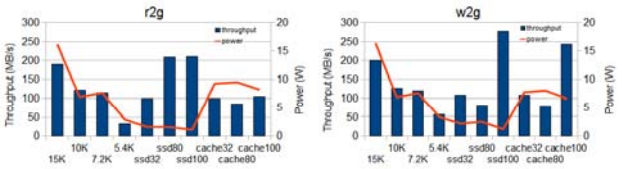
14

SSDキャッシュの設定

- キャッシュ割り当てポリシー
  - R/Wミス時割り当て，Rミス時のみ割り当ての2種類を測定
- キャッシュ置換ポリシー
  - 空きセット数が閾値を下回るとLRUにより選択したセットを新しい空きセットにする
  - キャッシュ割り当てオーバヘッドの測定ではキャッシュセットの置換は発生していない
- ブロック：8セクタ（4KB），セット：256ブロック（1MB）
- 今回は7.2K（1TB 7200rpm HDD）のディスクキャッシュとして適用させて測定

15

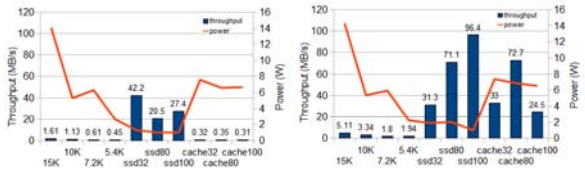
測定結果 – r2g, w2g



- r2g: スループット＜SSDの書き出し速度
  ＜HDDの読み込み速度
- w2g: SSDそのままとほぼ同じ性能

16

測定結果 - rrand, wrand



- rrand: HDDよりも低速．→ドライバのオーバヘッド
- wrand: SSDそのままとほとんど同じ性能

17

割り当てオーバヘッドのまとめ

- 基本的に書き出しミス時の割り当てはSSDにそのまま書き出す場合とほとんど同じ性能が出ており、オーバヘッドは少ない
- 読み込みミス時はHDDから読み込んだデータをSSDに書き出すので，HDDの性能が大きく影響する．また，アトミックな割り当て操作を実現するためにドライバ側で細かい状態遷移を管理しており，その部分でオーバヘッドが生じていると予想される．
- 改善策：割り当てを選択的に行うポリシーなどで、割り当ての総数を減らす

18

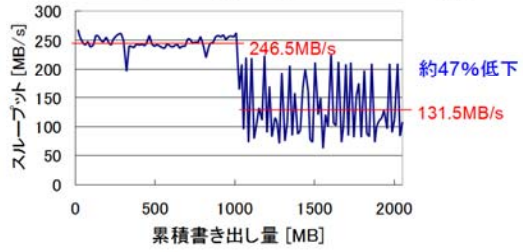
評価: ダーティキャッシュ 追い出しのオーバヘッド

- SSDがキャッシュデータで埋まり，空きセットが少なくなると，新しいセットの割り当てに備えるため，LRUによって選択されたセットを空きセットにする
- このとき，選択したセットにダーティブロックが含まれていた場合，このブロックをHDDに同期する
- この追い出し処理により，I/O要求のスループットがどれだけ低下するかを測定した
- 測定環境
  - HDD: 3.5インチ 15000rpm 600GB
  - SSD: MLC 100GB
  - SSDのキャッシュ領域をおよそ1GB，1セット=1MBに設定

19

評価: ダーティキャッシュ 追い出しのオーバヘッド

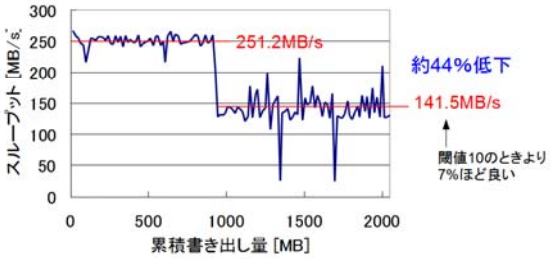
空きセット数閾値を10に設定し，SSDキャッシュ容量1GBに対して，2GBのシーケンシャルwriteを行った際のスループットの推移



20

評価: ダーティキャッシュ 追い出しのオーバヘッド

空きセット数閾値を100に設定して同じ処理を行ったときのスループットの推移



空きセット数の閾値を10倍してもわずかな性能改善にとどまった

21

追い出し処理の問題点

- 現在の実装では，I/O要求到着時に空きセット数を見て，閾値以下ならその場で非同期の追い出し処理を行っているため，I/O要求が集中すると追い出し処理と要求処理が混在し，性能が悪化する
- 改善策: I/O負荷が高いときは追い出し処理を行わず，負荷が低くなったら処理を開始する

22

今後の課題

- 無駄なキャッシュ割り当てを防ぐ選択的な割り当てポリシーの実装
- キャッシュ追い出しアルゴリズムの改善

23

仮想計算機モニタ Xen における RTOS 向け割込み通知機構

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，携帯端末などの組込みシステムでは，多機能化に伴い，リアルタイム性に加えて，高い機能性の両立が求められる．中でもスマートフォンなどの情報端末では，小型化や省電力化が求められる．従来から組込みシステムで利用されるリアルタイム OS(RTOS) には，μITRON や VxWorks がある．RTOS 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．一方，高い機能性を提供する OS(高機能 OS) には，Windows や Linux がある．高機能 OS は，豊富な機能提供のために複雑な構成をしており，処理時間の予測可能性は低い．このように，リアルタイム性と高機能性は相反する性質であり，その両立は容易ではない．

リアルタイム性と高機能性を両立する手法として，2 通りの既存手法がある．一つは，RTX[1] に代表される，RTOS と高機能 OS の特性を併せ持ったハイブリッド OS による手法である．しかし，ハイブリッド OS はリアルタイム性と高機能性という相反する性質を持つため，開発は容易ではなく，ソフトウェアコストの増加を招く．もう一つは，OMAP[2] に代表される，単一システムに複数の計算機を搭載し，RTOS と高機能 OS を個別に動作させる手法である．この手法は，結果的にシステムに搭載する計算機の数が増加し，計算機間の通信機構が必要となるため，小型化や省電力化が達成できない．

これらの問題を解決する新たな手法として，仮想化技術により複数の OS を共存させる方法が考えられる．単一の計算機上で RTOS と高機能 OS を共存させ，リアルタイム性と高機能性を両立しつつ，小型化や省電力化を達成できる．また，組込み機器向けプロセッサ市場において，SH4A-MULTI のようにマルチコア化されたものや，Cortex-A15 のように仮想化支援機能を搭載した製品の開発も進んでいる．このように，組込みシステムに仮想計算機モニタ (VMM) を搭載し，複数の仮想計算機 (VM) を実現することも現実的になっている．

これに伴い，RTS-Hypervisor[3] や SPUMONE[4] などのリアルタイム性を意識した VMM の開発が行われているが，これらは，ゲスト OS に全ての計算機資源を排他的に割り当てる必要があったり，ゲスト OS 間の保護が実現できていないという問題がある．以上の背景から，既存の VMM である Xen[5] を拡張し，リアルタイム性を保証する必要がない資源の共有や，ゲスト OS の保護を実現しつつ，RTOS と高機能 OS の同時動作を可能とする Natsume-Xen(Natsume) を開発している．

以下，2 章で関連研究について述べ，3 章で Xen の概要とその資源管理について述べ，4 章で提案機構である RTOS 向け割込み通知モデルについて述べる．そして，5 章で評価について述べ，6 章で本論文をまとめる．

2 関連研究

RTOS がゲスト OS として動作することを想定したプラットフォームの開発は，既存の研究においても行われている．単一の計算機上で，RTOS と高機能 OS を共存させる研究として，RTS-Hypervisor[3] や SPUMONE[4]，MobiVMM[6] が挙げられる．

RTS-Hypervisor[3](RTS) は，Intel VT が利用可能な環境を対象とした，仮想計算機環境である．RTS では資源をゲスト OS に対して排他的，かつ直接割り当て，デバイスドライバからの直接操作を可能にする．また，RTS はシステムの初期化に必要な機能のみを有し，ゲスト OS の起動後は，仮想ネットワークの提供を除いて，全ての機能を停止する．これにより，処理の遅延をほぼゼロに抑え，リアルタイム性を保証している．しかし，RTS は全ての資源を排他的に割り当てる必要があるため，OMAP などの複数の計算機を搭載する手法と同様に，小型化や省電力化が困難になるという問題を有する．

SPUMONE[4] は，準仮想化型の軽量な VMM である．SPUMONE は資源のうち，CPU のみを仮想化し，複数のゲスト OS を動作させる．さらに，割込みとゲスト OS を固定的に対応付けることで，割込みを特定のゲスト OS が占有することを可能にする．しかし，ゲスト OS 間における資源の共有をサポートしないため，ゲスト OS 毎に資源を用意する必要がある．すなわち，資源管理については，RTS と同様の問題を有する．また，SPUMONE は CPU 以外の仮想化を行わないため，ゲスト OS のメモリの保護を実現できない問題を有する．

MobiVMM[6] は，携帯電話に特化した準仮想化型 VMM である．MobiVMM は，RTOS が動作する RTVM と，高機能 OS が動作する Best-effort VM を提供し，効率的で柔軟な CPU スケジューリングを実現する．また，割込み通知処理には擬似ポーリング I/O 方式を用いる．この方式は，割込みが発生した際，ゲスト OS へ割込みを転送せず，VMM 内のフラグをセットする．そして，ゲスト OS からフラグの変化をポーリングすることで，割込みの検知を実現する．これにより，他のゲスト OS に対する割込み負荷の影響の抑制を図っている．しかし，

この方式では，ポーリングする間隔がCPUスケジューリングの間隔に影響されるため，リアルタイム性が十分であるとは考えにくい．また，ゲストOSがこの方式に対応する必要があり，既存の資産の流用が困難である．

以上で述べたような既存研究における問題点を解決するために，Natsumeでは既存のVMMとして実績のあるXenにリアルタイム性を付加するアプローチを採る．Xenを基にすることで，計算機資源の柔軟な割当てやゲストOSの保護の実現，豊富なソフトウェア資産やノウハウの流用による効率的なシステム開発が可能になる．

3 仮想計算機モニタ Xen

3.1 概要と優位性

Xenはオープンソースソフトウェアの VMM である．Xenは1章や2章で述べた問題を解決する基盤として，以下の理由で優れる．

- **計算機資源の柔軟な割当て**
  VMM主体の資源管理機構により，VM間で柔軟な資源割り当てを実現できる．リアルタイム性に影響しない資源を共有することで，システムの小型化や省電力化の実現に寄与できる．
- **ゲストOS間の保護の実現**
  Xenでは，VMに提供されるメモリ空間は独立しており，MMUにより保護される．これにより，高機能OSの不具合からRTOSを保護できる．
- **豊富なソフトウェア資産やノウハウの存在**
  Xenには，対応済みのOSや管理ツールなど豊富なソフトウェア資産が存在する．また，既存のOSをゲストOSとして動作させるノウハウ公開されており，ソフトウェアの開発コストを削減できる．

3.2 RTOSと資源管理

VMでRTOSを動作させる際，以下の資源におけるリアルタイム性確保が必要になる．以下，リアルタイム性が必要な資源はRTOSが占有し，それ以外は各OSでの共有を前提として，Xenの資源管理について検討する．

■CPU資源 Xenが提供するXenToolsにより，ゲストOSによるCPUの占有を実現できる．しかし，ゲストOSがアイドルになった場合，CPUはIdleDomainと呼ばれる仮想VMに遷移する．この際，割込みが発生すると，再割当てに伴う遅延が発生する．これは，RTOSのリアルタイム性の保証を困難にする．

■主記憶資源 Xenの主記憶管理機構であるdirect-modeは，MMUへのアクセスを管理しつつ，MMUによる保護を行い，ゲストOSが主記憶に直接アクセスすることを可能にする．これにより，メモリアクセスにおいて，仮想化による遅延は発生しない．

■入出力資源 XenはPCI Pass-throughと呼ばれるI/O管理機能を提供する．これは，ゲストOSにPCIデバイスを排他的に割り当て，直接アクセスを可能にするものである．しかし，デバイスから発生する割込みは，イベントチャネルを経由して通知される．イベントチャネルは，割込み発生元に関わらず，全ての割込みを平等に扱い，処理を直列化する．結果，割込みによる負荷がかかっている場合，RTOSに優先して通知すべき割込みが発生しても，即座に通知できない．これは，割込みの遅延の原因となり，リアルタイム性の保証を困難にする．

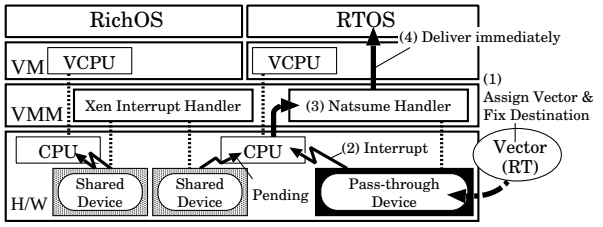


図1 RTOS向け割込み通知モデル

これらの解決には，CPUをRTOSに固定し，イベントチャネルから独立した割込み通知を実現する必要がある．そこで，XenにRTOS向け割込み通知機構を追加し，Natsume実現を目指す．

4 RTOS向け割込み通知モデル

提案モデルは，デバイス毎に独立した割込み通知機構を提供する．これにより，他のデバイスの割込みによる負荷の影響を抑制する．提案モデルの特徴を図1に示す．

1. 共有デバイスと占有デバイスにおける割込み通知機構を分離するために，デバイスを割込みレベルで明確に分離する．
2. 占有デバイスから通知される割込みを，既存の割込みハンドラから独立した専用の割込みハンドラ(Natsumeハンドラ)でハンドルする．
3. Natsumeハンドラを，既存の割込みハンドラより高い優先度で実行する．これにより，他の割込みによる負荷の影響を受けず，即座に割込みをゲストOSへ転送する．
4. ゲストOSに，CPUを固定することで，Natsumeハンドラから転送された割込みを即座にゲストOS内の割込みハンドラで処理することを可能にする．

提案モデルの実現にあたって，以下の機構を実装した．

4.1 専用の割込みベクタを割り当てる機構

占有デバイスからの割込みを区別し，Natsumeハンドラで処理するために，デバイス毎に専用の割込みベクタを割り当てる機構を実装した．現在，実装しているx86アーキテクチャでは，ベクタはIRQによって決定され，かつIRQは全PCIデバイスで最大4つに制限される．つまり，専用ベクタの割り当てには，IRQに依存しないベクタ割当て機構が必要になる．具体的には，PCIが提供するメッセージベースの割込みであるMSI(Message

Signaled Interrupts) や，ARM アーキテクチャが提供する VIC(Vectored Interrupt Controller) などの利用が必要がある．今回の実装では MSI を用いる．これにより，既存の機構から独立した割込み通知を可能にする．

4.2 割込み通知先 CPU を固定する機構

発生した割込みを即座に処理するため，割込みの通知先を特定の CPU に固定する機構を実装した．通常 Xen は，割込みを最も低負荷の CPU に通知する．しかし，ゲスト OS がその CPU を利用しない場合，割込み転送による遅延が発生する．そこで，通知先を RTOS が占有する CPU に固定し，割込み通知の遅延を防止する．

4.3 Natsume ハンドラ

既存のハンドラより高優先度で動作する軽量な割込みハンドラを実装した．具体的には，占有デバイス以外の割込みに関する処理や，割込みの共有に関する処理を削除し，軽量化・最適化を施した．

4.4 CPU の再割当てを防止する機構

CPU が IdleDomain へ遷移することを抑制するため，CPU の再割当てを防ぐ機構を実装した．具体的には，ゲスト OS 上で最低優先度で動作する CPU バウンドなプログラムを実装した．

5 評価

5.1 目的

提案モデルにより，他の VM での割込み負荷に関わらず，RTOS における割込みの遅延や，処理時間の揺らぎを抑制できれば，リアルタイム性の保証に有効だといえる．そこで，あるゲスト OS(負荷用 OS) に割込み負荷をかけ，別のゲスト OS(評価用 OS) に対する割込み通知に影響が無いことを確認する．具体的には，以下の実験において，負荷用 OS の数を 0〜2 に変化させ，割込み負荷の変化が計測用 OS へ与える影響を観察した．

1. **Xen**: Xen 元来の割込み処理性能を明確にするための実験を行う．具体的には，負荷用 OS と評価用 OS に NIC(負荷用デバイス，評価用デバイス) を割り当て，負荷用 OS に大量のパケットによる割込み負荷をかける．同時に，評価用 OS に 10000 個のパケットを送信し，それを契機とした割込みの通知に要する時間 (割込み通知時間) を計測する．割込み通知時間は，VMM の割込みハンドラと，ゲスト OS 内の割込みハンドラで TSC(Time Stamp Counter) の値を取得し，その差分をとることで算出する．
2. **Natsume-CPU**: Xen に，提案手法のうち CPU の再割当てを防止する機構を加えたものについて，その効果を明確にするために (1) と同様の実験を行う．
3. **Natsume-Vector**: Xen に，提案手法のうち専用ベクタを割り当てる機構と Natsume ハンドラを加えたものについて，その効果を明確にするために (1) と同様の実験を行う．
4. **Natsume-ALL**: 提案手法すべてを適用した Natsume-Xen において，(1) と同様の実験を行う．

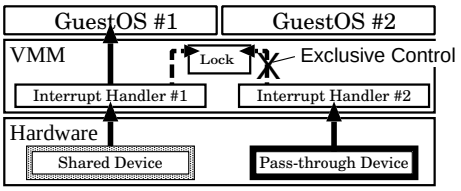


図 2 ロックを共有した割込みハンドラ

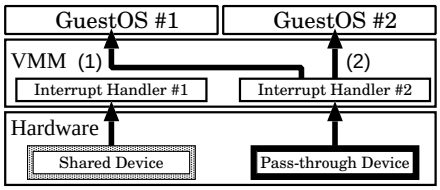


図 3 全ゲスト OS への割込み通知

5.2 性能評価用 Xen の構築

評価用デバイスの割込み通知時間を計測するためには，既存の割込み通知機構から処理を分離する必要がある．また，割込みを受信した CPU が評価用 OS が利用する CPU と異なる場合，TSC のソースが異なるため，VMM とゲスト OS で取得した値から，割込み通知時間を算出できなくなる．すなわち，オリジナルの Xen を用いた実験を行う場合，正確な時刻に外部から割込みを発生させる手段が必要になり，実験は容易ではない．そこで，提案機構の一部である専用の割込みベクタを割り当てる機構と割込み通知先 CPU を固定する機構をオリジナルの Xen に適用した．さらに，これらを適用しつつ，オリジナルの Xen を用いた場合と同等の評価を行う目的で，図 2 と図 3 に示す，評価用デバイスと負荷用デバイスで IRQ を共有する状態を再現する機構を追加した評価用 Xen を用意した．

- **割込みハンドラのロックを共有する機構 (図 2)**
  イベントチャネルでは，同一の IRQ に対応する割込みハンドラは，割込みの直列化のために排他制御して実行される．この現象を再現する目的で，評価用割込みハンドラと負荷用デバイスの割込みハンドラ間でロックを共有する機構を実装した．
- **割込みを全てのゲスト OS に通知する機構 (図 3)**
  複数のデバイスで IRQ が共有されている場合，割込みをハンドルした時点では，割込み発生元のデバイスを特定することはできない．そのため，複数のゲスト OS に，IRQ を共有したデバイスを割り当てた場合，IRQ を共有しているデバイスを持つ全てのゲスト OS に対して割込みが通知される．この現象を再現する目的で，評価用デバイスから割込みが発生した場合，負荷用ゲスト OS に対しても，割

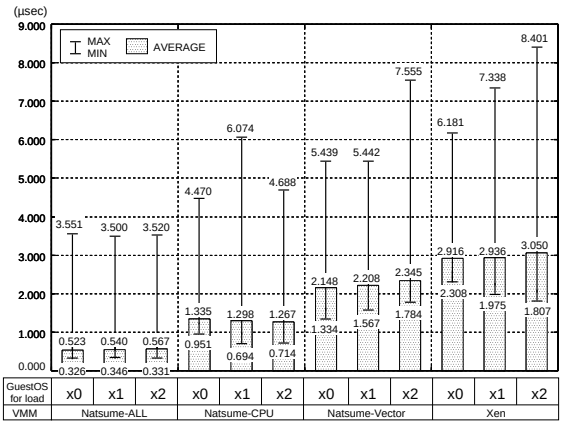


図 4　平均・最小・最悪実行時間

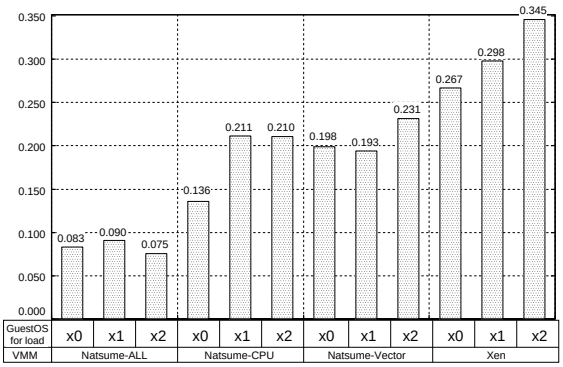


図 5　標準偏差

込みを通知する機構を実装した．なお，負荷用デバイスから発生した割込みを評価用ゲスト OS に転送する機構は，評価用ゲスト OS における割込み発生時刻の計測が困難になるため，実装していない．

これらは，余分な条件分岐やデータの探索などが発生しない様に実装している．また，評価用 Xen は，割込み通知先 CPU を固定する機構が適用されている点，負荷用デバイスから発生した割込みが評価用 OS へ転送されない点において，オリジナルの Xen と比較して性能が向上していると考えられる．すなわち，評価用 Xen と比較して性能の改善が確認できる場合，オリジナルの Xen と比較しても，同様に性能の改善が確認できる．

5.3　実験結果と考察

実験結果を図 4 と図 5 に示す．図 4 は最悪・平均・最小実行時間を表し，図 5 は標準偏差を表したものである．

図 4 から，Xen は負荷の増加に伴い最悪実行時間が増加するが，Natsume-ALL では大きな変化はなく，最大約 5μsec 高速化できていることが分かる．すなわち，Natsume は Xen と比較して RTOS が利用できる時間が大きく，RTOS を動作させるプラットフォームとして，有効であるといえる．次に，図 5 から，Xen では負荷の増加に伴い標準偏差が増加するが，Natsume-ALL では一定範囲で安定している．つまり，処理時間の揺らぎの抑制という観点からも有効であるといえる．

最後に，機構毎の有効性を考察する．図 4 から，平均実行時間は Natsume-Vector より，Natsume-CPU の方が約 0.8μsec 高速だと分かる．つまり，オーバヘッド削減は CPU の再割当てを防止する機構の効果が大きいといえる．一方，図 5 では，Natsume-CPU は負荷の有無により標準偏差が増大するが，Natsume-Vector は一定範囲で安定することが分かる．すなわち，処理時間の揺らぎの抑制効果は，専用ベクタと Natsume ハンドラの効果が大きく，提案モデルのアプローチである，割込み通知の占有はリアルタイム性に有効であるといえる．

6　おわりに

本論文では，組込みシステムにおけるリアルタイム性と高機能性の両立という要求に対して，仮想化技術を活用し，単一計算機上で RTOS と高機能 OS の同時動作を実現する，仮想計算機モニタ「Natsume-Xen」を提案した．また，Natsume-Xen の実現に向けて，ゲスト OS による割込み通知機構の占有を可能にする，RTOS 向け割込み通知機構の設計と実装を行い，評価用 Xen を用いた性能評価を行った．その結果，Natsume-Xen の基である Xen と比較して，平均実行時間が最大約 85%，最悪実行時間が最大約 60%，最小実行時間が最大約 76%の性能改善を確認し，処理時間の揺らぎも約 21%まで削減することを確認した．また，実装した機構別に評価を行い，提案機構のアプローチである割込み通知の占有は，割込み負荷による影響を抑制に寄与しており，リアルタイム性の保証において有効であることを確認した．

参考文献

[1] Bill Carpenter, Mark Roman, Nick Vasilatos, and Myron Zimmerman. The RTX Real-Time Subsystem for Windows NT. *In Proceedings of the USENIX Windows NT Workshop*, pp. 33–37, 1997.

[2] Texas Instruments. OMAP Technology. `http://focus.ti.com/general/docs/gencontent.tsp?contentId=46946`, 2011.

[3] Gerd Lammers and Real-Time Systems. Embedded Real-Time Virtualization and Partitioning. *Small Form Factor Boards Conference*, 2009.

[4] Wataru Kanda, Yu Yumura, Yuki Kinebuchi, Kazuo Makijima, and Tatsuo Nakajima. SPUMONE: Lightweight CPU Virtualization Layer for Embedded Systems. *Embedded and Ubiquitous Computing, IEEE/IFIP International Conference on*, 2008.

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

[6] Seehwan Yoo, Yunxin Liu, Cheol-Ho Hong, Chuck Yoo, and Yongguang Zhang. Mobivmm: a virtual machine monitor for mobile phones. *MobiVirt '08, Proceedings of the First Workshop on Virtualization in Mobile Computing*, pp. 1–5, 2008.

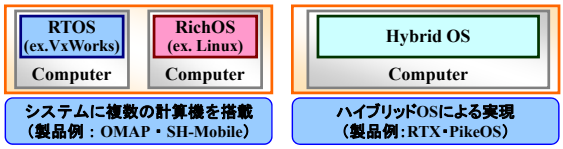
仮想計算機モニタXenにおける RTOS向け割込み通知機構

立命館大学大学院　理工学研究科
毛利研究室
M2
渡邉　和樹

2011/10/3

研究背景

- 携帯端末などの組込みシステムにおける要求
  - リアルタイム性と高機能性の両立
  - 製品の小型化や省電力化の達成
- 既存の解決手段



- 既存手段の課題点
  - 小型化や省電力化の達成が困難
  - リアルタイムタスクと高機能タスク間の保護の問題

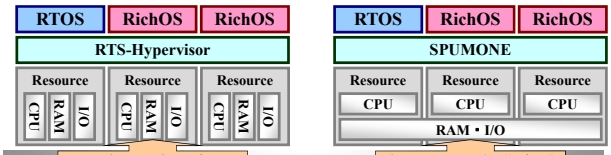
解決へのアプローチ

- 組込みシステムへの仮想化技術の適用
  - 組込み向けプロセッサ市場の動向
    - マルチコア化（例：Cortex-A7 MPCore・SH4A-MULTI）
    - 仮想化支援機能の搭載（例：Cortex-A15）
- 単一の計算機でRTOSと高機能OSを共存
  - リアルタイム性と高機能性の両立
  - 部品数減少による省電力化・小型化の達成

仮想化技術により2種類のOSを共存させるシステム “Natsume-Xen” の研究と開発

関連研究（1/2）

- RTS-Hypervisor
  - 全資源を排他的に割り当てる
    - 初期化後はVMMは経由しない
    - 処理の遅延をほぼ 0 に抑制
  - 主記憶保護はIntel-VTで行う
  - 既存手法と同様に，小型化や省電力化が困難
- SPUMONE
  - CPUのみを仮想化する
    - 割込みとCPUを対応付け，資源占有を実現（共有は不可）
  - 主記憶の保護ができない
    - ゲストOSが他のOSを意識して動作する必要がある
  - 小型化や省電力化が困難



関連研究（2/2）

- MobiVMM
  - 全資源を仮想化・共有し，RTOSに優先的に割り当てる
  - 擬似ポーリングI/O方式の採用
    - 割込みをVMMに溜め込み，OSがポーリングして回収
    - VMMの仕組みをシンプルにし，割込み負荷の影響を抑制
    - ポーリング間隔が，CPUスケジューラに影響される
  - 既存のソフトウェア資産やノウハウの流用が困難
    - RTOS・高機能OS共にチューニングなどが必要

− RT-VMM に求められる性質 −
- 特定のOSに依存しない，柔軟な資源管理
- 各環境における主記憶・割込みの保護
- 既存のソフトウェア資産を有効活用できる環境

仮想計算機モニタ Xen

- オープンソースで開発されているVMM
- VMMとしての優位性
  - 計算機資源の柔軟な割当て:
    - ゲストOS間における資源の共有が容易
    - 小型化や省電力化に貢献できる
  - ゲストOS間の保護の実現:
    - ゲストOSをDomainと呼ばれる環境として分離する
    - 高機能OSやアプリの不具合からRTOSを保護できる
  - 豊富なソフトウェア資産やノウハウの存在:
    - 高機能OS向けのVMMとして実績を持つ
    - 管理ツールや，対応済みの高機能OSが豊富

リアルタイム性と資源管理

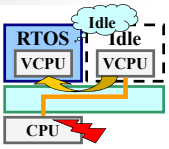
- VM上でRTOSを動作させる場合の課題
  - VM間で資源が共有され，性能が低下する問題
  - 他VMやVMMの負荷に，性能が影響される問題

RTOSの処理時間の予想可能性が低下し，リアルタイム性の確保が困難になる

- リアルタイム性を意識した資源管理が必要
  - リアルタイム処理に必要な資源はRTOSが占有
  - そうでない資源は各OSで共有

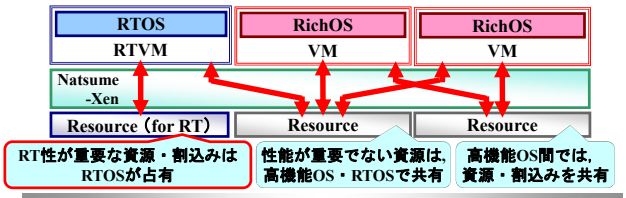
Xenにおける資源管理

- CPU資源→△
  - XenToolsにより，CPUコアの占有が可能
  - ただし，OSがアイドルになると切り離される
  - **割込み発生時，即座に反応できない可能性**
- 主記憶資源→◎
  - direct-mode により，VMMはMMUの監視のみを行う
  - OSによる主記憶アクセス時に実環境以上の遅延はない
- 入出力資源・割込み→×
  - PCI Pass-throughにより，占有が可能
  - 割込みはイベントチャネルで直列化される
  - **RTOSに即座に割込みを通知できない**



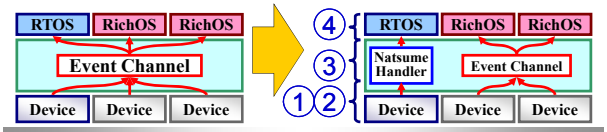
仮想計算機モニタ Natsume-Xen

- 仮想計算機モニタXen を基にした RT-VMM
  - 各タスク間における保護の実現
  - 特定OSに依存しない，VMM主体の資源管理
  - 既存の豊富なソフトウェア資産を利用可能
- RTVMを定義，リアルタイム性を意識した資源管理
  - RTOS向け割込み通知モデルによるリアルタイム性の確保



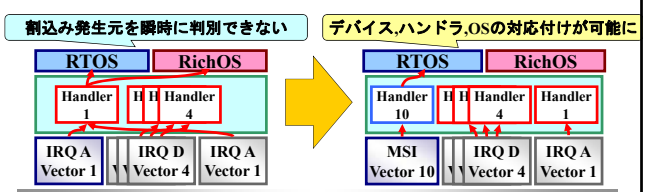
RTOS向け割込み通知モデル

- デバイス毎に独立した割込み通知機構を提供
- 割込み処理の流れ
  1. 占有デバイス・専用割込みハンドラ・RTOSを対応付ける
  2. 割込み通知先CPUをRTOSが占有するCPUに固定
  3. 高優先度で動作する専用ハンドラで即座に割込みを転送
  4. RTOSからCPUが切り離されることを防ぎ，RTOSのハンドラで即座に割込みを処理可能にする



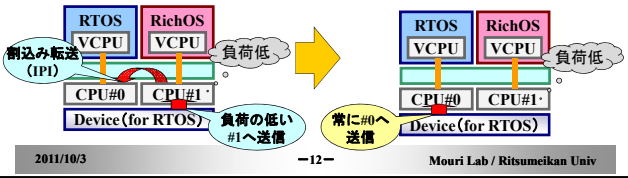
1.デバイス・ハンドラ・OSの対応付け

- デバイスへ専用の割込みベクタを割り当てる
  - 通常，デバイスのベクタは，IRQ番号で決定される
  - PCI の仕様上，割当て可能なIRQには限りがある（最大4つ）
- MSI（Message Signaled Interrupts）を採用
  - PCIが提供するメッセージベースの割込み通知方式
  - IRQを用いない，任意のベクタ割当てを可能にする



2.割込み通知先CPUを固定する機構

- 占有デバイスからの割込みの通知先を常にRTOSが利用するCPUへ固定する
- Xen の標準は Lowest Priority Mode
  - 最も負荷の低いCPUを通知先にするモード
  - 全CPUを共有する場合は，スループットが高い
  - 受信したCPUを通知先OSが利用してない場合，IPIによる割込み転送が必要

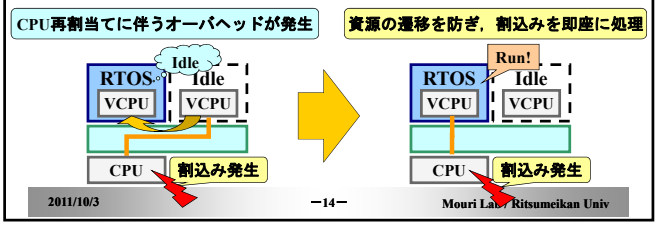


3.Natsume割込みハンドラ

- 既存の割込みハンドラより高い優先度で動作する割込みハンドラ
  - 既存のハンドラより高い優先度を持つ割込みベクタを割り当てることで実現
- 提案モデル・MSIに最適化し，軽量化
  - 割込みの共有に伴う処理を削減
    - ゲストOSによるデバイスの共有（割込み通知先の共有）
    - デバイスによるIRQの共有（割込み発生元の共有）

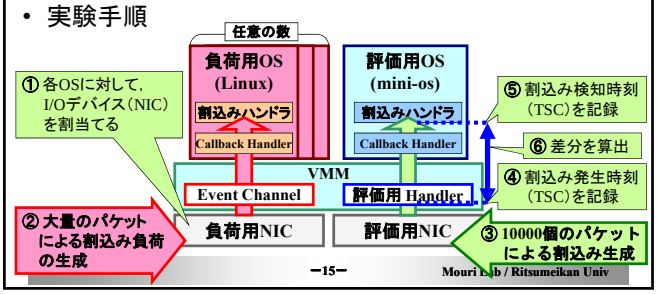
4.CPUの再割当てを防止する機構

- 全プロセス中，最低優先度で動作するCPUバウンドなユーザプログラムを実装
- OSがアイドルになることを防ぎ，CPUがRTOSから切り離されることを抑制する
  - RTOSで即座に割込みを処理することを可能にする



Natsume-Xenの性能評価

- 評価の目的
  - 提案機構により，割込み負荷が割込み通知時間に与える影響を抑制できていることを確認する
    - 割込みの遅延や処理時間の揺らぎの増大を抑制できていれば良い
- 実験手順



評価パターン

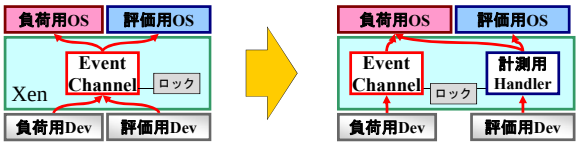
- **実験1（Xen）**：Xen元来の割込み性能を明確にする
- **実験2（Natsume-CPU）**：提案手法のうち
  - CPUの再割当てを防止する機構
  - 割込み通知先CPUを固定する機構

  の効果を明確にする
- **実験3（Natsume-Vector）**：提案手法のうち
  - デバイス・ハンドラ・OSの対応付け（専用ベクタ割当て）
  - Natsumeハンドラ

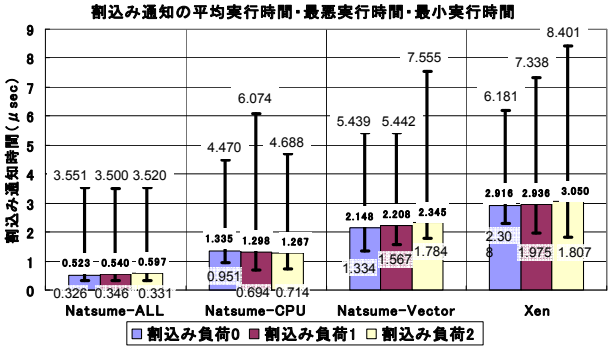
  の効果を明確にする
- **実験4（Natsume-ALL）**：提案手法を全て適用し，Natsume-Xenの割込み処理性能を明確にする

性能評価用Xenの構築

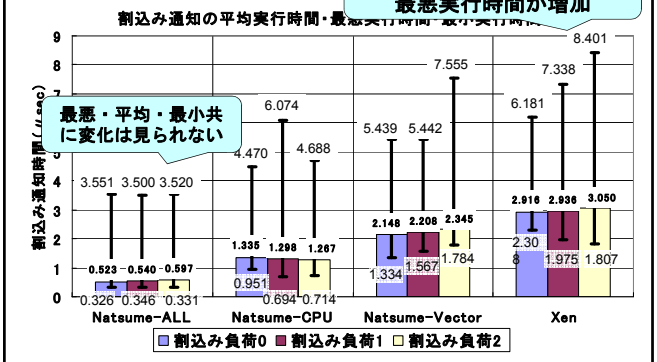
- オリジナルのXenを用いた評価は困難
  - 負荷用と評価用の割込みを区別する必要があるため
- 比較対象として性能評価用Xenを用意
  - 評価用デバイスへ専用の割込みベクタを割り当てる機構
  - IRQを共有している状況を再現する機構
    - 割込みハンドラのロックを共有する機構
    - 割込みを全てのゲストOSに通知する機構



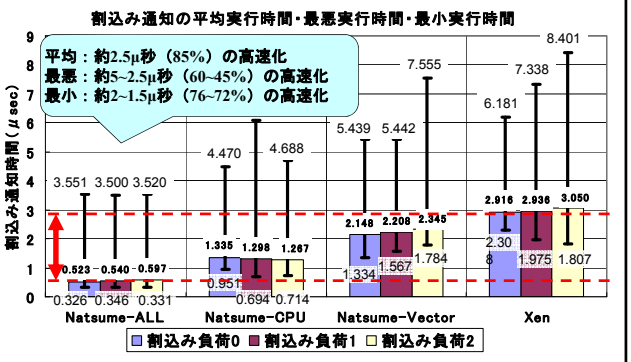
結果（割込み通知時間）（1/4）



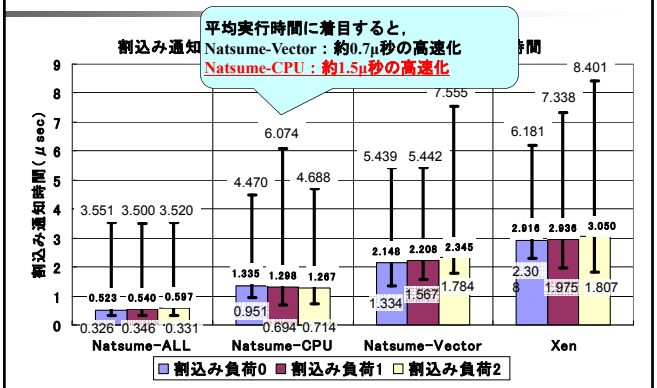
結果（割込み通知時間）（2/4）



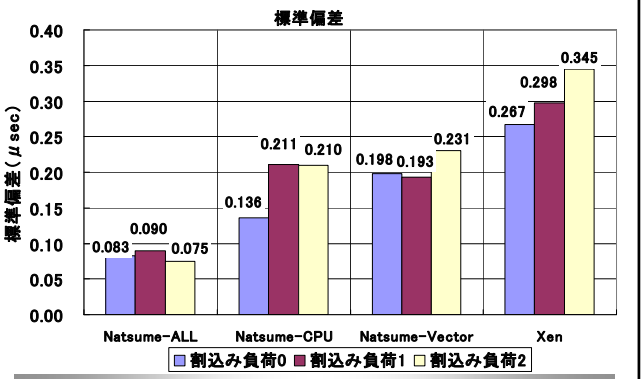
結果（割込み通知時間）（3/4）



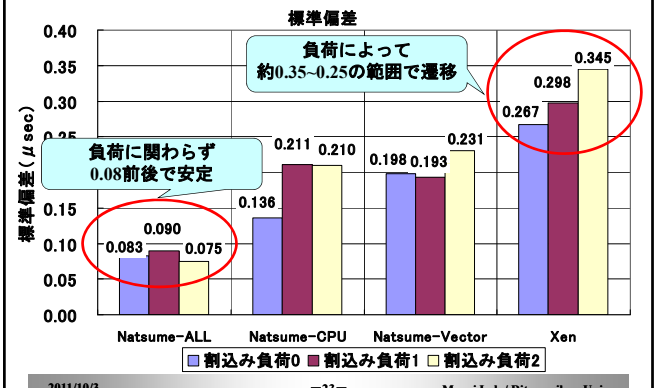
結果（割込み通知時間）（4/4）



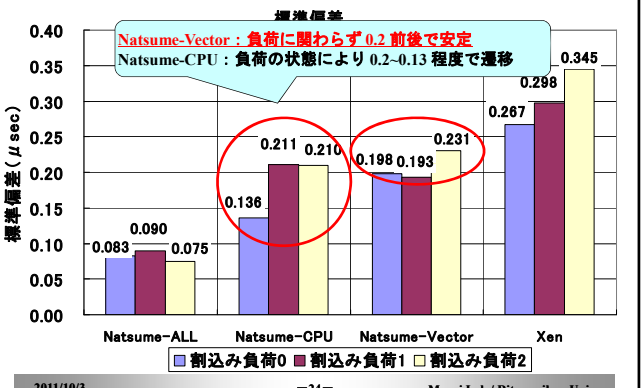
結果（標準偏差）（1/3）



結果（標準偏差）（2/3）



結果（標準偏差）（3/3）



まとめと考察

- 提案機構は割込み負荷の影響を抑制し，リアルタイム性の保証に有効であると考える
  - 負荷の大きさに関わらず最悪実行時間・処理時間の揺らぎ（標準偏差）を抑制できた
  - Natsume-Xenの最悪実行時間（3.520μ秒）は，Xenの平均実行時間（3.050μ秒）の約0.5μ秒差
    - RTOSによるシステム最適化などでリアルタイム性の保証を可能に出来るマージンが大きい
- 今後の課題
  - 消費電力などの観点から，CPUの再割当てを防止する機構の見直し

コグニティブマルチホップ環境における通信制御方式

瀧本 栄二 †　毛利 公一 †
† 立命館大学情報理工学部

1　はじめに

近年の PC，スマートフォンなどでは，複数種類の無線通信デバイスを備えていることが多い．また，利用可能な AP 探索と利用可能周波数帯の検知は，現在でもすでに実現されている．しかし，既存の無線通信は，複数通信デバイスから選択的に使用する通信デバイスを選択するため，周波数資源も含めリソースの有効活用が十分であるとはいえない．そこで，周波数資源の利用状況に応じた通信技術であるコグニティブ通信技術 [1] を採用することで，複数デバイスを並列的に活用し最適な通信をユーザに提供することが期待できる．

一方，ネットワークプロバイダは，ホットスポットを複数配置することで，広範囲にサービスを展開することが可能である．しかし，AP・端末間の通信は，直接電波が到達可能な範囲でのみ可能であるため，屋外などの広範囲でのサービス提供のためには大量のアクセスポイントが必要となる．そこで，マルチホップ通信技術を適用することで，通信可能エリアを拡大することが可能である．また，実際の運用にあたっては，複数のネットワークプロバイダがサービスを展開することが予想される．ユーザが複数のネットワークプロバイダを利用することが可能な場合，それらを同時利用することによってさらなる通信品質の向上が期待できる．

以下，本稿では，複数の無線通信デバイスを搭載した端末を想定し，それらを並列的・効率的に利用し，かつマルチホップ技術による広範囲通信環境を実現するための通信制御方式について述べる．

2　先行研究と想定環境

本稿では，図 1 に示す複合型ネットワーク環境を想定する．想定環境は，GW，AP，端末で構成され，GW・AP 間は有線接続，AP・端末間および端末・端末間は無線接続とする．AP，端末ともに通信性能の異なる無線インタフェースである IEEE802.11a 無線インタフェース（以下，11a インタフェース）と IEEE802.11b 無線インタフェース（以下，11b インタフェース）とを具備する．無線接続部分はアドホックモード接続とし，隣接端末を中継端末，アクセスポイントをシンクノードとするマルチホップネットワークを構成する．

先行研究 [2][3][4] では，待ち行列理論に基づくモデル化，モデルに基づく遅延メトリックによる経路選択と切替えおよびパケット分配方式が提案されている．経路選択と切替えは，複数存在する端末-GW 間の経路から，平均通信遅延が最小となる経路を使用することによる通信遅延の最小化とスループットの向上を実現する．パケット分配方式は，通信速度が異なる複数の通信インタフェースを並列利用する際に，それぞれにかかる負荷，すなわち遅延の割合を平均化することで同様の効果を実現する方式である．先行研究では，経路切替えに関しては単一インタフェースを想定し，GW による集中制御によって実現している．また，パケット分配方式に関しては，通信速度は異なるが通信可能距離が等しい無線インタフェースの使用を想定している．

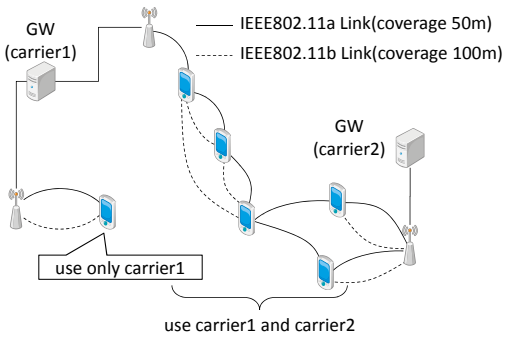


図 1　想定環境

提案方式では，各無線インタフェースの通信可能距離が異なる点で先行研究と異なる．また，マルチホーム環境を想定しているため，GW による集中型制御が困難であることから，先行研究で行ってきた各制御をすべて自律分散型制御で実現することが必要である．以降，本稿では，遅延メトリックに基づく経路制御とパケット分配を自律分散型制御で実現する通信制御方式の詳細について述べる．

3　提案方式

3.1　遅延メトリックと経路制御

提案方式では，平均遅延時間をメトリックとして経路制御を行う [4]．ここで，平均遅延時間は，各インタフェースにおけるパケット送信にかかる遅延時間の平均である．各端末は，パケットを MAC キューにパケットを挿入してから，送信先端末からの ACK を受信するまでの時間を計測し，それに基づいて平均遅延時間を算出する．なお，ACK は，IEEE802.11DCF における ACK

A Communication Control Method on Cognitive Multihop Environments.
Eiji Takimoto†, and Koichi Mouri†
†College of Information Science and Engineering, Ritsumeikan Univ.

を指す.

3.1.1 アップリンク経路の構築

アップリンク経路を作成するためには，GW 情報，次ホップ情報，平均遅延時間，平均遅延時間の総和が必要である．初期時点では，GW がこの情報を制御パケットとして周期的にブロードキャストする．各端末は，このパケットを受信すると，次ホップ情報に自端末情報，自端末の平均遅延時間と受信パケットに含まれる平均遅延時間との和を平均遅延時間総和とした制御パケットを作成し，GW と同様にブロードキャストする．

制御パケットはブロードキャストであるため，複数受信場合がある．そのとき，端末は，制御パケットから得られる平均遅延時間の総和が最も小さい端末を次ホップとして選択することで，最小遅延経路を作成することができる．このように，GW からパケット情報を上書きしながらブロードキャストしていくことで，GW と次ホップのアドレスを全端末に広告することが可能になる．

3.1.2 ダウンリンク経路の構築

上述の制御パケットのブロードキャストによって，アップリンク経路を決定した端末は，GW 宛に経路決定パケットをユニキャスト送信する．これにより，GW に対して自端末の存在を通知する．また，経路決定パケットを中継する端末は，送信元端末の IP アドレスと前ホップとなる端末の IP アドレスとを登録することで，ダウンリンク経路が確率される．

3.2 経路切替え

平均遅延時間は，送信パケット数や周辺端末からの干渉等により，動的に変化する．したがって，最小遅延経路も動的に変化することになる．提案方式では，アップリンク経路作成時に用いる制御パケットで周期的に平均遅延時間と経路全体の平均遅延時間を広告しており，各端末は周期的に最小遅延経路を選択・切替えることでネットワークの状態変化に適応する．経路切替え時は，経路決定パケットを再度送信することでダウンリンク経路も同時に更新する．

3.3 パケット分配

複数の無線インタフェースを備えている端末では，それらにパケットを分配して並列利用することで通信性能の向上が期待できる．想定環境のように通信速度の異なる無線インタフェースを並列利用する場合，パケットの分配比率を 1：1 ではなく通信速度に見合った比率にすることが効率利用の観点から重要である．さらに，干渉等の影響により平均遅延時間は変動するため，その変動に応じて分配比率を変更することが必要である. 提案方式では，文献 [?] で示されたパケット分配特性に基づき，

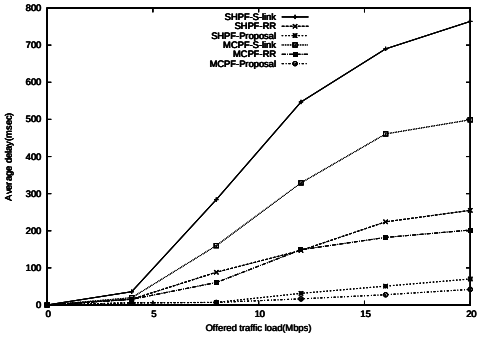


図 2　CBR トラフィックと平均遅延時間

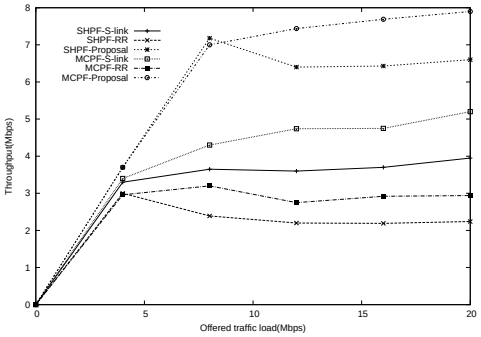


図 3　CBR トラフィックとスループット

各無線インタフェースの平均遅延時間が等しくなるように分配比率を動的に調整することで，最適な分配比率を実現する．

4 シミュレーション評価

現在，提案方式を既存シミュレータにアップリンク経路の構築と，パケット分配機能の実装が完了している．本章では，実装が完了した部分の評価を行った結果について述べる．

シミュレーションは QualNet を用い，GW1 台，AP2 台，端末 98 台の全 101 台をランダムに配置し，各端末が IEEE802.11a と IEEE802.11b の 2 つの無線インタフェースを搭載した．シミュレーションでは，CBR トラフィックを 50 台のランダムに選択された端末から GW へ送信した．提案方式の経路構築およびパケット分配の有効性を確認するため，比較対象として最小ホップ経路選択方式（SHPF）を用い，提案方式（Proposal）が採用する最小コスト経路選択（MCPF）と比較を行った．また，パケット分配の有効性を確認するため，パケット分配を行わないケース（S-Link）．1：1 の分配比率固定のケース（RR）と比較した．

4.1 結果と考察

シミュレーションによる評価の結果，平均遅延時間，スループット共に，提案方式（MCPF-Proposal）によって最も改善されることが示された．同様に，経路構築方

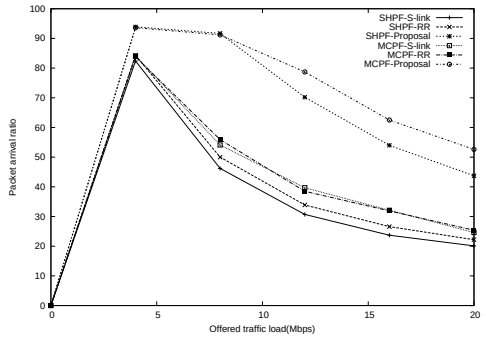


図 4　CBR トラフィックとパケット到達率

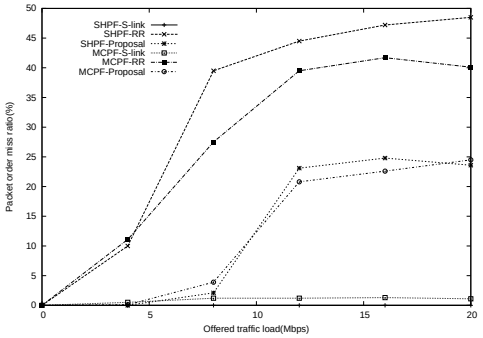


図 5　CBR トラフィックとパケット順序ミス率

式である SHPF と MCPF にそれぞれのパケット分配制御を適用したすべての場合において，平均遅延時間をメトリックとする MCPF の方がネットワークコストを削減することが確認できた．提案方式の平均遅延時間は，RR 適用時の約 20%，S-link 適用時の約 10%まで削減された．スループットについては，同じく約 2.3 倍，約 1.6 倍に向上した．以上から，想定環境において，提案方式はネットワーク性能を向上させる上で有効な手段であるといえる．

一方で，パケット到達率およびパケット順序ミス率の計測を行ったところ，パケット到達率は高いもののパケット順序ミスによるパケットロスが多いことが明かとなった．提案手法では，トラフィック量が 12Mbps 以上でおよそ 25%のパケットロスが発生していた．パケット順序ミスが発生する原因は，経路変更とループ経路である．経路変更によるパケット順序ミスは，大きな順序乱れはないと考えられる．したがって，TCP の順序制御やアプリケーションレベルでの順序制御で十分対応が可能だと考えられる．ループ経路による順序乱れは，最悪の場合パケットの TTL エラーとなる．今回，制御パケットのループ解決を行っているが，データパケットに対するループ解決を行っていない．基本的には，制御パケットの場合と同じ方法で解決できると考えられるため，今後対応を行っていく．

5　おわりに

本稿では，コグニティブマルチホップ環境における，通信制御方式について述べた．提案手法では，端末の各インタフェースごとの平均遅延時間と経路を構成する各端末の平均遅延時間の総和に基づき，最小遅延経路の選択・切替えを行うことによって，適応的に最適な通信経路を利用可能とする．また，実装済みの一部機能について評価を行い，その効果が期待できることを示した．

しかし，パケット順序エラーが多く発生するという新たな課題もでてきている．今後は，残りの機能の実装と共に，その原因であるループや経路切替え時のパケット順序エラーの解決も図っていく．

参考文献

[1] Mitora, III, Y. and Maguire, Jr, G.: Cognitive Radio:Making Software Radios More Personal, *IEEE Personal Communication*, Vol. 6, No. 4, pp. 13–14 (1999).

[2] 滝沢泰久，谷口典之，山中佐知子，山口　明，小花貞夫：コグニティブ無線ネットワークにおけるマルチホップアクセス経路トラフィック制御方式，情報処理学会論文誌，Vol. 48, No. 7, pp. 1234–1248 (2007).

[3] 滝沢泰久，植田哲郎，小花貞夫：IEEE802.11 と IEEE802.16 を用いた複合アクセス経路のパケット分配制御方式，情報処理学会論文誌，Vol. 52, No. 2, pp. 543–557 (2011).

[4] 滝沢泰久，谷口典之，山口　明，小花貞夫：IEEE802.11 と IEEE802.16 を収容する無線アクセスネットワークにおけるパケット分配制御特性，情報処理学会論文誌，Vol. 49, No. 9, pp. 3245–3256 (2008).

コグニティブマルチホップ環境における通信制御方式

立命館大学
助手　瀧本栄二

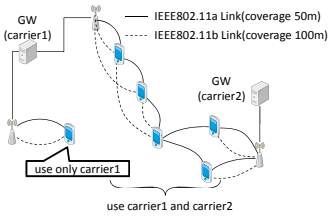
はじめに

- 無線通信技術の発達と普及
  - 無線資源の枯渇という新たな問題
- コグニティブ無線技術
  - 無線資源の有効活用
  - 既存研究：異種技術の組み合わせ（1ホップのみ）
    - Wi-Fi（IEEE802.11）・WiMAX（IEEE802.16）・3G
  - Wi-Fiによるマルチホップ技術との連携に関しては不十分
- コグニティブかつマルチホップ通信技術の研究・開発

はじめに

- コグニティブマルチホップ通信制御
  - 先行研究
    1. 複数Wi-Fiインタフェースの並列利用
    2. 通信遅延に基づく経路選択
    3. 通信遅延に基づくパケット分配
  - 新規研究
    - 距離特性の異なる無線方式の並列化
    - 上記2と3を組み合わせた制御
    - 自律分散化とそれに伴う課題の解決
      - ループ経路・経路の発振

想定環境



- 各端末はIEEE802.11a/b I/Fを持つ
- 複数プロバイダへのアクセスが可能
  - 自律分散制御が必要

通信モデル

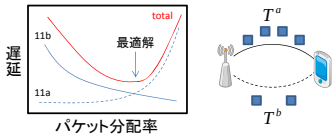
$D = F \cdot T$　D:通信コスト　F:パケット到達率　T:平均遅延時間

1）経路のコストは経路を構成するリンクの和

2）ネットワーク全体のコストは全経路のコストの和

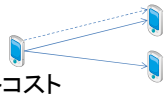
- 平均遅延時間を最小化することでスループットの増加が見込まれる→遅延を最小化する通信制御を行う
- インタフェースの通信遅延特性とパケット分配特性に基づき、パケットを分配して平均遅延時間を削減

$T^a = T^b$のとき最適解



通信コストと遅延メトリック

- リンクコスト
  - 当該リンク（I/F）を使用することで生じるコスト



2つのリンクは同じコストとして考える
異種I/Fによるリンクは別コスト

- 経路コスト
  - 経路を構成する端末群のリンクコストの総和

・最もコストが小さい経路が最適経路
・最小コスト経路＝最小遅延リンクの集合

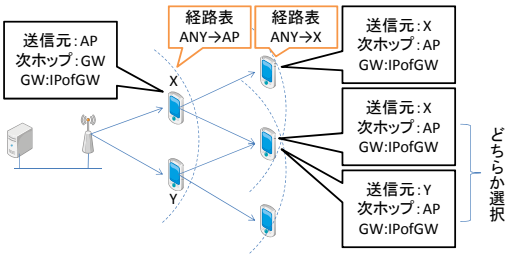
通信遅延をメトリックとして経路選択を行う

経路の構築と切替え

- 初期状態：経路情報なし
  1. APが経路構築用情報をブロードキャスト
  2. 受信端末は情報を更新してブロードキャスト
     - 受信により上り経路が選択可能になる
  3. GWに下り経路通知パケットをユニキャスト送信
  4. GWは応答パケットを送信
     - 下り経路を利用可能
- 周期的に最小遅延経路を再選択→切替え
  - 通信環境への適応的な対応
  - 切替え時にはパケット順序エラーが発生
    - 切替え頻度は抑えるべき課題
- 2つの課題（自律分散の弊害）
  - ループ経路の発生
  - 経路の発振

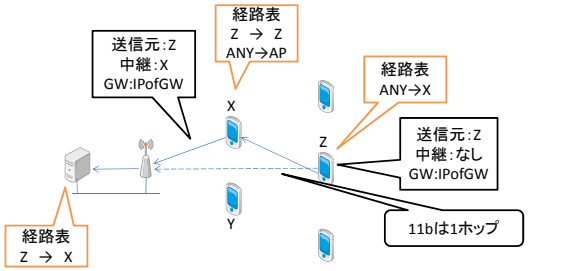
通信経路の構築（上り）

- Notificationメッセージのブロードキャスト
  - 自ノード・経路コスト（遅延）・次ホップ、GWのアドレス
  - 周辺ノードの情報を通知
  - 実際には各インタフェースから別個に送信



通信経路の構築（下り）

- GWに対して経路確定メッセージ送信
  - 自ノード経路コスト・次ホップ・中継ノード情報
- GWは受信後応答メッセージを返信
  - 応答メッセージ受信によって当該経路が利用可能になる（ループ対策）



経路の発振

- トラフィックの集中と分散が繰り返される現象
  - 周辺端末が一斉に経路変更or送信パケット数の多い端末が経路変更
  - 負荷の偏りが発生
  - 次周期に元の状態に戻る
- 遅延増⇔減を繰り返す

対策

1. 経路更新周期に重みを付ける
   - トラフィック小→こまめに経路更新 ⇒ パケット順序エラー発生しやすい課題
     - 経路変更の影響が小さいため
   - トラフィック大→あまり更新しない
     - 経路変更の影響が大きいため
2. 経路変更要求メッセージの利用
   - トラフィックが集中する端末から、集中の原因となる端末を適宜選択して送信
   - 受信端末は可能であれば経路を変更
     - ループまたは選択肢がない場合は切替えず

パケット分配

- インタフェース（A,B）の平均遅延時間

T^A, T^B

が等しくなるようパケット分配率を調整

$T^A > T^B$：T^Aの分配率減少またはT^Bの分配率増加

$T^A < T^B$：T^Aの分配率増加またはT^Bの分配率減少

現状

- 未実装機能
  - 経路応答メッセージ
    - そのためループが発生している
  - 経路変更要求メッセージ
    - 経路の発振が発生している
    - 経路更新周期の調整だけでは効果が薄い

（過去の）シミュレーション評価

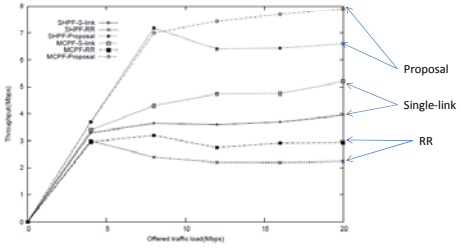
シミュレーションパラメータ	設定値
シミュレータ	QualNet4.5
評価エリア	500m四方
端末数	101台（GW1台・AP2台・端末98台）
端末配置	ランダム配置
シミュレーション時間	310秒
伝搬モデル	2波モデル
最大送信距離＆レート	100m・11Mbps（11b）50m・54Mbps（11a）
アプリケーション	CBR
パケットサイズ	1KB
通信台数	50台
通信レート	80Kbps（4Mbps）・160Kbps（8Mbps）・240kbps（12Mbps）・300Kbps（16Mbps）・380Kbps（20Mbps）

比較対象

- 経路構築：最小ホップ数優先
  - 経路は固定
- パケット分配
  - なし
  - シングルリンク：使用するI/Fを1つに限定
  - あり：ラウンドロビン・適応制御（提案方式）
    （負荷分散あり）
- 上記組み合わせたものと比較

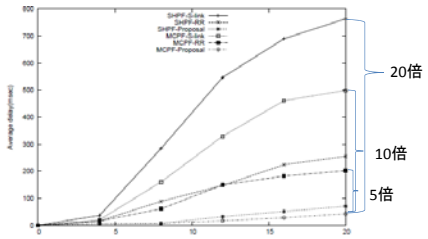
評価結果（スループット）

- 提案手法は約8Mbpsで最も高い値である
- RRよりも単一リンクの方が性能が良い
  - 高速な11aリンクにトラフィックが集約されるため
  - 11aと11bとの性能差が大きいため



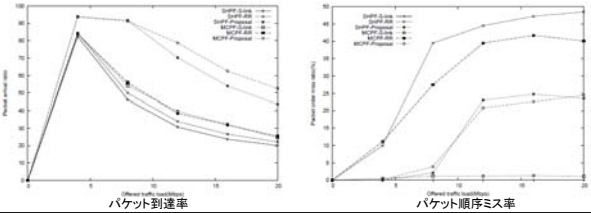
評価結果（遅延）

- 提案手法の平均遅延時間は50ms以下
  - 他の手法は5倍～20倍も遅い
  - パケット分配による改善率が高い
- 遅延の主な原因は11bリンクでの混雑



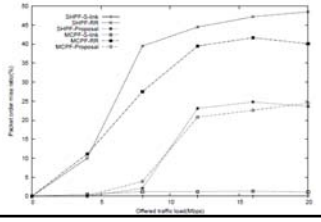
評価結果
（パケット到達率と順序ミス率）

- スループットの低さに比べてPARが高い
- 順序ミスによるパケット破棄が原因
  - 経路変更・パケット分配による遅延差
  - ループによるバースト的な順序エラー



評価結果(到着順序ミス率)

- RR＞提案方式
- パケットは届いても順序ミスで破棄されている
  - CBRはUDPであるため順序補正がない
  - QualNetのCBRはアプリレベルでも補正しない



考察

- 提案方式によって通信性能は向上
  - 遅延20～95%減少、スループット1.5～4倍向上
  - 通信コストのモデルは想定環境でも有効
- パケット順序エラーが大きい
  - 原因:経路変更・パケット分配・一時的なループ
    - 理論的はパケット分配によるエラーは発生しないが頻発
  - TCPまたはアプリでの補正で改善可能
    - どこまで可能か検討が必要
- 11aがあるにも関わらず10Mbps程度が限界
  - 11aリンクを使用できる端末が20台もない
  - 全体的に11Mbpsの11bが選択されている
    - トポロジ・端末の密度の影響が大きい

おわりに

- コグニティブ環境におけるマルチホップ通信の活用とその制御方式
  - 通信遅延をメトリックとした経路制御
  - パケット分配による性能向上
  - 経路の発振を抑えるための制御
  - 一部実装した部分の評価
    - 低遅延・高スループットを確認
    - パケット順序エラー多発により性能劣化

    →これを解決すればより高性能化が期待される

無線センサネットワークを用いた災害モニタリングシステムにおける優先度を考慮した通信手法

大西 潤也 †† 大久保 英嗣 † 横田 裕介 †
† 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

センサネットワークは，複数のセンサノードで構成されるネットワークである．センサネットワークを用いたアプリケーションとして，災害モニタリングシステムが挙げられる．災害モニタリングシステムでは，センサノードの省電力化と重要なデータを優先的に取得したいという要求がある．1 つ目の要求は，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. 2 つ目の要求は，災害を検知したノードにより送信されるパケットは, 災害が発生したことを知るために必要なため, 他のパケットよりも優先的に取得する必要がある．そのため，センサノードの省電力化，および通信の優先度を考慮した通信手法が必要となる. 以上の背景から，本稿では，災害モニタリングシステムにおけるセンサノードの省電力化と通信の優先度を考慮した通信手法の提案を行う.

2 災害モニタリングシステム

災害モニタリングシステムは，継続的に観測を行い, 災害を検知するアプリケーションである．温度の計測や雨量の計測を継続的に行うことで，洪水や火災の検知を行う．災害が発生した場合に，災害が発生している場所が分からず，避難経路を誤り，被害を被ることがある．そのため, 災害が発生した場合に，避難経路の通知を行い，危険な場所を知らせることは重要である．そのため, 本研究では，災害が発生した場合に被災者に災害の情報を早急に提供することとセンサノードの省電力化を目的とした手法を提案する.

3 提案手法

3.1 概要

提案手法では, ノードの位置があらかじめ分かっており, ノード間の時刻の同期が取れていることを前提としている．また各ノードにはノードを識別するための ID が割り当てられているとする．災害を検知したノードにより, 災害の発生を伝えるために送信されるパケットをリアルタイムパケット, 災害を検知していないノードにより, 平常であることを伝えるために送信されるパケットを非リアルタイムパケットとする．またリアルタイムパケットにはデッドラインが設けてあり, デッドラインまでの時間によって通信の優先度が異なる.

観測した値が，予め定めた閾値を超えた場合に，災害を検知したと判断する．災害が発生していない場合は, 平常であることを知らせるために，非リアルタイムパケットのみが送信される．災害発生時は，災害の発生を早急に知らせるために，リアルタイムパケットを非リアルタイムパケットよりも優先して通信する．また, 災害が発生した場合, 時間が経つにつれ, 被害の拡大により, リアルタイムパケットを生成するノードの数が増加する状況が考えられる．リアルタイムパケットを送信するノードが多い場合, 全てのパケットをデッドラインまでに基地局へ届けることが困難になる．このような場合, デッドラインを守ることよりも, 災害を検知した各ノードのパケットを公平に基地局に届けることを重視する．これは，災害を検知した全てのノードからのデータを集めることで，災害の範囲を知ることができ，危険な場所の通知が可能になるためである．そのため，中継ノードは送信元ノードを管理し, 各ノードからのパケットを公平に基地局へ送信することを優先する．また, 災害を新たに検知したノードにより送信されるパケットは, 新たに災害が発生した場所を知り, 危険な場所を知らせることができるため, 最優先で通信を行う．このように, 状況により通信の優先度を変更することで, 災害モニタリングの要求を満たすことを目的とする.

3.2 ルーティング

提案手法では, リアルタイム性とネットワークの稼働時間の長期化を実現するためのルーティングを行う．リアルタイム性の実現のために, 隣接ノードの位置情報, ネットワークの稼動時間の長期化のために隣接ノードの残電力情報を利用する．隣接ノードの残電力情報は, 定期的に残電力情報を載せたパケットを通信することで入手する.

3.2.1 ノードの選択

ノードの選択は，パケットの通信時間, パケットのデッドライン, ノードの通信可能範囲, ノードの位置情報から決定する．提案手法では，MAC プロトコルとしてスロット占有方式を用いるため，パケットの通信時間は, 1 スロット分の時間とする．パケットの通信時間とデッドラインから，何ホップ以内であれば，デッドラインを

守ることができるかを計算する．また，ノードの通信可能範囲と隣接ノードと基地局の位置情報から，基地局までのホップ数を計算し，どのノードであればデッドラインまでに基地局へパケットを届けられるかを調べる．そして，デッドラインまでに届けることが可能となるノードの中から残電力が多いノードを選択する．例としてパケットのデッドラインが短い場合のノードの選択を図 1 に示す．また，パケットのデッドラインに余裕がある場合のノードの選択を図 2 に示す．図 1 では，2 ホップ以内でないとデッドラインまでに基地局へパケットを届けられないとする．また，ノードの残電力は，node2 よりも node1 のほうが多いとする．この場合，ノードの通信可能範囲と位置情報から，node1 と node2 が基地局まで 1hop でパケットを届けることが可能であると分かる．そのため，残電力が多い node1 を送信先ノードとして選択する．図 2 では，デッドラインに余裕があり，5hop 程度余裕があるとする．この場合は残電力を重視したノードの選択を行う．図 1 において node1 よりも node2 のほうが残電力が多いとする．この場合, 残電力が多い node2 を選択する．

3.2.2 中継するパケットの選択

災害の範囲が狭い場合は, デッドラインまでにパケットを送信するために, デッドラインまでの時間が短いパケットから送信する．災害の範囲が広い場合は, 送信したノード ID と送信した時間をデータベースで管理し, 長い間送信されていないノードからのリアルタイムパケットから送信する．また，災害の範囲に関わらず, 新たに災害を検知したノードにより送信されるパケットは，新たな災害場所を知る上で重要である．そのため，データベースに登録されていないノードからのパケットは，最優先で通信を行う．

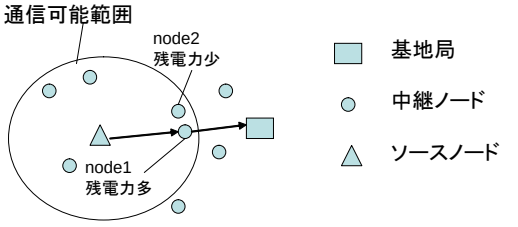


図 1　デッドラインが短い場合のノードの選択

3.3 MAC プロトコル

提案手法では，時間がフレームを単位として分割されているとする. パケットの送信には，スロット占有方式を用いる. 提案手法のフレーム構成を図 3 に示す．これ

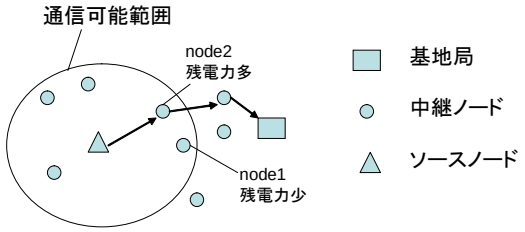


図 2　デッドラインまでに余裕がある場合

は，1 つのノードが 1 つのスロットを占有し，通信を行う手法である. パケットの送信手順を図 4 に示す. 送信を行いたいノードは，バックオフ時間 backoff だけ待つ. backoff の長さは，通信の優先度によって異なり，優先度が高いほど短い. これにより，優先度に応じた通信が可能になる. 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，送信先ノード以外のノードは，次のスロットまでスリープする. これにより，送信権を得られなかったノード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，ノードの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，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.

3.4 通信の優先度

優先度が高い程, バックオフ時間が短くなる．災害の範囲が狭い場合は，パケットのデッドラインまでの時間が短いノード程高い優先度が割り当てられる．また，災害の範囲が広い場合は，前回送信してからの経過時間が長いノードからのパケットを持つノード程優先度が高くなる．新たに災害を検知したノードは, 優先度が一番高くなる．

3.5 通信の優先度の切り替え

災害の範囲が狭いか広いかによって通信の優先度が異なる．災害の範囲が狭いか広いかの判断は基地局が行う．複数のノードからリアルタイムパケットを受信した時に, ノードの位置情報から災害の範囲が狭いか広いかを判断する．優先度の切り替えを行う場合は, 優先度の切り替えを要求するパケットを基地局からフラディング

により送信する．

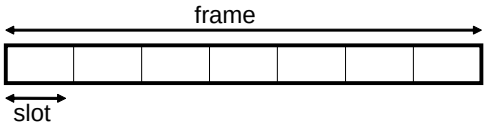


図 3　提案手法のフレーム構成

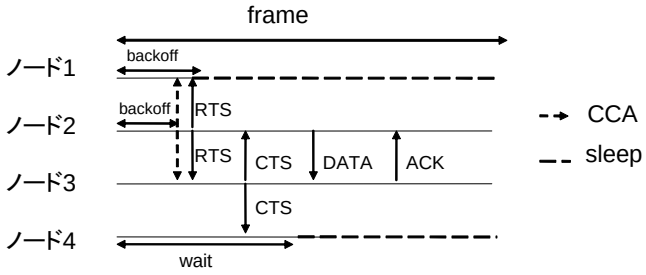


図 4　パケットの送信手順

4　評価方針

提案手法の評価では，ネットワークシミュレータ ns-2[1] を用いて既存の手法 [2] との比較を行う．ネットワークの稼働時間, デッドライン内のリアルタイムパケットの到達率, リアルタイムパケットが増加した場合の公平性，パケットの送信間隔を変化させた場合の公平性を評価する．

5　おわりに

本稿では無線センサネットワークを用いた災害モニタリングシステムにおける優先度を考慮した通信手法の提案を行った．今後は,ns-2 を用いて, 提案手法を実装し, 評価を行う予定である．

参考文献

[1] The Network Simulator ns-2, “http://www.isi.edu/nsnam/ns/”

[2] Abinash Mahapatra, Kumar Anand,Dharma P. Agrawal, ”QoS and energy aware routing for real-time traffic in wireless sensor networks,” Computer Communications 29 (2006) 437-445

無線センサネットワークを用いた災害モニタリングシステムにおける優先度を考慮した通信手法

立命館大学大学院　理工学研究科 M2
大久保･横田研究室
大西　潤也

発表内容

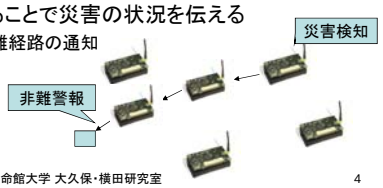
- はじめに
- 災害モニタリングシステム
- 提案手法
  - 優先度の変更
  - ルーティング
    - 送信先ノードの選択
    - 送信するパケットの選択
  - MACプロトコル
    - バックオフによる優先制御
- 評価方針
- おわりに

はじめに

- 無線センサネットワーク
  - 複数の無線機能を持つセンサで構成されるネットワーク
  - 温度や雨量などの観測が可能
- 無線センサネットワークのアプリケーション
  - 災害モニタリングシステム
    - 洪水や火災などの監視を行う
- 災害モニタリングシステムの目的
  - 災害が発生していることを早急に伝える
  - 災害が発生している範囲を知らせる
    - 被害拡大の対策
    - 避難経路，危険な場所の通知

災害モニタリングシステム(1/2)

- 平常時
  - 災害が発生していないことを伝えるために観測
    - 安全なことを確認
- 災害発生時
  - 災害の発生を伝える
  - 継続して観測することで災害の状況を伝える
    - 災害の範囲　避難経路の通知



災害モニタリングシステム(2/2)

- 災害モニタリングシステムで発生するパケット
  - リアルタイムパケット
    - 災害発生していることを知らせるパケット
      - 災害を検知したノードが送信
  - 非リアルタイムパケット
    - 災害が発生していないことを知らせるパケット
      - 災害を検知していないノードが送信
- 災害の検知
  - 温度や雨量の観測を行い予め定めた閾値を超えた場合災害が発生していると判断する

災害モニタリングシステムの要求

- センサノードの省電力化
  - 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
- 重要なデータを優先的に取得
  - 災害を検知したノードのリアルタイムパケット
    - 災害の範囲が狭い場合
      - デッドラインに間に合うように通信
        - » 早急に災害の発生を知るため
    - 災害の範囲が広い場合
      - 全てのパケットをデッドラインまでに届けることは困難
      - 多くのソースノードからのリアルタイムパケットを公平に通信
        - » 被害の拡大状況の把握
        - » 避難経路の通知

提案手法

- 災害の状況によりパケットの優先度を変更する
  - 災害の範囲が狭い場合
    - 各ノードのパケットをデッドラインまでに送信する
  - 災害の範囲が広い場合
    - 各ノードのパケットを公平に送信する
- ルーティング
  - 送信先ノードの選択
  - 送信するパケットの選択
- MACプロトコル
  - バックオフによる優先制御
- 前提条件
  - ノードの位置情報はGPSにより予め分かっているとする
  - 各ノードを識別するIDが割り当てられているとする

優先度の切り替え

- 災害の範囲が狭いが広いかの判断
- 複数のノードからリアルタイムパケットを基地局が受信した時
  - ノードの位置情報から災害の範囲が狭いか広いか判断
- 優先度の切り替えを要求するパケットをフラッディングにより全ノードへ送信する

ルーティング

- 災害の範囲が狭い場合
  - 送信先ノードの選択
    - パケットのデッドラインと残電力から送信先ノードを選択
  - 送信するパケットの選択
    - デッドラインが短いパケットから優先的に送信
- 災害の範囲が広い場合
  - 送信先ノードの選択
    - 残電力の多いノードを選択
  - 送信するパケットの選択
    - 長い間送信されていないノードからのパケットを優先的に送信

送信先ノードの選択(1/2)

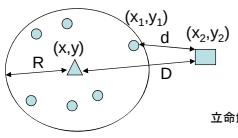
- パケットの通信にかかる時間
  - MACプロトコルとしてスロット占有方式を用いる
    - 1スロット分の時間
  - パケットのデッドラインまでの時間と通信時間から何ホップならデッドラインに間に合うか計算
  - ホップ数=デッドラインまでの時間/通信時間
- ノードの通信可能範囲とノードの位置情報から何ホップで通信可能か計算
- 条件に合うノードの中から残電力の多いノードを選択

送信先ノードの選択(2/2)

- 通信可能範囲Rとノード間の距離からホップ数を計算
- ソースノードから基地局までのホップ数=D/R
- 中継ノードから基地局までのホップ数=d/R

$$d = \sqrt{(x_2^{\,2} - x_1^{\,2}) - (y_2^{\,2} - y_1^{\,2})}$$

$$D = \sqrt{(x_2^{\,2} - x^2) - (y_2^{\,2} - y^2)}$$



基地局

中継ノード

ソースノード

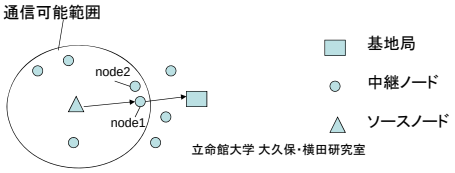
R 通信可能範囲
D ソースノードから基地局までの距離
d 中継ノードから基地局までの距離

災害の範囲が狭い場合

- パケットのデッドラインが短い場合
  - 基地局までの距離を重視したノード選択
- パケットのデッドラインまでに余裕がある場合
  - 隣接ノードの残電力を重視したノードの選択
  - 定期的に隣接ノードとノードの残電力情報を入手

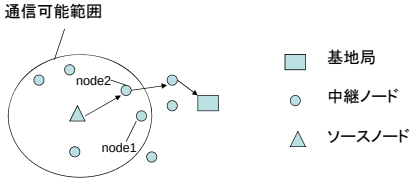
デッドラインが短い場合

- 例 2hop以内でないと間に合わない場合
  - 隣接ノードの中から2hop以内で通信可能なノードの検索
  - ノードの位置情報によりnode1とnode2はデッドラインまでに通信可能と判断する
  - node1のほうが残電力が多いとする
  - node1を経由して基地局へ送信



デッドラインまでに余裕がある場合

- 例 5hop程度余裕がある場合
  - ノードの位置情報からノードの選択
  - デッドラインに余裕があるため残電力を重視する
  - 一番残電力の多いnode2を選択する



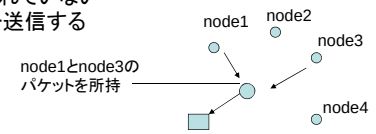
パケットの選択

- 災害の範囲が狭い場合
  - デッドラインが短いパケットを優先的に送信
    - 災害の発生を早急に伝えるため
  - 送信元ノードをデータベースで管理
    - ノードID
- 災害の範囲が広い場合
  - 送信元ノードをデータベースで管理
    - ノードID,前回送信した時間
  - 各ノードのリアルタイムパケットを公平に送信
  - 長い時間送信していないノードからのパケットを優先的に送信
    - 災害の範囲,危険な場所を把握するため

災害の範囲が広い場合

- 例 4つのノードのパケットがある場合
- 送信履歴から次に送信するパケットを選択する
- 現在node1とnode3からのパケットを所持しているとする
- 一番長い間送信されていないnode1のパケットを送信する

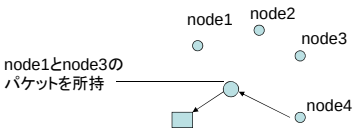
ノードID	送信時間
node1	0時1分
node2	0時2分
node3	0時3分
node4	0時4分



新たに災害を検知したノード

- 例 node4が新たに災害を検知
- node4のパケットを先に送信
  - 新たに災害が発生した場所を知ることで危険な場所を知らせることができるため

ノードID	送信時間
node1	0時1分
node2	0時2分
node3	0時3分
node4	

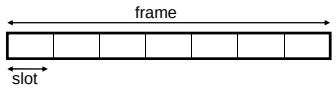


MACプロトコル

- 消費電力の削減と優先度に応じた通信を行う
- スロット占有方式を用いる
- スロット占有方式
  - 送信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
- 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

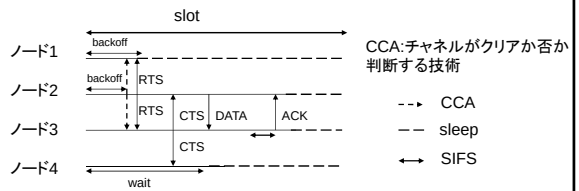
スロット占有方式

- 時間をフレームを単位に分割する
- 1つのフレームは複数のスロットに分割される
- 1つのノードが1つのスロットを占有し通信を行う



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
- 高優先度のノード2がノード3にデータを送信



バックオフによる優先制御

- 優先度が高いほど短いバックオフ時間で通信
- 災害の範囲が狭い場合
  - デッドラインが短いパケット程優先度が高くなる
    - デッドラインまでにパケットを届けるため
- 災害の範囲が広い場合
  - 前回送信してからの経過時間が長いノードからのパケット程優先度が高くなる
    - 公平に通信を行うため
- 新たに災害を検知したノード
  - 優先度が一番高い
    - 新たに発生した危険な場所を知らせることができるため

優先度のまとめ(1/2)

- 災害の範囲が狭い場合
  - パケットの選択
    - デッドラインが短いパケット
  - バックオフ時間
    - デッドラインが短いほどバックオフ時間が短い
- 災害の範囲が広い場合
  - パケットの選択
    - 長い時間送信していないソースノードからのパケットを優先的に送信
  - バックオフ時間
    - 前回送信されてからの時間が長いほどバックオフ時間が短い

優先度のまとめ(2/2)

- 新たに災害を検知したノード
  - パケットの選択
    - 災害の範囲に関わらず最も優先される
  - バックオフ時間
    - 最も短いバックオフ時間

評価方針

- Network Simulator2(ns-2)上に実装
- シミュレーションによる評価
- 既存の手法との比較を行う
- 評価項目
  - ネットワークの稼働時間
  - デッドライン内のパケット到達率
  - リアルタイムパケットが増加した場合の公平性
  - パケットの送信間隔を変化させた場合の公平性

おわりに

- 災害モニタリングシステム
  - 災害モニタリングシステムに求められる要件
- 提案手法
  - 残電力とデッドラインを考慮したルーティング
    - ノードの選択
    - パケットの選択
  - スリープによる電力消費を考慮したMACプロトコル
    - バックオフによる優先制御
  - 優先度の切り替え
- 評価方針
- 今後の予定
  - 提案手法の実装　評価

RT-VMM 構築のための Xen と RTOS の割込み処理時間の可視化

金川 高久 †　　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, 計算機の性能向上に伴い仮想計算機技術が注目されている. 仮想計算機は, その上で動作する OS に仮想化した資源を提供する. これらの資源は, 仮想計算機モニタ (以降, VMM) と呼ばれるソフトウェアによって, 物理資源を仮想化し計算機環境をエミュレートすることで実現している. VMM を用い, ハードウェア資源を効率良く利用することで, OS に複数の仮想計算機環境を提供し同時に動作させることができる [1].

組込みシステムの分野では, リアルタイム OS(以下, RTOS) が利用されている [2]. 組込みシステムでは, システムの信頼性と機能性の両立のために, RTOS と高機能 OS を組み合わせて用いることがあり, そのために必要な資源は, システムの規模によって増加する. 組込みシステムでは, システムの信頼性と機能性を維持しながら資源のサイズやコストを抑えたい要求がある. その要求に対し, 仮想化技術を適用することで, RTOS と高機能 OS をその上で同時に動作させるリアルタイム VMM(RT-VMM) により, 機能性の向上や適切な資源配分により多様な要求を満たすことが可能になる. そのため, 我々は, ハイパーバイザ型の VMM である Xen を用い RT-VMM の開発行っている. Xen を基に開発することで, Xen に対応したソフトウェアが利用できる.

RTOS は, 約束された割込み処理時間を基にタスクの処理完了時間を予測し, デッドラインまでに RTOS が処理を完了するようにスケジューリングを行うことで, 処理のリアルタイム性を保つ. そのため, VMM 上の RTOS がリアルタイムシステムの処理を行うには, RTOS がタスクの処理完了時間を予測するために, RT-VMM は RTOS に対し一定の処理時間を約束し, RTOS の処理を考慮した資源の割当てを行う必要がある. しかし, Xen は, RTOS が動作することを目的としていないため, 仮想化によるオーバヘッドや処理時間のばらつきがありリアルタイム性を保証していない. RTOS が最悪実行時間を守るスケジューリングを行うには, VMM の処理時間を一定に保ちリアルタイム性を保証する必要がある.

VMM が一定の応答性能を保つには, VMM の処理時間として割込み処理時間の遅延と揺らぎを抑える必要がある. そのため, VMM 内の割込み処理と RTOS 内の処理を可視化し, 割込み処理時間の揺らぎと, これによるタスク処理時間の遅延や揺らぎを理解する必要がある. その可視化を行うツールを Xen に追加し, リアルタイム VMM の構築につなげる.

本稿では, Xen と RTOS の割込み処理時間と RTOS のタスク処理時間の可視化について述べる. 2 章で, Xen とゲスト OS の割込み処理について述べ, 3 章で, Xen と RTOS の割込み処理時間とタスク処理時間の可視化について述べる. 4 章では, 取得した割込み情報の可視化について述べる. おわりにで本稿をまとめる.

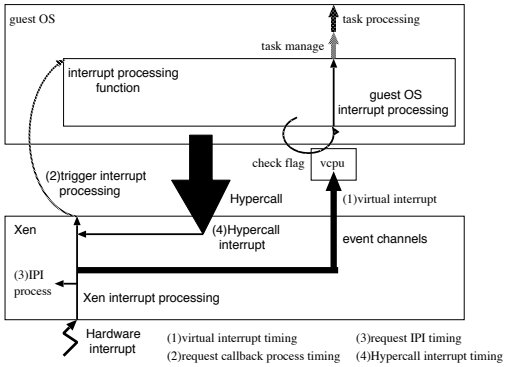


図 1　イベントチャネルによる仮想タイマ割込み通知

2　Xen とゲスト OS の割込み処理

Xen は, ハードウェア割込みを仮想割込みに変換して, VMM 上のゲスト OS へ通知する. Xen は, 図 1 に示すように, イベントチャネルを用いてゲスト OS へ仮想割込みを行うことを予め通知するために, 割り込み先のゲスト OS に割り当てられている仮想化された CPU(vcpu) に仮想割込みが通知されたことを示す値と, 通知されたポートから仮想割込みの種類を示す値をイベントチャネルを用いて設定する (図 1 の (1)). 通知された仮想割込みは, Xen の割込み処理後に, 通知を受けたゲスト OS へ復帰することで処理する (図 1 の (2)). ハードウェア割込みを受けた CPU がゲスト OS を実行していない場合, CPU は, ゲスト OS を実行している CPU へ CPU 間割込みを行う (図 1 の (3)). CPU 間割込みを受けた CPU は, Xen 内での割込み処理ハンドラ実行後に, ゲスト OS に処理を復帰し仮想割込みを処理する. また, Xen がゲスト OS からハイパーコール割込みを受けた場合 (図 1 の (4)), Xen 内でのハイパーコール割込み処理完了後に, ゲスト OS に処理を復帰する.

ゲスト OS は, 仮想割込みが通知されているかチェックし, 通知されたすべての仮想割込みを処理する [3]. 仮想割込みの処理後, この処理の間に通知された仮想割込みをチェックする. 通知された仮想割込み処理が完了した場合, ゲスト OS へ復帰し処理を再開する.

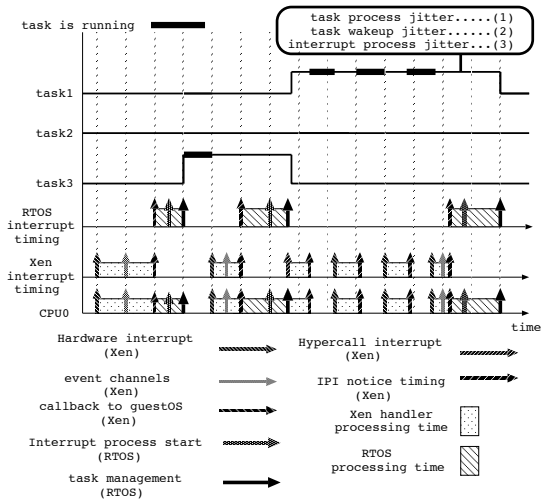


図 2　割込み処理時間とタスク処理時間への影響を可視化

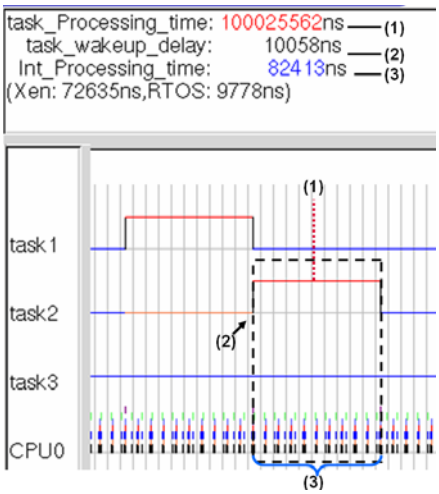


図 3　割込み処理時間とタスク処理を可視化

3　Xen と RTOS の割込み処理時間とタスク処理時間の可視化

Xen と RTOS の割込み処理時間やタスク処理時間の遅延や揺らぎを表示するために, Xen と RTOS の割込み処理時間とタスク動作を可視化する. 表示した処理時間とその揺らぎによる RTOS の応答時間の遅延が表示できる. この可視化を行うには, 割込み処理情報として, 以下の値が必要となる.

- Xen がハードウェア割込みを受ける時刻.
- Xen が RTOS へ仮想割込みを通知する時刻.
- Xen の割込み処理から RTOS へ処理を復帰する時刻.
- RTOS が仮想割込み処理を開始する時刻.
- RTOS がタスク管理処理を行う時刻.
- Xen がハイパーコール割込みを受ける時刻.

取得情報から, Xen がハードウェア割込みを受け, RTOS へ処理を復帰するまでの時間を Xen 内の割込み処理時間とし, RTOS の仮想割込み処理開始から各タスクの管理処理完了までを RTOS の割込み処理時間とする. 求めた処理時間の Xen と RTOS の割込み処理とタスク処理可視化として, 図 2 に示す. 可視化内容から, タスク 1 処理中の割込み処理によるタスク処理時間の遅延と揺らぎや、その基となる割込み処理の処理時間や発生時刻 (図 2 の (1)) が分かる. また, タスクの起床時刻の揺らぎ (図 2 の (2)) や, タスク処理中の割込みの処理時間の揺らぎ (図 2 の (3)) が分かる.

割込み処理時間の揺らぎによるタスク処理時間の揺らぎを理解し, 割込み処理時間の揺らぎによるタスク処理時間の揺らぎを抑えることで, RTOS の応答時間を一定に保ちリアルタイム性の保証につなげることができる.

4　取得情報による可視化

Xen と RTOS の割込み処理情報とタスク動作情報を可視化情報として保存する機能を実装し, RTOS を CPU0 上で動作させ情報取得を行った. 可視化情報をもとに, Xen と RTOS の割込み処理時間とタスク処理時間の可視化図を図 3 に示す. 図 3 は, タスクの処理時間 (図 3 の (1)) とタイマ割込みからタスクが起床時刻 (図 3 の (2)) までの割込み処理時間, Xen の割込み処理時間 (図 3 の (3)) を表示している. 表示した割込み処理時間を基にその揺らぎを求めることで, 割込み処理時間の揺らぎによるタスク処理時間の揺らぎを表示することができる.

5　おわりに

本稿では, 割込み処理時間の揺らぎと, これによるタスク処理時間の遅延や揺らぎを理解するための, 割込み処理時間とタスク処理時間の可視化について述べた. 実装した可視化機構を用いて, 現在, タスク処理時間中の割込み処理時間や割込み処理のタイミングなど割込み処理内容の表示まで完了していることを述べた.

今後の課題として, 割込み処理時間とタスク処理時間の揺らぎを表示し, 揺らぎがリアルタイム性を保証できる範囲であるかの評価を行う.

参考文献

[1] 大島孝子, 平初, 長谷川猛, 宮本久仁男: “Xen 徹底入門,” 株式会社翔永社, pp. 16-19 (2007).

[2] 永井正武, 澤井勉, 権藤正樹: “実用組込み OS 構築技法,” 共立出版株式会社出版, pp. 7-11 (2002).

[3] David Chisnall, 渡邉 了介: “仮想技術 Xen-概念と内部構造,” 毎日コミュニケーションズ出版, p 191 (2008).

RT-VMM構築のための XenとRTOSの割込み処理時間の可視化

立命館大学大学院毛利研究室
M2
金川高久

はじめに(1/2)

- 組込みシステムで仮想計算機モニタ(VMM)を利用しリアルタイムOS(RTOS)と汎用OSを動作させたい
  - 機能性の向上や機材などのコスト削減
- Xenを用いてRTOSを動作させるVMMを開発
- RTOSの処理
  - 時間内での割込み処理を約束
  - 最悪実行時間を守るタスクスケジュール
- XenはRTOSの動作を目的としていない
  - 仮想化による割込みのオーバヘッドが大きい
  - 一定でないVMMの処理時間により、タスク処理時間が揺らぐ

RTOSは最悪実行時間を守れないため
リアルタイム性が保証できない

はじめに(2/2)

- VMMの処理時間を約束しタスク処理時間の遅延の揺らぎを抑える
- タスク処理時間の揺らぎである VMMの割込み処理時間の揺らぎを抑える

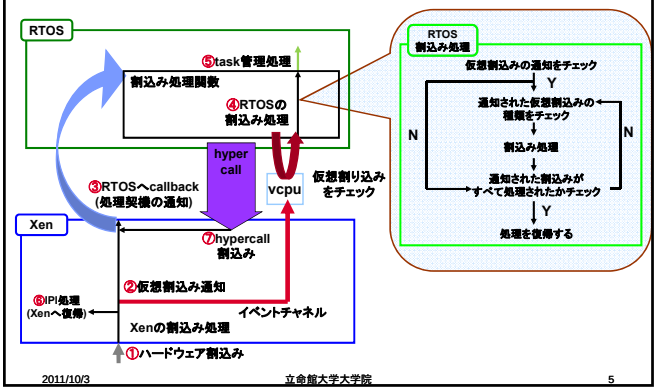
VMMの割込み処理時間の揺らぎと
処理の内容と知る必要がある

可視化の目的

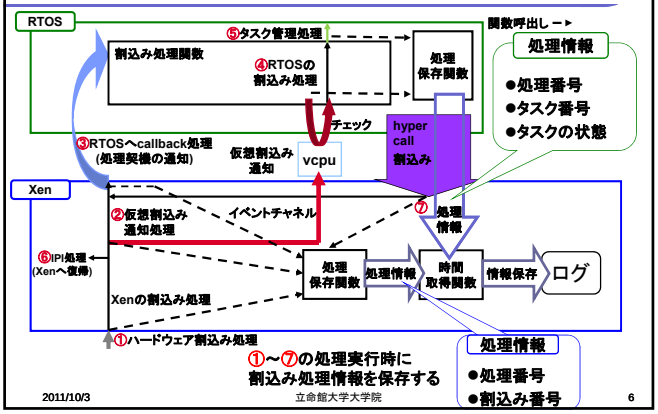
- VMMとRTOSの処理可視化ツール
  - 割込み処理による タスク処理時間の遅延とその揺らぎを表示
  - 揺らぎの要因となる割込み処理の情報を表示

割込み処理時間とタスク処理時間の揺らぎの大きさと影響を可視化し
リアルタイムVMMの構築に役立てる

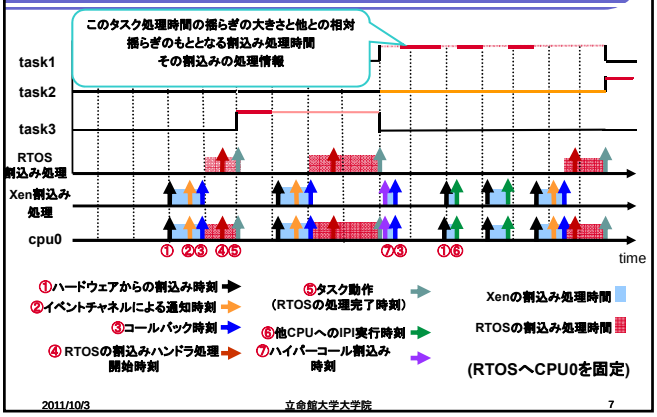
XenとRTOSの割込み処理



割込み処理情報保存機能



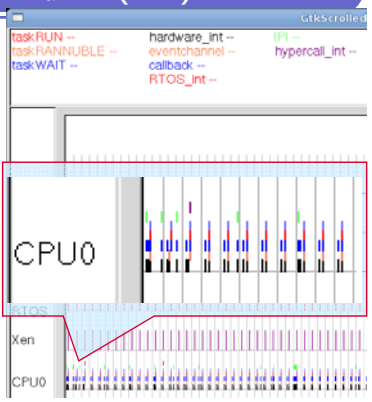
割込み処理時間の可視化



取得情報による可視化(1/3)

表示内容

- タイマ割込み処理
  - ハードウェア割込み
  - イベントチャネル
  - Callback
  - RTOS割込み処理
- IPI処理
  - ハードウェア割込み
  - 他CPUへのIPI
- ハイパーコール割込み処理
  - ハイパーコール
  - Callback

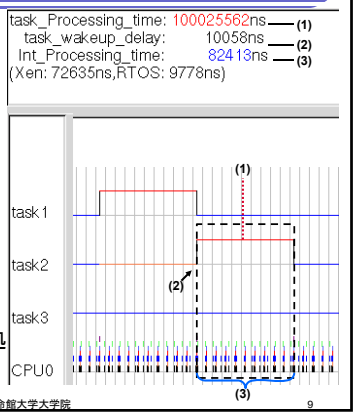


- 表示内容から分かること
  - 発生した割込みの種類

取得情報による可視化(2/3)

表示内容

- タスク処理時間
- タスク実行中の割込み処理時間
- タスク起床までの割り込み処理時間
- タスクの状態
  - 待ち状態
  - 実行状態
  - 実行可能状態

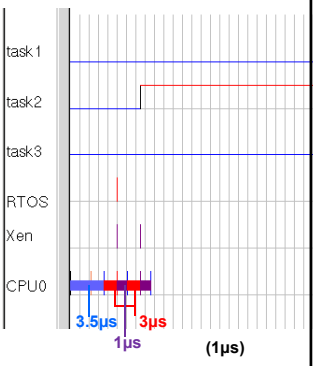


- 表示内容から分かること
  - 割込みが発生し
    タスク処理時間が遅延
  - タスク起床までの割込み処理による遅延

取得情報による可視化(3/3)

表示内容

- タイマ割込み処理
- ハイパーコール処理
- タスクの起床



- 表示内容から分かること
  - 割込み処理内容
  - 割込み処理とタスク処理タイミング

おわりに

- RTOSのリアルタイム性を保証するために
  VMMの割込み処理の揺らぎを抑える
- 割込み処理時間の可視化
  - Xenの割込み処理時間とタスク処理時間の揺らぎを表示
- 現在のXenとRTOSの割込み処理可視化
  - タスク処理中の割込み処理時間
  - 割込み処理とタスク処理タイミング
- 今後の課題
  - 割込み処理時間とタスク処理時間の揺らぎを表示する
  - 揺らぎがリアルタイム性を実現できる範囲であるかの表示

Androidにおけるプロセス可視化環境の開発

中川　裕貴[†]　Praween Amontamavut[††]　西野　洋介[†††]　早川　栄一[††]
拓殖大学　大学院　電子情報工学専攻[†]　拓殖大学　工学部　情報工学科[††]
東京都立　八王子桑志高等学校[†††]

1. 研究の背景

Android を携帯電話や組込み OS として利用する機会が増加している。その背景としては、Android の端末の開発時にかかるライセンス料金がなく、コストの削減ができる。他にも、Eclipse による統合開発環境が提供されており、一般の開発者が無償で Android のアプリケーションを作成、デバックが可能であることが挙げられる。そして、Android 利用者の増加に伴い、Android のアプリケーションやオペレーティングシステムに関する学習をする者も増加している。

しかし、Android の学習には問題点がある。OS 学習のためには、基礎としての OS 各機能の仕組みを理解しなければいけない。その基礎としてプロセス管理を学習することが理解を深めるためには有効である。また、Android には DalvikVM という仮想マシンがある。Android は Linux カーネルの C 言語と DalvikVM の Java 言語の二つの言語で動いている。この C 言語と Java 言語の間でプロセスがどのように生成されているのか理解が難しい。

2. 目的

本研究の目的は、可視化対象を Android とし、その動作を可視化するプログラムの開発を行う。その中でも、プロセス管理の可視化を行う。これにより OS の基礎であるプロセス管理を理解できる。また、プロセスの動作を把握することにより、複数あるプロセスの動作も理解できるようになる。

3. 設計

3.1　全体構成

全体構成を図 1 に示す。Android 内にある ftrace と logcat でログ情報をサーバへ転送し、ログファイルを生成する。データサイズが大きくなるので、ログの保存を Android 内ではなく、サーバ内に保存することにした。サーバ内で可視化するためにログファイルを加工する。

加工したログファイルを jQuery で可視化をする。jQuery とは、JavaScript と HTML の両方で使用できる JavaScript ライブラリである。フリーソフトでかつオープンソースなので拡張性がある。なので、既存の jQuery プラグインを使用すること以外にも自身で新しいプラグインを作成することができる。よって、三種類の可視化を行うので、それら三つのプラグインを作成した。状態遷移の可視化を行う際に使用する OS モデル図可視化プラグイン、時間変化グラフの可視化を行う際に使用する時間変化グラフ可視化プラグイン、ツリー構造の可視化を行う際に使用するツリー構造可視化プラグインの三種類を作成した。

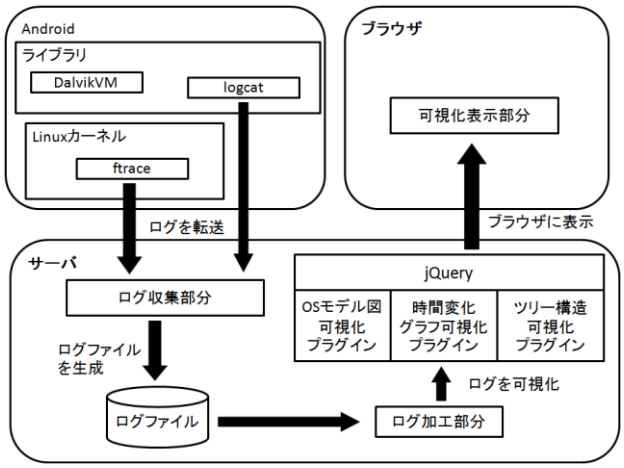


図 1　全体構成

可視化の実行画面をアニメーションで表示する。アニメーションで表示させることにより、学習者にとって可視化結果が見えやすくなる。それだけではなく、ツリー構造の場合は、プロセスの生成、消滅といった様子や時間変化グラフは時間経過の様子、状態遷移図の場合、プロセスの状態の遷移を動的に表現することにより学習者が視覚的に理解できる。

3.2　可視化手法

Android の可視化部分としてはプロセス管理と

Development of Visualization Environment for Process on Android
†Postgraduate cource, Takushoku University
††Faculty of Engineering, Takushoku University
†††Hachioji Soshi High School

して次の三つで可視化をする。

（1）OS モデル図

プロセスの状態を三種類に分けて表示していき、実行状態、実行可能状態、待ち状態となる。待ち状態にも様々な状態が存在する。入出力とは無関係に遷移する待ち状態やディスクの読み書きの待ち状態、停止状態のために待ち状態、トレース中による待ち状態、ゾンビ状態による待ち状態、存在しないため待ち状態といった六種類の待ち状態があり、それらを可視化していく。

（2）**時間変化グラフ**

各プロセスの状態遷移を棒グラフによる表示をする。プロセスの実行状態、実行可能状態、待ち状態は三つの色に分けてグラフ上に表示していく。また、待ち状態の場合、どのような理由で待ち状態になっているのかを表示する。

（3）**ツリー構造**

プロセス同士の繋がりを目で見えるようするために可視化する。プロセスの親子関係やプロセスの生成、消滅をツリー構造で表示していく。

3.3 プロセスの情報取得

Android の可視化を行うためにプロセスの状態遷移や次に実行されるプロセスといった情報を取得する必要がある。その情報を取得するため ftrace と logcat を使用する。[1][2]ftrace とは、Linux カーネル内にあるトレーサである。logcat とは、Android のデバック用ツールである。プログラムを実行させ、そのプログラムが実行している間のプロセスをトレースして状態遷移などの情報も取得する。そして、トレース中にオーバーヘッドが起きない。なので、プログラムを実行しながら、トレースをしてもシステムの負担はない。

Android 内部でログの解析、収集を行わずに外部のサーバ上で行う。それにより、Android での実行でのコストがかからなくなり、OS の動作に影響を与えない。

4. 実装

4.1 実装環境

CPU ボードは Armadillo-440 を使用し、Android は ver2.2 の Froyo を使用した。

4.2 実装結果

実装結果を図 2 に示す。左上が OS モデル図を可視化したものである。四角形のボックスにはプロセス名があり、その上には状態を表示している。プロセスの状態が変わる度にボックスが変わった状態へと移動し、その状態の色に変化する。

左下が時間変化グラフを可視化したものである。縦軸にプロセス名、横軸に時間を表示している。状態を色で表しており、その色は状態遷移図の色と同じである。時間の経過に伴い、グラフを左へ動かしていく。また、待ち状態の場合、棒グラフの上にどのような理由で待ち状態なのかを表示する。

右下がツリー構造を可視化したものを表示している。四角にはプロセス名があり、そのプロセス同士を繋いでいる。上下の繋がりで親子関係を表している。プロセスの生成、消滅をアニメーションで表示している。

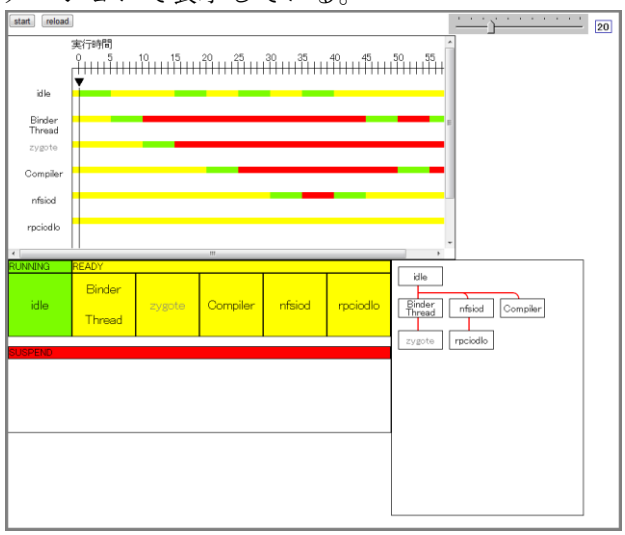


図 2　実装結果

Android の CPU ボードから動作ログをサーバへ送り、ログファイルをサーバ内で生成、可視化をしてブラウザで表示した。

5. おわりに

Android のプロセスのログデータを取得し、可視化の表示することによりプロセスの動作を視覚的に理解できるようになった。

今後の課題としては、Java 言語の可視化を表示すること。

参考文献

[1]安藤 友樹、柴田 誠也、本田 晋也、冨山 宏之、高田 広章：組込みマルチプロセッサシステムの設計改善支援、SWEST12（2010）pp.23-26

[2]本橋 大樹、西野 洋介、早川 栄一：組込みシステム学習支援環境「港」における Linux プロセス可視化環境の開発（コンピュータシステム）、電子情報通信学会（2010）pp.279-285

Androidにおけるプロセス可視化環境の開発

中川裕貴†、Praween Amontamavut††、
西野洋介†††、早川栄一††
†拓殖大学　大学院　電子情報工学専攻
††拓殖大学　工学部　情報工学科
†††東京都立　八王子桑志高等学校

背景

- Androidを組込みOSに利用する機会が増加
- 利用の増加に伴い、学習者も増加
  - ソースコードを見ることができる
  - Eclipseによる統合開発環境の提供により、Androidの開発、デバックが可能

問題点

- C言語とJava言語の間でのプロセスの対応
  - 2つ言語の間でどのような対応しているか
- 学習用のアプリケーションのインストールの手間
  - 複数のマシンにインストールする時間
  - マシンやOSによってアプリケーションエラーが起きる

目的

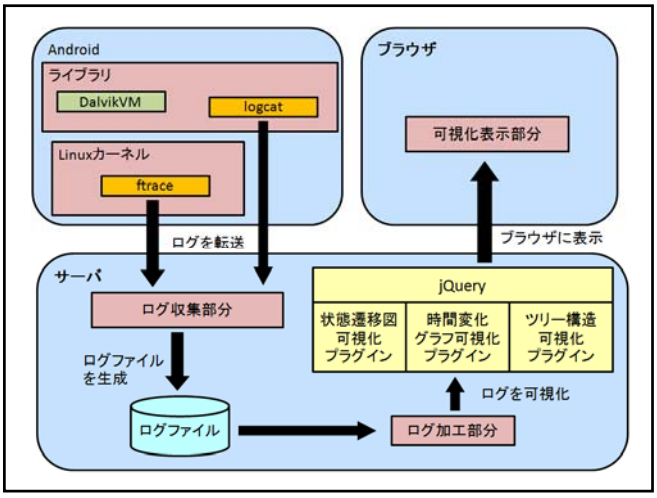
- Androidのプロセスを可視化をし、表示
  - プロセス管理の可視化
    - OSの基礎を理解できる
    - プロセスの動作の把握が可能
    - 複数あるプロセスの動作も理解できる
  - 学習者の対象として学校の専門教育や企業の学習者

特徴

- 可視化の実行画面をブラウザ上で表示
  - アプリケーションのインストール不要
  - ネットが使える環境であれば使用できるので利用しやすい
  - アニメーション表示で学習者に理解しやすい
- ログファイルをサーバに保存
  - ログファイルをサーバに送ることによりAndroid内の保存領域を使わずにすむ
  - サイズの大きいログファイルも保存可能

関連研究

- TraceLogVisualizer
  - マルチプロセッサのトレーサログを可視化し表示
  - アプリケーションで表示している点が異なる
- Vzet
  - LinuxシステムのCPUのプロセスを可視化
  - グラフのみの表示している点が異なる
- 港システム
  - 早川研究室で行っている組込みシステム学習支援環境
  - 使用するのにインストールの必要がない点が異なる



可視化の表示

- 可視化の表示にはJavaScriptを使用
  - Flashのようなインストールが不要
  - 動作確認やデバックが容易
- 可視化の表示にはアニメーションを使用
  - 静的な表示より動的な表示の方が理解しやすい
    - プロセスの状態遷移を動的に表示
  - アニメーションの表示速度は調節が可能
- アニメーション表示にはjQueryを使用
  - さまざまなプラグインで描画
  - プラグインの自作が可能
  - 可視化専用のプラグインを作成

プラグイン

- 状態遷移図可視化プラグイン
  - プロセスの状態の遷移を知るために表示
- 時間変化グラフ可視化プラグイン
  - プロセスの時間経過による状態の変化を理解するために表示
- ツリー構造可視化プラグイン
  - プロセス同士の関係性を知るために表示

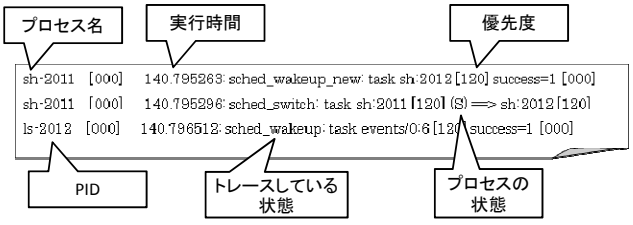
可視化方法

- プロセスの可視化にftraceとlogcatを使用
  - ftraceはトレーサ
  - logcatはデバック用ツール
  - Androidの動作ログを生成
- Android内で生成したログをサーバに送る
  - Android内の保存領域に関係なく保存可能
  - サーバ内にあるログをブラウザ上で可視化
  - 動作ログとブラウザの通信はJSONを使用

プロセス可視化情報

- ftraceからログを取得
  - ftraceはLinuxカーネル内にあるトレーサ
  - プロセスの情報取得に利用
  - 以下の情報を利用して可視化
    - プロセス名
    - プロセスID
    - 実行時間
    - プロセスの状態
    - トレースしている状態

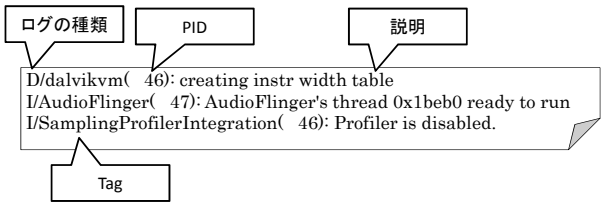
ftraceのログ



プロセス可視化情報

- logcatからログを取得
  - Androidのlogcatを利用
  - Androidのデバックするためのツール
  - ftraceのログ情報と合わせて可視化
    - ftraceはC言語にプロセス可視化
    - logcatはJava言語のプロセス可視化
  - プロセスIDを利用して可視化
    - DalvikVMで動作しているPIDとftraceのログのPIDが同じものの可視化表示を変更

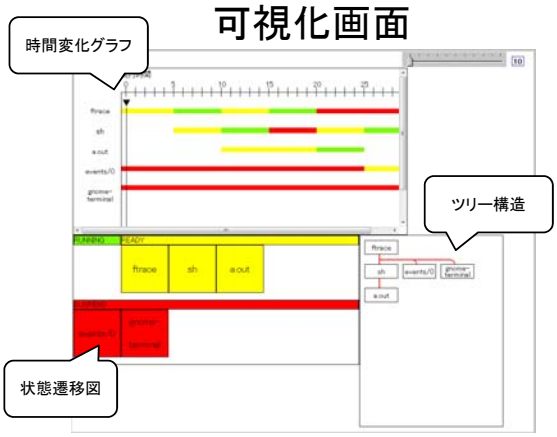
logcatのログ



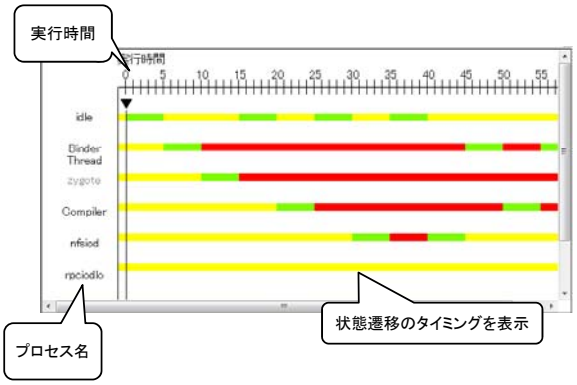
可視化内容

- Androidのプロセス可視化内容
  - 状態遷移図
    - プロセスの状態遷移を図式化
  - 時間変化グラフ
    - プロセスの状態遷移の時間変化をグラフ化
  - ツリー構造
    - プロセス同士の関係性をツリー構造で表示

可視化画面



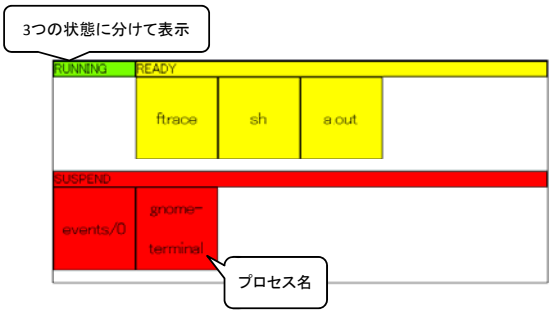
時間変化グラフ



時間変化グラフ

- プロセスの状態を色で識別する
  - 実行状態→緑色
  - 実行可能状態→黄色
  - 待ち状態→赤色
- C言語とJava言語の違いをプロセス名の色の違いで表現
  - ftraceのログのPIDとDalvikVMのログのPIDが同じプロセスの名前の色を黒から灰色に変化

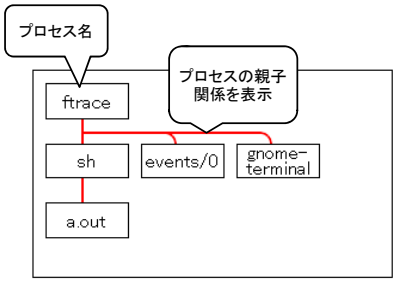
状態遷移図



状態遷移図

- どのタイミングでプロセスか遷移するか理解できる
- 状態遷移をアニメーションで表示
  - 状態の遷移を動的に理解できる

ツリー構造



ツリー構造

- プロセス同士の関係性や親子関係を理解できる
  - プロセスの相互関係の把握
  - プロセスたちがどのような関係で実行されているか把握できる
- アニメーションで表示
  - プロセスの生成と消滅の様子が理解できる

デモ

おわりに

- 研究の成果
  - Androidのプロセス可視化環境の開発
    - 複数あるプロセスの動作を把握
    - プロセス同士の関係性を理解
    - 可視化の表示速度の調整が可能
- 今後の予定
  - Java言語部分の可視化の表示
  - 可視化の表示部分の改良
  - 可視化表示のアニメーションの改良

クラウドシステム管理支援用可視化ツールの開発

落合 秀晴† 早川 栄一‡

1. 研究の背景と目的

仮想マシン提供型のプライベートクラウド[1]やハイブリッドクラウドの仮想環境を管理[2]するうえで問題となることは,すべての起動している仮想マシンの状態の把握,ユーザが仮想マシンをどの程度使用しているかを判断すること,仮想マシンを生成したマシンの特定に時間がかかることである.

問題を解決するためには仮想マシンと仮想マシンを稼働させている物理マシンの関連付けと状態の監視が必要である.しかし,従来のMRTG[3]や Ganglia[4]などのツールでは,直観的な状態の把握や仮想マシンと物理マシンの対応が把握しづらい問題がある.

2. 目的

本報告では上記背景の問題点を踏まえて,システムを構成する物理マシンと仮想マシンの各 CPU, メモリの利用率を可視化と物理マシンと仮想マシンとの対応の把握が可能なツールの開発を目的とした. クラウドシステム管理支援を目的とした可視化ツールの開発を行った.

なお, プライベートクラウドで仮想マシン提供型のクラウドとして Eucalyptus[5][6]を用いた.

3. 設計

3.1 可視化対象

Eucalyptus の構成は，仮想マシンを稼働させるノードコントローラ(以下 NC)と，それら NC 群を管理するクラスタコントローラ(以下 CC), ユーザからのリクエストを受け付けるクラウドコントローラ(以下 CLC)，ストレージや仮想マシンイメージを管理する(Walrus, SC)の 5 種類のサーバで構成されている.

†拓殖大学大学院　工学研究科
Graduate School of Engineering,Takushoku University
‡拓殖大学　工学部　情報工学科
Department of Computercience,Faculty of Engineering,
Takushoku University

本研究では特に最もスケールアウトする NC マシンとその上で稼働する仮想マシンを可視化対象とする.

3.2 可視化方法

可視化は NC と仮想マシンの利用率を把握できるように全体を棒グラフ化する．棒グラフを用いるのは，詳細な数値ごとに表示する線グラフに比べ，全体の傾向を読み取るのに適しているからである．棒グラフでは時系列ごとの状態を判断することができないが，今回は監視中で一定時間以上続くマシンの異常の特定を対象としたため，棒グラフを採用した.

物理マシンと仮想マシンの対応に関しては表を用いた可視化を行う．また，可視化ツールは Web ブラウザ上で閲覧可能なものとする.

3.3 全体構成

図 1 に全体構成を示す．可視化ツールは、データ取得部、データ加工部、可視化部の三つで構成される.

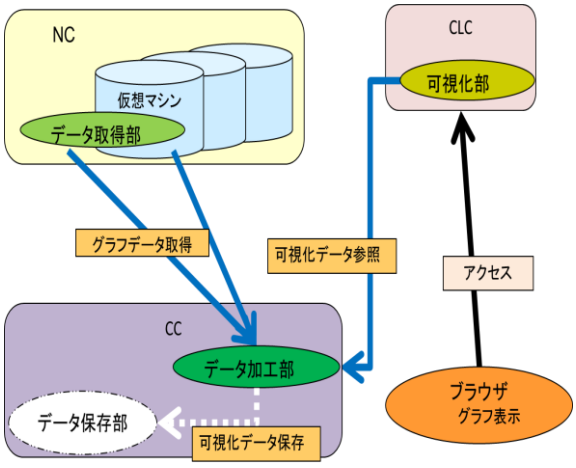


図 1.全体構成(データ取得部を NC と仮想マシンに, データ加工部を CC に可視化部を CLC に設置した例)

3.3.1　データ取得部

データ取得部は NC や仮想マシンのログから CPU，メモリ，ID，IP などのデータを取得し，取得したデータをデータ加工部に送る．

3.3.2　データ加工部

データ加工部は Eucalyptu 内部の状態を把握できるように euca2ools がインストールされているマシンに設置するものとする．

データ加工部は EucalyptusAPI や euca2ools から各 Eucalyptus を構成する物理マシンと仮想マシンとの対応に関するデータを取得し，それをもとに各 NC・各仮想マシンに設置したデータ取得部から送られたデータを JSON 形式にまとめる．

3.3.3　可視化部

可視化部はデータ加工部で生成された JSONデータを参照し，グラフの描画を行う．

ユーザは可視化部にアクセスすることで可視化ツールを利用する．

4.　実装

データ取得部は，一定時間ごとに Linux の proc ファイルから CPU・メモリのデータを，/sbin/ifconfig から IP データを取得する.この際 CPU は/proc/loadavg から 1 分間の平均値データを取得し,次の式(1)で CPU の使用率を算出する．

1 分間の loadavg／CPU コア数　　(1)

メモリに関しても同様に/proc/meminfo から総メモリと空きメモリのデータを取得し,次の式(2)でメモリの使用率を算出する．

1—(空きメモリ／総メモリ)　　(2)

上記で取得し,割り出した数値や IP などのデータを JSON データに変換した後にデータ加工部に送信する．

データ加工部は EucalyptusAPI や euca2ools をもとに各 NC と各仮想マシンの対応情報を取得する．次にデータ取得部から送られたデータを EucalyptusAPI から得た書くマシン間の対応に従い JSON データを生成する．

可視化部は JavaScript ライブラリの jQuery を使用し，グラフ描画にはプラグインの Highcharts を利用した．可視化部はデータ加工部で生成された JSONデータに Ajax を用いた非同期通信を行いリアルタイムに近い表示を行う．

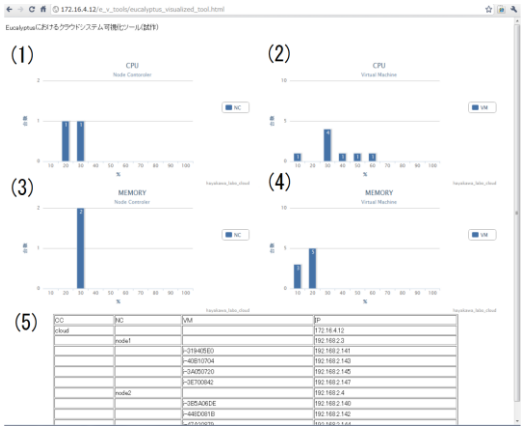


図 2.実行画面

図 2 に可視化ツールの実行画面を示す．(1)～(4)は縦軸にマシン台数，横軸は(1)(2)が CPU 利用率,(3)(4)がメモリ利用率を表示する．(5)は CC，NC，仮想マシンをツリー状にしたものを一覧表示する．

5.　まとめ

クラウドシステムの管理支援を目的とした，棒グラフを用いたサーバの状態把握と，表による仮想マシンと物理マシンとの対応把握が可能な可視化ツールの開発を行った．

今後の課題として，現時点での実装と評価．また，可視化情報を貯めておくデータ保存部の設計・実装や，保存したデータの可視化方法の検討である．

参　考　文　献

[1] 独立行政法人 情報処理推進機構「クラウドコンピューティング社会の基盤に関する研究会」報告書 http://www.ipa.go.jp/about/research/2009cloud/index.html

[2] 新麗「ネットワークシステム運用管理への応用」情報処理 50(12),2009.12

[3] MRTG.jp:http://www.mrtg.jp/

[4] Ganglia Monitoring System: http://ganglia.sourceforge.net/

[5] 日本 Eucalyptus ユーザーズグループ「Eucalyptus（OSS Elastic Computing)日本語情報」http://eucalyptus.linux4u.jp/wiki/index

[6] 羽深修・志田隆弘・田中智文「Eucalyptus ではじめるプライベートクラウド構築」インプレスジャパン 2011.6.1

クラウドシステム管理支援用可視化ツールの開発

†拓殖大学大学院工学研究科電子情報工学専攻
‡拓殖大学工学部情報工学科
落合　秀晴†
早川　栄一‡

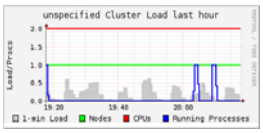
背景

- クラウドコンピューティングの普及
  - クラウド利用者の増加
  - 各企業・機関においてクラウドシステムの導入
- 導入する場合のクラウドシステム
  - プライベートクラウド
  - ハイブリッドクラウド

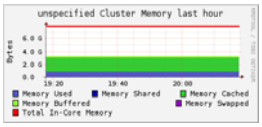
問題点

- クラウドシステム管理の煩雑さ
  - スケールアウトによるサーバの増加
  - 仮想化による仮想マシンの増加
  - サーバと仮想マシンの稼働状態の把握
- サーバと仮想マシンとの対応
  - 仮想マシンを生成しているサーバの特定

- 例）Gangliaの表示画面
  - CPU負荷



  - メモリ負荷



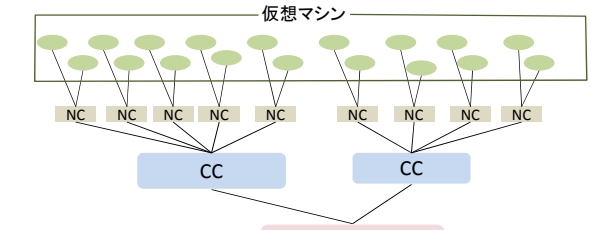
目的

- 管理者の手間を軽減
  - サーバ稼働状態の直観的な把握
  - 仮想マシンとサーバの関係を把握
- プライベートクラウド環境で実現
  - オープンソースであるEucalyptusを使用

可視化ツールの特徴

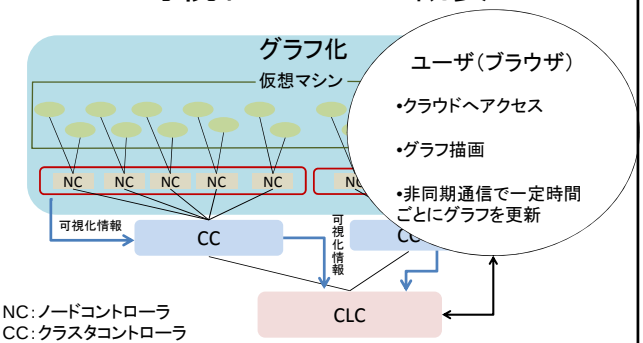
- Eucalyptus上で実現
  - プライベートクラウドに対応
- CPU及びメモリ使用率のグラフを用いた可視化
  - 棒グラフを用いて複数のマシンンの状態を表示
- 仮想マシンとノードマシンの関連性を把握
  - 表にすることで対応を把握

Eucalyptusの構成



NC：ノードコントローラ
CC：クラスタコントローラ
CLC：クラウドコントローラ

可視化ツールの概要



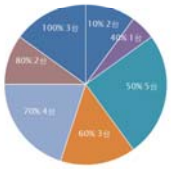
NC：ノードコントローラ
CC：クラスタコントローラ
CLC：クラウドコントローラ

設計方針

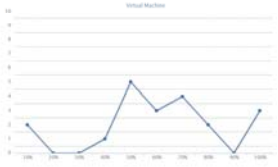
- Webブラウザ上で表示
  - インストール不要
  - ネットワークがつながればどこからでも監視可能
- NCと仮想マシンを対象
  - システム構成の中で最も台数が多い
  - ユーザに提供する部分

設計方針2

- システム全体に負荷がかかっているのか？
- 一部のマシンの負荷が高いのか？
  - 例）

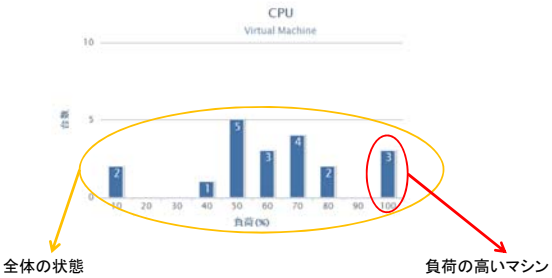


・円グラフの場合
-対象の台数が多いため0台や、少ない台数のものが見えなくなる可能性が高い



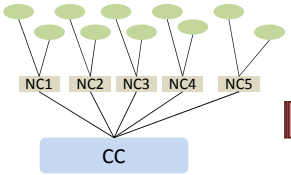
・折れ線グラフの場合
-各パーセンテージの点と点が繋がっているため見づらい

- 棒グラフによる可視化
  - マシン全体の大まかな状態把握
  - 負荷の高いマシンの把握



設計方針3

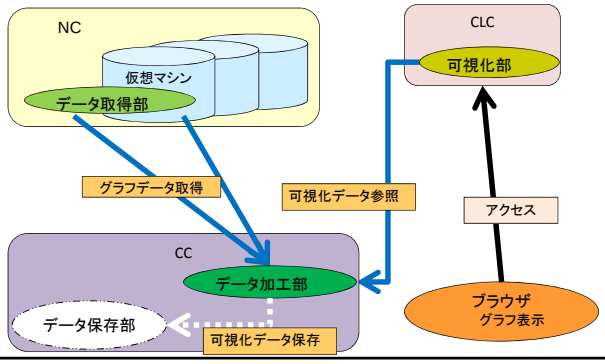
- CC,NC,仮想マシンの対応表
  - クラウド上での物理マシンの役割の把握
  - 仮想マシンと物理マシンとの対応の把握



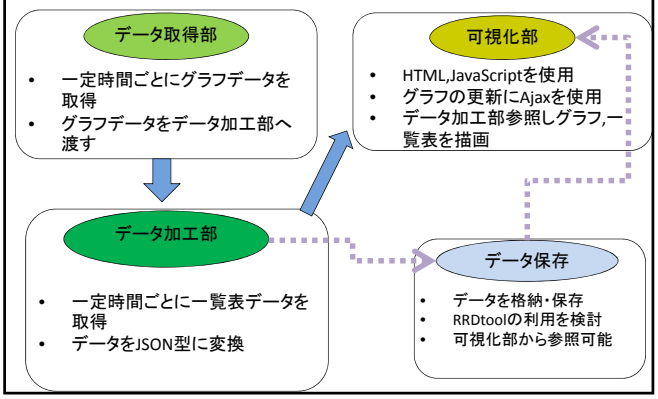
CC	----------	----------
----------	NC	----------
----------	----------	VM1
----------	----------	VM2
----------	NC2	----------
----------	----------	VM3
----------	----------	VM4
----------	NC3	----------

構成

例）データ取得部をNCとインスタンスに,データ化後部をCCに,可視化部をCLCのマシンに設置した場合の構成



可視化の流れ



可視化するデータ

- 可視化対象から取得する情報
  - CPU負荷
    - Linuxの/proc/loadavgから取得
    - 1分間の平均値をCPUのコア数で割った値を使用
  - メモリ負荷
    - Linuxの/proc/meminfoから取得
    - 1－（空きメモリ／総メモリ）の値を使用
  - IPアドレス
    - Linuxの/sbin/ifconfigから取得

可視化するデータ2

- EucalyptusAPIから取得する情報
  - CC,NC,VMに関する情報
    - 名前
    - IP
    - ID
    - NCの所属しているクラスター(CC)の名前
    - VMの所属しているノード（NC）の名前

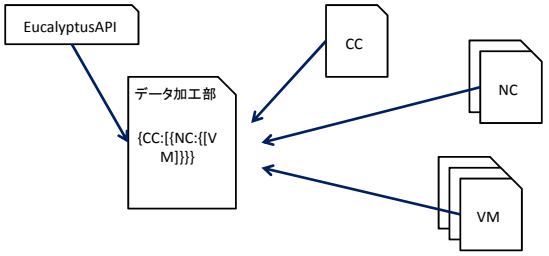
データの形式

- 可視化データはJSON形式

例）CC1台,NC1台,稼動しているVM2台の場合

{"CC" :
[{"NAME":"cc1","IP":"133.36.58.65","CPU":50,"MEMORY":40,
"NC":[{"NAME2":"nc1","IP2":"192.168.2.2","CPU2":40,"MEMORY2":30,
"VM":[{"NAME3":"vm1","IP3":"133.36.58.100","CPU3":30,"MEMORY3":20},
{"NAME3":"vm2","IP3":"133.36.58.101","CPU3":20,"MEMORY3":10
}]}]}]}

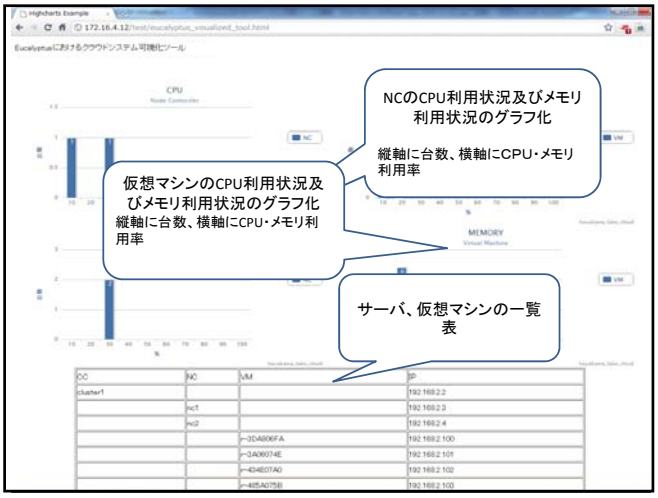
データの加工

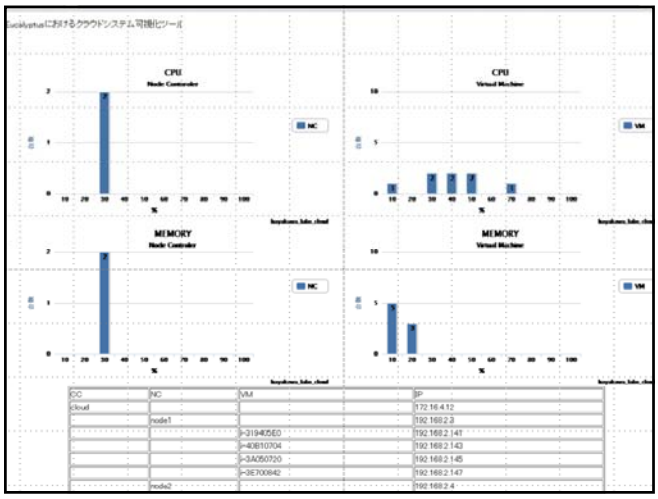


– データ加工部ではCC,NC,VMから受け取ったデータとEucalyptusAPIを元にJSONデータに変換

表示画面

- HTML及びJavaScriptを使用
  - JavaScriptライブラリ
    - jQuery
  - jQueryプラグイン
    - Highcharts
- グラフの更新はAjax機能を利用
  - jQueryのAjax用メソッドを使用





おわりに

- まとめ
  - 棒グラフを用いた可視化
  - 表による仮想マシンと対応したサーバの特定
- 今後の課題
  - 実装と評価
  - 保存部の設計

非同期システムコール機構の依存関係を持つシステムコール群への拡張の提案

安井 裕亮 †　　齋藤 彰一 †
† 名古屋工業大学

1　はじめに

システムコール発行時の割り込みを削減する手法として，非同期システムコール機構である Exception-Less System Calls[1] が提案されている．この手法では独立なシステムコール，すなわち依存関係を持たないシステムコールの一括処理によって高い効果を得ている．しかしその一方で適用可能なアプリケーションの範囲が狭いという問題点を持っている．

そこで本研究では，Exception-Less System Calls を拡張し依存関係を持つシステムコール群の一括処理機構を追加した手法である Sakura を提案する．Sakura により Exception-Less System Calls の適用範囲を広げることが本研究の目的である．

2　既存手法：Exception-Less System Calls

2.1　概要

Exception-Less System Calls はシステムコール発行時に割り込みを必要としない非同期システムコール機構である．これを Linux カーネル上に実装した FlexSC と共に提案されている．

Exception-Less System Calls は，システムコール処理専門のカーネルスレッドである **syscall thread** と，システムコール発行のインタフェースとして機能する共有メモリである **syscall page** という二つの構成要素から成る．ユーザスレッドは syscall page を介して syscall thread に対する要求を発行し，結果を受け取る．すなわちシステムコールは非同期に処理される．

2.2　問題点

Exception-Less System Calls は，その特長から図 1 に示すような独立なシステムコールを多く含むアプリケーションにおいて高い効果を得ているが，独立なシステムコールをあまり含まないアプリケーションでは効果が期待できないという問題点を持っている．一般にアプリケーション内のシステムコールは図 2 に示すような変数や制御構造を介して依存関係を持つものが多いため，Exception-Less System Calls を適用可能なアプリケーションの範囲は狭いといえる．

```
fd1 = open("a.txt");
fd2 = open("b.txt");
fd3 = open("c.txt");
```

図 1: 独立なシステムコール

```
fd = open("a.txt");
if (fd >= 0) {
  i = 0;
  while (i < 10000) {
    write(fd, "HELLO");
    i++;
  }
  close(fd);
}
```

図 2: 依存関係を持つシステムコール群

3　提案手法：Sakura

3.1　概要

Sakura は Exception-Less System Calls を拡張し依存関係を持つシステムコール群の一括処理機構を追加した手法である．Sakura では図 2 に示したような依存関係を持つシステムコール群をカーネル内で一括処理することができる．アプリケーションプログラマは Sakura が提供する API を用いて，カーネル内で一括処理させたい依存関係を持つシステムコール群を含む一連の手続を記述する．図 2 に示した一連の手続を Sakura の API を用いて記述したものを図 3 に示す．

これらの API の実体は関数やマクロであり，これらが実行されると，拡張された syscall page 上に依存関係を指定した状態で syscall thread に対する要求が発行される．

このように Sakura では依存関係を持つシステムコール群の一括処理が可能なため，独立したシステムコールをあまり含まないアプリケーションでも効果が期待できると考えている．

3.2　適用可能なアプリケーション

具体的に Sakura が適用可能なアプリケーションとしてはイベント駆動サーバが挙げられる．一般的なイベント駆動サーバーでは main スレッドにおいてイベントを捕獲し，その後スレッドを生成し，生成したスレッドにイベントを処理させるが，このスレッドに任せる

Yusuke YASUI †　Shoichi SAITO †
†Nagoya Institute of Technology

```
sakura_call(
CMM(
  ASN(VAR(fd), SYSCALL(SYS_open, "a.txt")),
  IF(isGE(VAR(fd), 0),
    CMM(
      ASN(VAR(i), 0),
      WHILE(isLT(VAR(i), 10000),
        CMM(
          SYSCALL(SYS_write, VAR(fd), "HELLO"),
          INC(VAR(i))
        )
      ),
      SYSCALL(SYS_close, VAR(fd))
    )
  )
)
);
```

図 3: Sakura の API を用いた記述

イベントの処理を 1 つ依存関係を持つシステムコール群に対応付けることによって，イベント駆動サーバは Sakura に適用可能であると考えている．

4 実装

本研究では Exception-Less System Calls の Linux 上の実装である FlexSC と提案手法 Sakura の実装を行った．Sakura は FlexSC の拡張であり，主に 3 つの拡張を施した．

1 つ目の拡張はシステムコール処理以外の演算や制御構造の定義である．システムコール処理要求以外に四則演算や比較演算，代入演算，カンマ演算などの演算と `if`，`while` といった制御構造を定義した．

2 つ目は syscall page のエントリ構造の拡張である．エントリの引数として定数だけでなく他のエントリへのポインタを持てるように拡張することで，他のエントリ上の要求の処理結果を引数にとることができるようにした．

3 つ目は syscall thread のアルゴリズムの拡張である．エントリの引数が他のエントリのポインタであった場合にそのエントリを先に処理するようにアルゴリズムを拡張した．これによりポインタでつながれたエントリ群が深さ優先で再帰的に処理されるようになっている．

5 評価

Sakura を Linux 上に実装し評価を行った．評価環境を表 1 に示す．評価項目は以下の通りである．

評価 1 単一の依存関係を持つシステムコール群の処理

評価 2 評価 1 の複数並列処理

表 1: 評価環境

CPU	Intel(R) Core(TM) i5 CPU 760 @ 2.80GHz
OS	Gentoo Linux
Kernel	Linux Kernel 2.6.38.4

表 2: 評価 1 の結果

手法	実行時間 (ms)	実行時間比
従来	11.638	1
Sakura	13.010	1.12

表 3: 評価 2 の結果

手法	実行時間 (ms)	実行時間比
従来	70.103	1
Sakura	96.423	1.38

なお，評価に用いた依存関係を持つシステムコール群は，図 2 で示したものである．評価 1，2 の結果を表 2, 3 に示す．

評価結果は，評価 1，2 のどちらの場合においても Sakura が従来のシステムコールに比べて低速であるということを示している．この原因は，カーネル内に実装したインタプリタの実行オーバーヘッドにあると考えられる．このオーバーヘッドを削減し，従来のシステムコールに対する Sakura の有効性を示すことが今後の課題である．これを達成するためにカーネル内インタプリタを廃止し，ネイティブ実行可能なコードを用いる手法を現在検討している．

6 まとめ

Exception-Less System Calls を拡張し依存関係を持つシステムコール群の一括処理機構を追加した手法である Sakura を提案した．Sakura により独立なシステムコールをあまり含まないアプリケーションにおいても効果が期待できると考えられる．現状ではオーバーヘッドが大きく，従来のシステムコールに対する有効性を示すことができていないため，このオーバーヘッドを削減することが今後の課題である．

参考文献

[1] Livio Soares, M. S.: FlexSC: Flexible System Call Scheduling with Exception-Less System Calls (2010), 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10).

非同期システムコール機構の依存関係を持つシステムコール群への拡張の提案

名古屋工業大学大学院
齋藤研究室
M1 安井裕亮

研究背景

- 従来のシステムコール
  - プロセッサの割り込みを用いた同期的な機構
  - 割り込みがアプリケーションの性能低下の原因となっている
    - 直接的なコスト：モード切り替え
    - 間接的なコスト：キャッシュの汚染、パイプラインフラッシュなど
- 既存 n-less system calls
  - 非 ル機構
  - 独立なシステムコールを多く含むアプリケーションで大きな効果を得ている
  - 独立なシステムコールを含まないアプリケーションでは効果が期待できず適用可能なアプリケーションの範囲が狭い

依存関係を持たない

提案手法：Sakura

- 既存手法を拡張し依存関係を持つシステムコール群の一括処理機構を追加
- 独立なシステムコールを含まないアプリケーションでも効果が期待できる
  - 既存手法に比べて適用可能なアプリケーションの範囲が広がる
    - 例）イベント駆動サーバなど

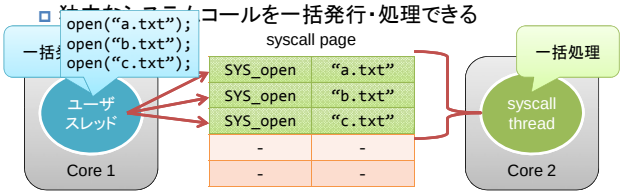
Exception-Less System Calls†

- 概要
  - システムコール発行時に割り込みを必要としない非同期システムコール機構
  - FlexSC：Exception-Less System CallsのLinux上の実装
- 構成要素
  - syscall thread
    - システムコール処理専門のカーネルスレッド
  - syscall page
    - システムコール発行のインタフェースとして機能する共有メモリ

†FlexSC: Flexible System Call Scheduling with Exception-Less System Calls, USENIX OSDI 2010

Exception-Less System Callsの特長

- システムコールの発行と実行が分離される
  - システムコールの発行と実行を別のコア上で処理できる
  - システムコールを一括発行・処理できる



- 独立なシステムコールを多く含むアプリケーションに対して有効
  - 複数のスレッド間の独立したシステムコールを集約する枠組みを提供しマルチスレッドのアプリケーションで大きな効果を得ている

Exception-Less System Callsの問題点

- 独立なシステムコールを含まないアプリケーションでは効果が期待できない
- 一般にアプリケーション内に独立なシステムコールは

提案
依存関係を持つシステムコールに着目
依存関係を持つシステムコール群の一括処理機構を導入して
適用可能なアプリケーションの範囲を広げる

```
fd1 = open(“a.txt”);
fd2 = open(“b.txt”);
fd3 = open(“c.txt”);
```

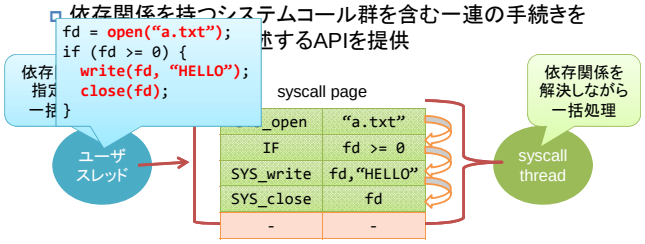
独立なシステムコール

```
fd = open(“a.txt”);
if (fd >= 0) {
  i = 0;
  while (i < 10000) {
    write(fd, “HELLO”);
    i++;
  }
  close(fd);
}
```

依存関係を持つシステムコール群

提案手法：Sakura

- exception-less system callsを拡張し依存関係を持つシステムコール群の一括処理機構を追加
  - 依存関係を持つシステムコール群を含む一連の手続きを記述するAPIを提供



- 独立したシステムコールを含まないアプリケーションでも効果が期待できる
  - 適用可能なアプリケーションの範囲が広がる

SakuraのAPI

- 演算や制御構造を記述するためのマクロ及び関数を用いて一連の手続きを記述

```
fd = open(“a.txt”);
if (fd >= 0) {
  i = 0;
  while (i < 10000) {
    write(fd, “HELLO”);
    i++;
  }
  close(fd);
}
```

従来のシステムコールを用いた記述

マクロ・関数	意味
CMM	カンマ演算（複合文）
ASN	代入演算
isGE	>=
isLT	<
SYSCALL	システムコール呼出

```
sakura_call(
CMM(
  ASN(VAR(fd), SYSCALL(SYS_open, “a.txt”)),
  IF(isGE(VAR(fd), 0),
    CMM(
      ASN(VAR(i), 0),
      WHILE(isLT(VAR(i), 10000),
        CMM(
          SYSCALL(SYS_write, VAR(fd), “HELLO”),
          INC(VAR(i))
        )
      ),
      SYSCALL(SYS_close, VAR(fd))
    )
  )
)
);
```

SakuraのAPIを用いた記述

適用可能なアプリケーション

- 例）イベント駆動サーバ
  - イベントの処理を一つの依存関係を持つシステムコール群に対応付ける
  - スレッドを生成するコストも削減できる

※擬似コード

```
main() {
  while (1) {
    sd = accept();
    thread_create(thread, sd);
  }
}

thread(int sd) {
  fd = open(“index.html”);
  do {
    size = read(fd, buf);
    send(sd, buf, size);
  }
  while (size > 0);
  close(fd);
}
```

イベントの処理

```
main() {
  while (1) {
    sd = accept();
    sakura_call(
      fd = open(“index.html”);
      do {
          size = read(fd, buf)
          send(sd, buf, size);
      }
      while (size > 0);
      close(fd);
    );
  }
}
```


実装

- FlexSC（exception-less system callsの実装）
  - syscall page
  - syscall thread
  - ユーザスレッドが利用するシステムコール
- Sakuraの実装
  - syscall page及びsyscall threadを拡張

FlexSC：Syscall Page

- ユーザスレッドとsyscall threadの間の共有メモリ
  - ユーザスレッド毎に1つずつ割り当てられる
  - システムコール発行のためのエントリの配列

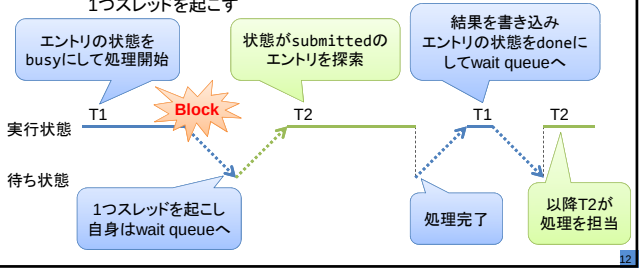
syscall number	status	arg1	…	arg6	return value
SYS_write	submitted	4	…	-	-
SYS_open	busy	“a.txt”	…	-	-
-	free	-	…	-	-
SYS_write	submitted	3	…	-	-
SYS_read	done	5	…	-	256
-	free	-	…	-	-
-	free	-	…	-	-

エントリの状態

status	意味
free	空き
submitted	未処理
busy	処理中
done	完了

FlexSC：Syscall Thread

- システムコール処理専門のカーネルスレッド
  - syscall pageのエントリ数だけ生成される
    - 基本的に1つスレッドのみが動作し他のスレッドはsyscall thread用のwait queueで待機
    - システムコール処理中に待ち状態に移行するときにwait queueから1つスレッドを起こす



FlexSC：ユーザスレッド用のシステムコール

- flexsc_register()
  - syscall pageを割り当てsyscall threadをエントリ数分生成
  - プログラムの冒頭で1度だけ呼び出す
- flexsc_wait()
  - 結果を待つためのシステムコール
  - 独立なシステムコールを可能な限り発行したのちに呼び出す

※これら2つのシステムコールは従来の同期式

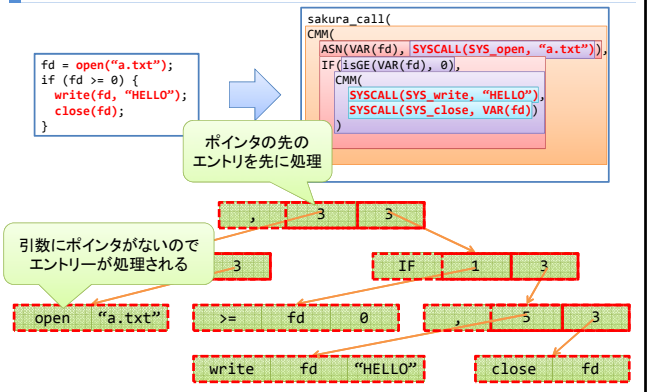
13

Sakuraの実装

- システムコール処理以外の演算や制御構造の定義
  - 四則演算、比較演算、代入演算、カンマ演算、if、whileなど
- syscall pageのエントリ構造の拡張
  - 引数に他のエントリへのポインタを持てるようにする
  - 他のエントリの処理結果を引数にとることができる
- syscall threadのアルゴリズムの拡張
  - 引数が他のエントリへのポインタであった場合にそのエントリを先に処理するようにする
  - 深さ優先で再帰的に処理

14

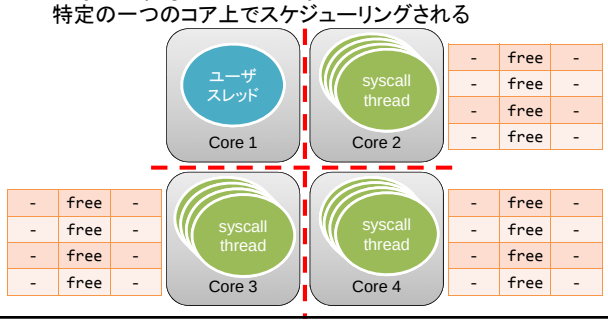
Sakuraの動作例



15

Syscall PageとSyscall Threadの割り当て

- 一つのユーザスレッドに(コア数 – 1)個のsyscall pageを割り当てる
  - 各syscall pageに対応するsyscall threadのセットは常に特定の一つのコア上でスケジューリングされる



16

評価

- 評価環境

CPU	Intel® Core™ i5 CPU 760 @ 2.80 GHz (2 core HT)
OS	Gentoo Linux
Kernel	Linux Kernel 2.6.38.4

- 評価項目
  1. 単一の依存関係を持つシステムコール群の処理での評価
     - 従来のシステムコール、FlexSC、Sakuraの3つで評価
  2. 複数並列処理での評価
     - 従来のシステムコール、Sakuraの2つで評価

17

評価対象プログラム

- 評価1（単一処理）

システムコール間に依存関係があるのでシステムコール発行の度に呼び出す必要がある

※擬似コード

```
fd = open("a.txt");
if (fd >= 0) {
  i = 0;
  while (i < 10000) {
    write(fd, "HELLO");
    i++;
  }
  close(fd);
}
```

従来のシステムコール

```
flexsc_open("a.txt");
flexsc_wait(&fd);
if (fd >= 0) {
  i = 0;
  while (i < 10000) {
    flexsc_write(fd, "HELLO");
    flexsc_wait();
    i++;
  }
  flexsc_close(fd);
  flexsc_wait();
}
```

FlexSC

```
sakura_call(
  fd = open("a.txt");
  if (fd >= 0) {
    i = 0;
    while (i < 10000) {
      write(fd, "HELLO");
      i++;
    }
    close(fd);
  }
);
sakura_wait();
```

Sakura

18

評価対象プログラム

- 評価2（複数並列処理）

処理毎に異なるファイルへ書き込み

```
thread(int arg)
{
 fd = open(filename[arg]);
  if (fd >= 0) {
    i = 0;
    while (i < 10000) {
      write(fd, “HELLO”);
      i++;
    }
    close(fd);
  }
}

main()
{
  for (i = 0; i < 16; i++)
    pthread_create(thread, i);

  for (i = 0; i < 16; i++)
    pthread_join();
}
```

16個の並列処理

従来のシステムコール

```
for (j = 0; j < 16; j++)
  sakura_call(
    fd[j] = open(filename[j]);
    if (fd[j] >= 0) {
      i[j] = 0;
      while (i[j] < 10000) {
        write(fd[j], “HELLO”);
        i[j]++;
      }
      close(fd[j]);
    }
  );

for (j = 0; j < 16; j++)
  sakura_wait();
```

Sakura

19

評価結果

- 評価1（単一処理）

手法	実行時間 (ms)	実行時間比
従来	11.638	1
FlexSC	30.243	2.60
Sakura	13.010	1.12

- 評価2（複数並列処理）

手法	実行時間 (ms)	実行時間比
従来	70.103	1
Sakura	96.423	1.38

20

オーバーヘッドの評価

- 評価方法

実行時間がSakuraのオーバーヘッド

```
sakura_call(
  fd = open(“a.txt”);
  if (fd >= 0) {
    i = 0;
```

```
sakura_call(
  fd = getpid();
  if (fd >= 0) {
    i = 0;
```

結論
オーバーヘッドが顕著に現われ
割り込み削減による効果が得られていない

- 評価結果

プログラム	実行時間 (ms)	差分 (ms)
write	13.010	
getpid	1.695	11.315

21

まとめと今後の予定

- 提案手法：Sakura
  - exception-less system callsを拡張し依存関係を持つシステムコール群の一括処理機構を追加
    - 依存関係を持つシステムコール群を含む一連の手続きを一つの要求として記述するAPIを提供
  - 独立なシステムコールを含まないアプリケーションでも効果が期待できる
- 今後の予定
  - イベント駆動サーバでの評価
  - オーバヘッドの削減

22

耐障害性を有するマルチカーネル OS の設計

加藤　雄大
名古屋工業大学大学院工学研究科

1　はじめに

計算機の信頼性への要求は年々高まっている. 信頼性を脅かす要因は様々であるが, 本研究では OS のカーネルクラッシュに着目した. カーネルクラッシュが発生すると全てのプロセスが停止し, 再起動する以外に対処法がない. その結果, 処理途中のデータが欠落したり, サービスが一時的に滞ることでユーザに大きな損害を与える可能性がある. しかしカーネルクラッシュを防ぐのは難しい. それは他のソフトウェア同様にバグを完全に取り除くことが難しいことや, ハードウェアの故障によっても影響を受けることが原因である. 本研究ではカーネルクラッシュを防ぐのではなく, カーネルクラッシュが発生した場合にプロセスの実行を継続することで被害を最小限に留めることを目的としている.

2　関連研究

カーネルクラッシュからプロセスの実行を保護する手法として Otherworld[1] が提案されている. Otherworld はカーネルがクラッシュした際に, カーネルをマイクロリブートすることで, プロセスの実行を再開する. マイクロリブートにより実行されたカーネルはクラッシュしたカーネルから実行継続に必要な最低限の状態を引き継ぐことでプロセスの実行を継続する. fault を自動的に挿入する実験において９７パーセントの確率でアプリケーションの実行継続に成功している. またオプションでアプリケーション毎に定義される crash procedure を登録することで復旧の確率を高めることができる. また懸念される処理負荷だが, 先述のオプション機能を用いなければ, クラッシュ時以外に処理を行わないため通常実行時に処理負荷をかけない.

3　提案

本研究は複数のカーネルを持った OS（マルチカーネル OS）と Otherworld 手法を応用し, カーネルクラッシュからプロセスの実行を保護する手法を提案する.

一般的な OS には一つのカーネルしか持たない. そのため唯一のカーネルがクラッシュすることで計算機上の全プロセスが停止することになる. この問題を解決するために本研究では OS を複数のカーネルで構成することでカーネルクラッシュにより停止するプロセス数の削減を狙う. またクラッシュしたカーネル上のプロセスを Otherworld 手法を応用し, 救出することで高い耐障害性を有する OS を構築する.

4　マルチカーネル OS

現在, メニーコアでスケールする OS としてマルチカーネル OS の研究が行われている [2][3]. Barrelfish[2] はヘテロジニアスメニーコアでスケールするように設計されている. Factored Operating System[3] はメニーコアとクラウドコンピューティングという二つのトレンドに適応するように設計されている. どちらもハードウェアの進歩, 利用形態の変化に対応することを目的に複数のカーネルで OS を構成している. 本研究はカーネルが単一故障点となる現在の OS の構造的問題を解決するためにマルチカーネル OS を用いる.

4.1　複数のカーネルを同時に実行する機構

マルチカーネル OS を実現するためには複数のカーネル (OS) を走行する機構が必要になる. 複数の OS を同時に走行させる機構として VMM の利用するのが一般的だが, 本研究ではソフトウェアによる LPAR 方式 [4] により複数カーネルの実行を実現している. この方式はメモリ, CPU コア, デバイスを物理的な単位に分割し, それぞれのカーネルに占有させることで複数の OS を同時に実行する. この方式はハードウェアを操作する時に仮想化機構を用いないため, ハードウェアアクセスにオーバヘッドを伴わないという特長をもっている. また動的に各カーネルの占有するデバイスを変更することが可能であり, この特徴を利用することで提案手法におけるデバイスの移植を実現することができる. デバイス移植ついては 6 章で述べる.

5　プロセスの救出

3 章で述べたように本研究は既存研究である Otherworld を応用しクラッシュしたカーネルからプロセスを救出する. 本手法は既存手法に比べて救出するプロセスの数が少ないこと, プロセスの救出に必要な時間が短いことの二点で優れていると言える.

マルチカーネル OS においてカーネルクラッシュの影響はそのカーネル上のプロセスに限定される. 例えばカーネルを N 個持ったマルチカーネル OS があるとすれば, カーネルクラッシュにより停止するプロセスの数は全体の N 分の 1 になると期待できる. 2 章で述べたように既存手法は高い確率でプロセスを救出できるが, 不完全なところもある. 本提案は救出するプロセスの数を減らすことで救出に失敗する確率を低くすることができる.

また既存手法はプロセスの救出のために新しいカーネルをマイクロリブートにより起動する. マイクロリブートには kexec[5] が用いられるが, kexec による再起動は通常の再起動時間から BIOS 起動時間を差し引いた時間が必要であり, 救出までに長い時間がかかると言える. 本提案手法では実行中のカーネルがプロセスの救出に当たる. そのためマイクロリブートが不要であり, プロセス救出時間を大幅に削減できる. また先述のように救出が必要なプロセスの数を削減できるので救出処理自体に必要な時間も短縮できる.

6 プロセス救出機構の動作

システムはマルチカーネル OS とその上に実現されるプロセス救出機構によって構成される. 5 章で述べたようにプロセスの救出はクラッシュしたカーネル以外のカーネルが行う.

ここで便宜上プロセスの救出を行うカーネルをレスキューカーネルと呼ぶ. プロセスの救出機構は前述のように Otherworld の手法を応用することで実現する. レスキューカーネルはクラッシュしたカーネルからプロセスに関するデータを取得し, そのデータを元にレスキューカーネル上にプロセスを復元する.

提案手法ではプロセスが利用していたデバイスをクラッシュしたカーネルからレスキューカーネルに移植する必要がある. これはそれらのプロセスが救出後に同じデバイスを使用するためであり, これにより例えば NIC によって外部と通信していたプロセスは救出後も同じネットワークアドレスで通信することができる. デバイスの移植は既存手法では必要はない. なぜならばシングルカーネルなので全てのデバイスをレスキューカーネルが占有するため, デバイスの専有に関して特別な配慮が不要だからである.

7 実装進捗

現在複数カーネルの同時実行機構の実装が完了している. 実装は Mint オペレーティングシステム [6] を参考にしている. 使用している Linux カーネルのバージョンは 2.6.38.7, 計算機の CPU は Intel(R) Core(TM) i5@2.80GHz であり, ２つのカーネルにそれぞれ論理コアを２つずつ割り当てている. またデバイスは NIC とディスクコントローラを一つずつ割り当てている. 今後はプロセスの救出機構の実装を行う予定である.

8 まとめ

計算機の信頼性を高めるべく, 耐障害性を有するマルチカーネル OS を提案した. 複数のカーネルで OS を構成することでカーネルクラッシュの影響を縮小し, プロセスの救出機構を用いることでプロセスの実行停止を防ぐことが狙いである. プロセスの救出には Otherworld 手法を応用することで, 高確率でプロセスを救出可能である. 本手法では既存手法よりもプロセスの救出に必要な時間を大幅に短縮する. 実装が終わり次第, バグの挿入実験による提案手法の耐障害性の評価を行う予定である.

参考文献

[1] Alex Depoutovitch and Michael Stumm. ”otherworld”: Giving applications a chance to survive os kernel crashes. In *EuroSys*, pp. 181–194, 2008.

[2] Andrew Baumann, Paul Barham, Pierre variste Dagand, Timothy L. Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Symposium on Operating Systems Principles*, pp. 29–44, 2009.

[3] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *Operating Systems Review*, Vol. 43, pp. 76–85, 2009.

[4] Taku Shimosawa, Hiroya Matsuba, and Yutaka Ishikawa. Logical partitioning without architectural supports. In *International Computer Software and Applications Conference*, pp. 355–364, 2008.

[5] Andy Pfiffer. Reducing system reboot time with kexec. http://www.osdl.org/.

[6] 中原大貴, 千崎良太, 牛尾裕, 片岡哲也, 乃村能成, 谷口秀夫. Kexec を利用した mint オペレーティングシステムの起動方式. 電子情報通信学会技術研究報告. CPSY, コンピュータシステム, Vol. 110, No. 278, pp. 35–40, 2010-11-05.

耐障害性を有する
マルチカーネルの設計

名古屋工業大学大学院
加藤雄大

発表概要

2

- 背景
- 提案
- 既存手法
  - マルチカーネルOS
  - Otherworld手法
- Otherworldと提案の比較
- 提案システムの設計と動作
  - 設計
  - 動作
- 実装状況

背景

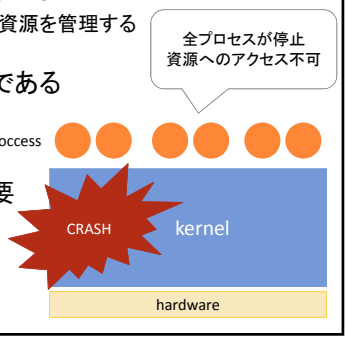
3

- 計算機に高い信頼性が求められている
  - コア数の増加により一台で提供可能なサービス数が増加
  - 一台の停止による影響が増大
- OSの障害が問題になる
  - カーネルクラッシュは再起動以外に対処不可能
    - 揮発性データの消失
    - サービスが一時的な停止

カーネル　単一障害点

4

- 問題
  - 一般的なOSはカーネルが一つ
    - 一つのカーネルが全ての資源を管理する
- カーネルは単一障害点である
  - 冗長性の欠如
- カーネルに冗長性が必要



発表概要

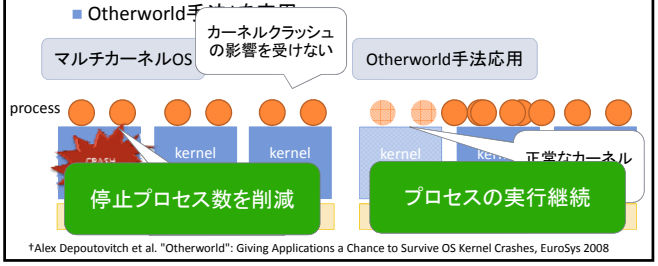
5

- 背景
- 提案
- 既存手法
  - マルチカーネルOS
  - Otherworld手法
- Otherworldと提案の比較
- 提案システムの設計と動作
  - 設計
  - 動作
- 実装状況

提案

6

- カーネルをフェイルオーバーする機構を備えたOS
  - マルチカーネルOS（複数のカーネルを有するOS）
  - フェイルオーバー機構
    - クラッシュ後にプロセスを正常なカーネルに移し実行を継続する
    - Otherworld[†]



[†]Alex Depoutovitch et al. "Otherworld": Giving Applications a Chance to Survive OS Kernel Crashes, EuroSys 2008

発表概要

7

- 背景
- 提案
- 既存手法
  - マルチカーネルOS
  - Otherworld手法
- Otherworldと提案の比較
- 提案システムの設計と動作
  - 設計
  - 動作
- 実装状況

マルチカーネルOS

8

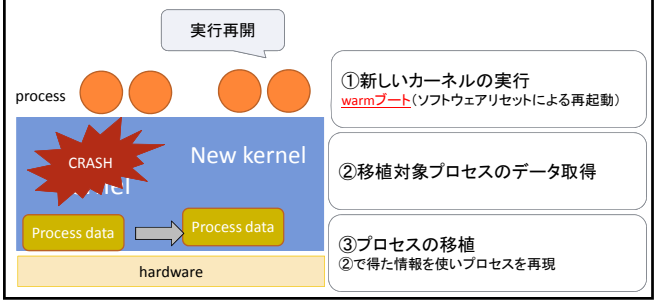
- メニーコアやヘテロジニアスコアに適したOSの研究が進んでいる
  - Barrelfish†
    - ヘテロジニアスメニーコアでスケール
  - Factored operating system‡
    - メニーコアでスケール
    - クラウドコンピューティングに適した構成
- 本研究では耐障害性に着目する

†Adrian Schüpbach et al. , The multikernel: a new OS architecture for scalable multicore systems, SOSP2009
‡David Wentzlaff et al., An operating system for multicore and clouds, Proceedings ACM SOCC (2010)

Otherworld概要

9

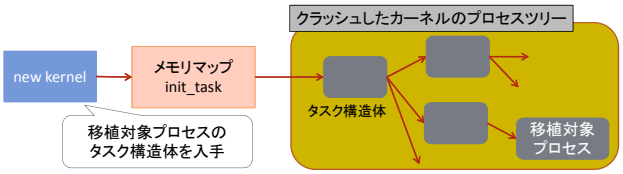
- カーネルクラッシュによるプロセスの停止を防ぐ
- 新しいカーネルでクラッシュしたカーネルを置換



移植対象プロセスのデータ取得

10

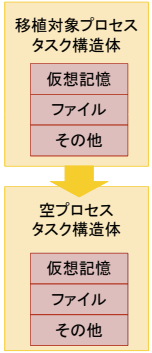
- Warmブートにより主記憶上のデータが温存
  - クラッシュ時のデータが取得可能
- 移植対象のプロセスのタスク構造体を取得



プロセスの移植

11

- 空プロセスを移植対象のプロセスに作り替える
  - 仮想記憶の貼り替え
    - ページテーブルのクローンを作成
    - 物理メモリの内容をコピー
  - ファイルを再オープン
    - 移植対象プロセスが開いていたファイルを開く
    - 読み書きしていた場所までポインターを進める
  - その他
    - 物理ターミナルなども復元
    - ネットワークコネクションの復元は課題として残されている



発表概要

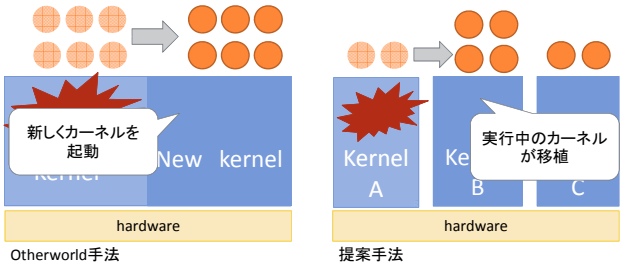
12

- 背景
- 提案
- 既存手法
  - マルチカーネルOS
  - Otherworld手法
- Otherworldと提案の比較
- 提案システムの設計と動作
  - 設計
  - 動作
- 実装状況

提案手法

13

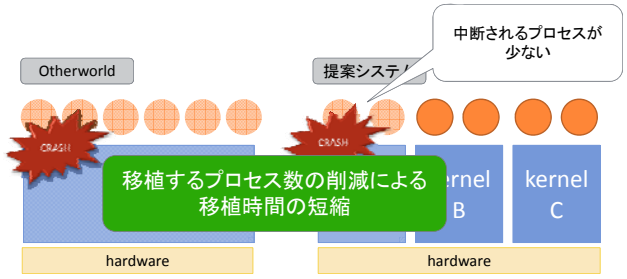
- 起動済み他カーネルによる停止プロセスの移植
  - カーネルクラッシュ発生時でも多くのプロセスは正常実行
  - プロセス移植に必要な時間を大幅に短縮



提案:プロセス移植時間の短縮(1/2)

14

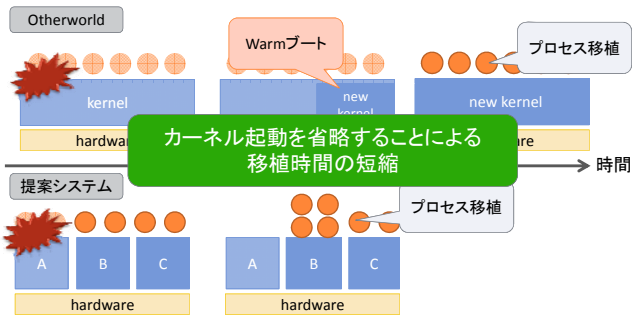
- 中断されるプロセス数の削減による移植時間の短縮



提案:プロセス移植時間の短縮(2/2)

15

- クラッシュ時に新しいカーネルの実行が不要
  - プロセスの再開までにかかる時間が短縮される



発表概要

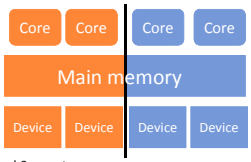
16

- 背景
- 提案
- 既存手法
  - マルチカーネルOS
  - Otherworld手法
- Otherworldと提案の比較
- 提案システムの設計と動作
  - 設計
  - 動作
- 実装状況

提案システムの設計(1/2)

17

- マルチカーネル実装方法
  - Software LPAR(Logical PARtitioning)†
    - CPUコア、メモリ、デバイスを分割しカーネルに割り当てる
    - ハードウェアではなくソフトウェア(カーネル)で実現
  - 特長
    - 仮想化と異なりオーバヘッドが無い
    - 分割されたハードウェアの譲渡が容易
      - ハードウェアアクセスの排他制御は各カーネルが行う
      - 必要に応じて他のパーティションへのアクセスが可能



†Taku Shimosawa et al., Logical Partitioning without Architectural Supports, Computer Software and Applications Conference, Annual International(2008)

提案システムの設計(2/2)

18

- 必要な機能
  - クラッシュ時にハードウェアをカーネル間で譲渡する機能
  - ユーザプロセスのPIDがOS全体で一意になるようにする
    - 移植時にPIDが衝突しないようにするため
    - PIDのプレフィックスとしてカーネルIDを使う

提案システム動作概要

19

- OS起動時
  - クラッシュ検知機構を始動
- クラッシュ発生時
  - デバイス移植
    - クラッシュしたカーネルから移植先のカーネルへ
  - Otherworld手法を使いプロセスを移植

クラッシュ検知機構の始動

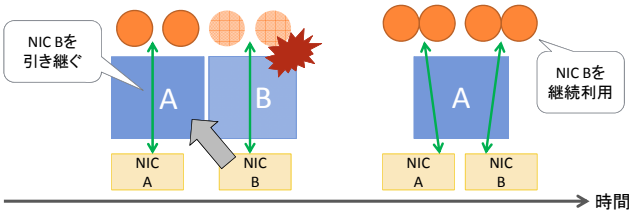
20

- OSの起動
  - 各カーネルをコア、メモリ、デバイスを与えて起動
  - 各カーネルに起動順でカーネルIDを付与
- クラッシュ検知機構の始動
  - カーネル同士で定期的な死活確認
    - メッセージの送受信
    - 返信が無い場合はクラッシュしたとみなす

クラッシュ発生後　デバイス付替

21

- デバイスの付け替え
  - クラッシュしたカーネルのデバイスをプロセス移植先のカーネルに移す
    - プロセスの再開後も同じデバイスを使用可能にするため



クラッシュ発生後　プロセス移植

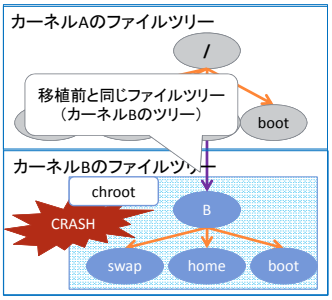
22

- Otherworld手法をマルチカーネルOS上で行う
- ファイルの再オープンに工夫が必要
  - ディスクのマウント
    - 引き取ったディスクのルートディレクトリをどこかにマウント
      - マウントポイントの例：/mnt/B/
  - 課題
    - 再オープンするファイルのパスをマウントポイントを起点にしたものに置きかえる必要がある
      - 例：/home/hoge.txt⇒/mnt/B/home/hoge.txt

ルートディレクトリの変更

23

- 解決方法
  - 移植されるプロセスのルートディレクトリを変更する
  - chrootを使う
- 移植先カーネルの動作
  1. クラッシュしたカーネルからディスクを引き取る
  2. ディスクをマウント
  3. プロセスを移植
     - Otherworld手法
  4. 引き取ったプロセスのルートディレクトリを変更



発表概要

24

- 背景
- 提案
- 既存手法
  - マルチカーネルOS
  - Otherworld手法
- Otherworldと提案の比較
- 提案システムの設計と動作
  - 設計
  - 動作
- 実装状況

実装進捗と予定

25

- Software LPAR
  - Mintオペレーティングシステム†を参考に実装
  - Linux-2.6.38を使用
  - カーネルオプションで各カーネルが使用可能なCPUコア、メモリ、デバイスを指定
  - kexecによって二番目のカーネルが起動
  - CPUコアとメモリの分割が完了
  - デバイスの分割を実装中

†中原 大貴ら，Kexecを利用したMintオペレーティングシステムの起動方式，
電子情報通信学会技術研究報告, vol.110, no.278, pp.35-40 (2010.11).

まとめと今後の予定

26

- 耐障害性を有するマルチカーネルOSを設計した
  - 複数のカーネル上のプロセスをOtherworld手法で移植することでカーネルクラッシュからプロセスを保護
  - プロセスの移植に必要な時間を短縮
- 予定
  - Otherworld手法をマルチカーネル上で実装
  - 評価
    - プロセス移植に必要な時間の計測
    - 耐障害性評価

プロセスを分散実行するためのシステムコール制御に関する検討

三添 匠 † 小鍛治 翔太 † 芝 公仁 ††

† 龍谷大学大学院理工学研究科 †† 龍谷大学理工学部

1 はじめに

近年，プロセッサの性能向上に伴い複数のスレッドを使用して動作するアプリケーションが増加している．しかし，一般的なシステムでマルチスレッドのプロセスを実行した場合，単一の計算機上でしかそのプロセスを動作させることができない．本研究では，マルチスレッドのアプリケーションを単一の計算機内だけで処理させるのではなく，複数の計算機を使用して動作させることが可能となるプロセス共有機構を構築している．

現在，我々が開発を行っているプロセス共有機構では，単一のプロセスを複数の計算機上にまたがって動作させることができ，プロセスが持つスレッドを各計算機に分散させることが可能である．これにより，プロセスは同時に複数の計算機の資源を利用することが可能となる．これは, プロセスの持つアドレス空間を計算機間で共有することで実現される [1]．すなわち，ある計算機でメモリへの書き込みが行われると他の計算機でもそれを読み出すことができる．また，システムコールは，それを処理すべき計算機に転送され実行される．これら，メモリの一貫性制御，システムコールの転送は自動的に行われ，プロセスは分散処理を意識する必要がなく，既存のアプリケーションをそのまま複数の計算機で動作させることができる．

本稿では，特に，各計算機上のスレッドから発行されたシステムコールをそれを実行すべき計算機に転送し実行するシステムコール制御について述べる．これにより，スレッドはどの計算機からでもシステムコールを発行することが可能となり，位置透過にプロセスのコードを分散実行することが可能となる．

2 プロセス共有機構

プロセス共有機構は，単一のプロセスを複数の計算機で共有することを可能とする．プロセス共有機構の構成を図 1 に示す．プロセス共有機構は，以下の 2 つから構成される．

- メモリ制御部
- システムコール制御部

メモリ制御部は，計算機間で共有するプロセスの持つアドレス空間の共有を実現する．これは，プロセス

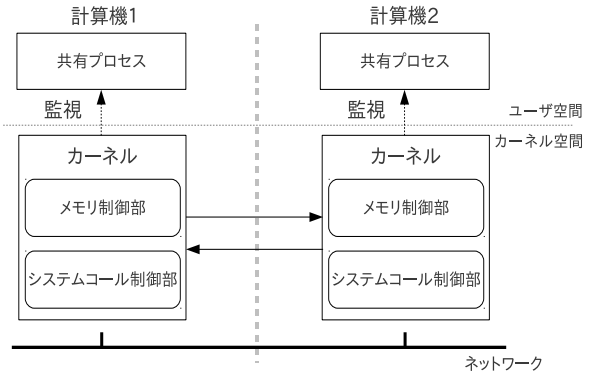


図 1 システム構成

を監視し，ある計算機でアドレス空間に書き込みが完了すると，他の計算機からでも書き込まれた値を読み出せるようにメモリの一貫性制御 [2] を行う．

システムコール制御部は，共有プロセスの持つスレッドが発行するシステムコールに対して，各システムコールの実際の処理が実行される前に実行され，転送と実行を行う．また，他の計算機のプロセス共有機構と以下の通信を行う．

- システムコール実行要求
- システムコール終了通知
- スレッド生成通知

システムコール実行要求では，システムコールの実行に必要なシステムコール番号とシステムコールの引数を，システムコールを実行する計算機のシステムコール制御部に送信する．システムコール番号と引数は，システムコールが発行された際のレジスタの値を参照することにより取得する．システムコール終了通知では，実行したシステムコールの戻り値を送信する．スレッド生成通知では，スレッドが生成された際に，プロセス番号，アドレスを送信する．これにより，共有プロセス内で生成されるスレッドを管理する．

3 システムコール実行

システムコールは，システムコール制御部により処理される．共有プロセス内でスレッドが生成されると，共有プロセスに属する実行スレッドがカーネル内に生成され，転送されるシステムコールはこの実行スレッ

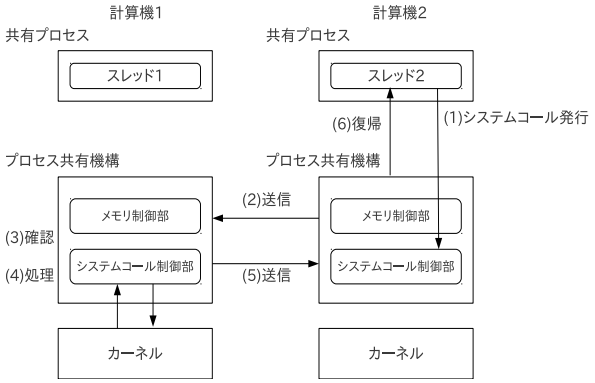


図 2　システムコール転送

ドにより処理される．この共有プロセスに属している実行スレッドで処理することにより，他の計算機で生成されたスレッドが発行したシステムコールを共有プロセスから発行されたかのように処理することができる．

図 2 に計算機 1 と計算機 2 でプロセスを共有し，共有プロセスのスレッド 1 を計算機 1，スレッド 2 を計算機 2 で実行させる様子を示す．スレッド 2 から発行されたシステムコールは，以下のように計算機 1 に転送され，実行される．

(1) スレッド 2 からシステムコールが発行される
(2) システムコール制御部が発行されたシステムコールを実行する計算機を調べ，その計算機にシステムコール実行要求を送信する
(3) システムコール番号が有効なものか確認する
(4) 処理する実行スレッドを判断し，システムコールを処理する
(5) 戻り値を送信する
(6) 戻り値をレジスタに格納し，ユーザモードの共有プロセスに復帰する

(2) においてシステムコールを処理する計算機を調べた際，システムコールを転送する場合はシステムコール実行要求を送信し，転送しない場合はシステムコールが発行された計算機で処理する．また，システムコール実行要求を送信したあとは，システムコールを発行したスレッドの状態を実行状態から待ち状態に移行させ，戻り値を受信するまで停止させる．

(4) において転送した先でシステムコールを処理する際，プロセスの持つアドレス空間を参照する場合でも，メモリ制御部により各計算機でメモリの一貫性が保証されているためシステムコールを処理することが可能である．これは，参照するメモリページを持っていなくても，他の計算機から取得しメモリページの矛盾なく参照が可能なためである．

表 1　システムコール処理時間

	A	B	C
open	0.275ms	0.279ms	4.304ms
mkdir	1.355ms	1.365ms	1.385ms
rmdir	0.985ms	0.992ms	1.029ms
symlink	1.755ms	1.756ms	1.772ms

4　性能評価

基本性能の評価として，本機構を Linux カーネル 2.6.33.7 に実装し，Core i7 2.8GHz のプロセッサ，2GB のメモリを搭載した計算機 2 台を 1000Mbps のイーサネットで接続した環境で，以下のような条件でシステムコールを実行し処理時間を測定した．

A 本機構を使用せず，システムコールを発行した計算機で実行
B 本機構を使用し，システムコールを発行した計算機で実行
C 本機構を使用し，システムコールを他の計算機に転送し実行

発行するシステムコールは open，mkdir，rmdir，symlink とし，それぞれの処理時間を測定する．また，ファイルシステムには NFS を使用し，第 5 階層のディレクトリ内のファイルを操作の対象とした．処理に要した結果を表 1 に示す．A と B の処理時間の差は，システムコールを実行すべき計算機を調べる際に起きるオーバヘッドであると考えられる．また，システムコールを転送するのに要する時間は B と C の差である．open の転送処理時間の増加は，アドレス空間の送受信が伴ったためであると考えられる．

5　おわりに

本稿では，分散されたスレッドが発行したシステムコールを適切な計算機に転送し実行するシステムコールの制御について述べた．本機構により，任意の計算機からシステムコールを発行することが可能になり，位置透過なスレッドの分散実行が実現される．

参考文献

[1] 小鍛治 翔太，芝公仁，岡田至弘: 分散共有メモリを用いたアドレス空間共有方式，平成 22 年度情報処理学会関西支部大会講演論文集，A-02，2010.
[2] Lamport, L: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, IEEE Trans. Comput. , Vol. C-28, No. 9, pp.690-691 (1979).

プロセスを分散実行するための システムコール制御に関する検討

龍谷大学
三添 匠　小鍛治 翔太　芝 公 仁

はじめに

・計算機で処理する内容が複雑化
・複数のスレッドを使用して動作するアプリケーションが増加
　webサーバ，エンコード，データベースサーバ・・・・

一般的なシステムでは単一の計算機上でしか動作しない

・単一計算機だけでは処理能力に上限がある
・処理能力の向上には大きなコストが必要

プロセス共有機構の実現

目的
プロセスを複数の計算機で共有し，計算機資源を有効利用
・複数の計算機を利用可能
・単一のプロセスが複数の計算機にまたがって動作可能
・既存のアプリケーションの変更を必要としない

プロセス共有機構

プロセス共有機構・・・プロセスを複数の計算機で共有,動作

・プロセスの持つアドレス空間を計算機間で共有
・システムコールの転送
・分散されるスレッドの管理

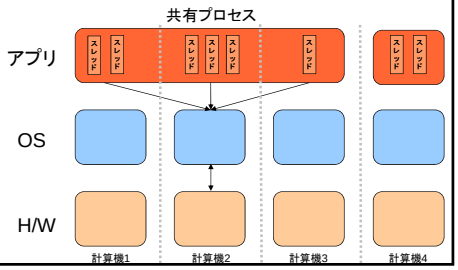
プロセスの共有が可能

Linux(x86アーキテクチャ)のカーネルモジュールとして実装

特徴
・アドレス空間を共有しているため複数の計算機でコードの共有が可能
・プロセスの持つスレッドを分散実行
・計算機にかかる負荷を分散し資源の有効利用

プロセス共有機構

- アプリケーションから見たプロセス共有機構
  - 共有するプロセスで生成されるスレッドは各計算機で処理
  - アプリからは一つのOSしか見えない（OSが共有される）



関連研究

- プロセスマイグレーション

・プロセスの移送
　- 計算機上で動作しているプロセスを他の計算機に移送
　- 動作中のプロセスを移送させる

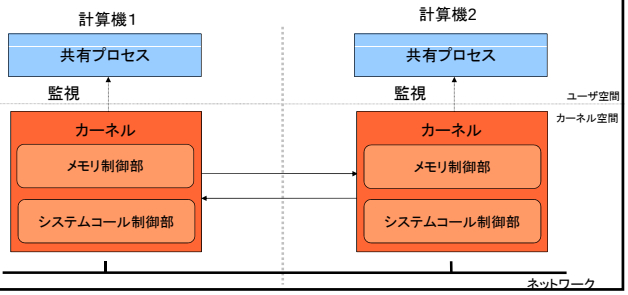
・システムコールの転送
　- プロセスの移送先で発生したシステムコールをもとの計算機に転送
　- 移送したプロセス一つに対応

プロセス共有機構

・単一のプロセスを複数の計算機で処理．プロセス自体の分散
・一つのOSを使用
・動作しているアプリケーションは一つの計算機で処理しているように見える

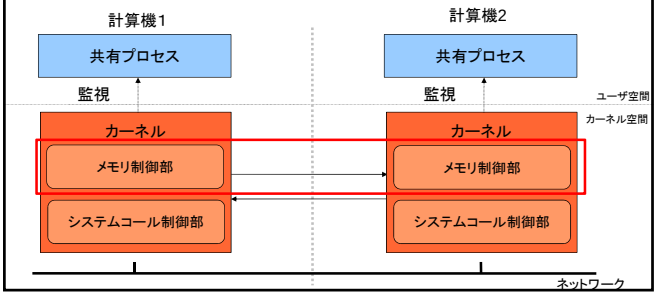
プロセス共有機構

- メモリ制御部
  - メモリ制御部で共有するプロセスを監視
  - アドレス空間に書き込んだ際に他の計算機からでも読み出せる
  - 順序一貫性のメモリモデルで管理



プロセス共有機構

- システムコール制御部
  - システムコールを適切な計算機に転送し，実行可能
  - スレッドを位置透過に分散可能



メモリ制御部

- メモリ制御部
  - 指定したプロセスもつアドレス空間の共有を実現
  - プロセスを監視しメモリの一貫性制御を行う

・メモリの一貫性制御
- 順序一貫性のメモリモデルによりメモリ内容を管理

▪ある計算機で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せる

▪同一計算機上でのスレッド間のメモリの共有と同じ

▪メモリはページ単位で管理を行う
-メモリの読み込みは複数の計算機で可能で，
メモリの書き込みは一つの計算機でのみ可能

メモリ制御部

- メモリの一貫性制御

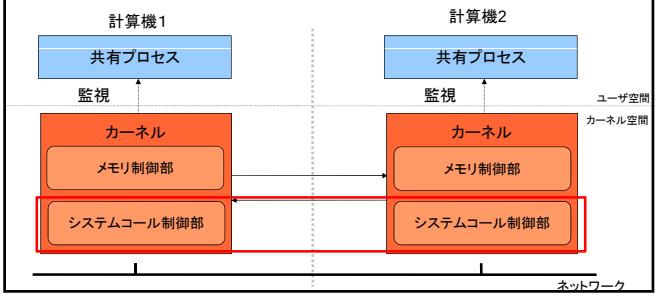
メモリの読み込み

・最新のページの内容を持っていないと読み込みは不可
・読み込みのアクセスをページフォルトの検出で取得
・最新のページの内容を持つ計算機からページの取得
・取得したページの状態を読み込み可能にして，スレッドの再開

メモリの書き込み

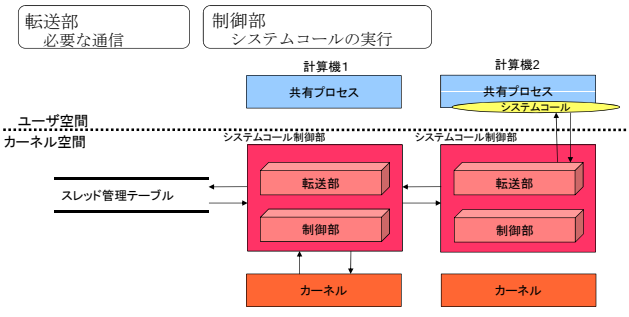
・書き込みアクセスをページフォルトの検出で取得
・他の計算機のページ状態を読み書き不可に変更
・ページの状態を書き込み可能にして，スレッドの再開

プロセス共有機構



システムコール制御部

・一つのOSで共有プロセスを実行させるためシステムコールの転送が必要
・システムコール制御部により共有プロセスから発行されるシステムコールを適切な計算機に転送し実行



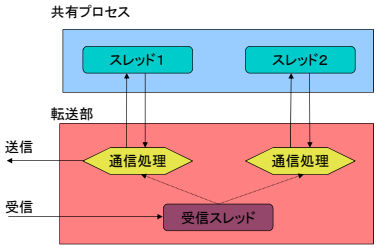
Linuxでのシステムコール

- システムコール（x86アーキテクチャ）
  - -ソフトウェア割り込みを利用して行われるカーネルに対する命令
  - アセンブリ言語で書かれたシステムコールハンドラを呼び出し処理する

▪処理の流れ
- ユーザープロセスがソフトウェア割り込みによりシステムコールを発行
- CPUをカーネルモードに移行し，すべてのレジスタの内容を退避する
- システムコールテーブルからシステムコールサービスルーチンを呼び出す
- 呼び出すサービスルーチンは，レジスタ内のシステムコール番号により判断
- システムコールに使用する引数はレジスタ渡し
- プロセスアドレス空間へのアクセスはcurrentマクロを使用
- 退避してあるレジスタ値を読み込む
- CPUをユーザモードに切り替え，ユーザプロセスを復帰

転送部

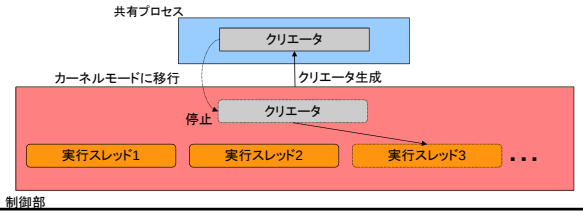
通信機能
- システムコール実行要求
  スレッド番号，システムコール番号，引数(レジスタ値)
- システムコール終了通知
  戻り値
- スレッド生成通知
  スレッド番号，アドレス
- スレッド終了通知
  スレッド番号，アドレス



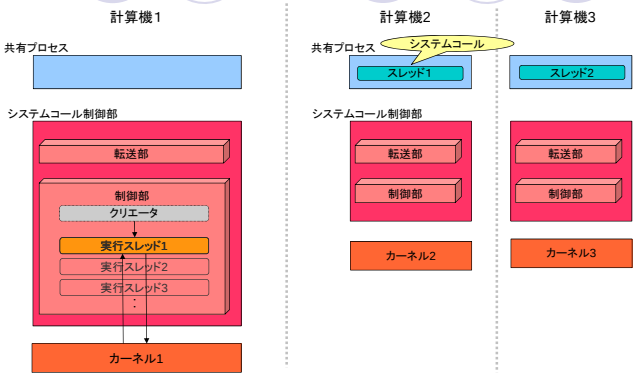
システムコール実行

実行スレッド
・共有プロセスでスレッドが生成されると制御部に実行スレッドを生成
・共有プロセスで生成されるスレッドに1対1に対応しシステムコールを実行
・制御部のクリエータにより実行スレッドを生成

クリエータ
・システムコールを実行する計算機で動作している共有プロセスにスレッドとして生成
・生成されるとカーネルモードに移行し待ち状態へ
・スレッドが生成されると実行スレッドを生成

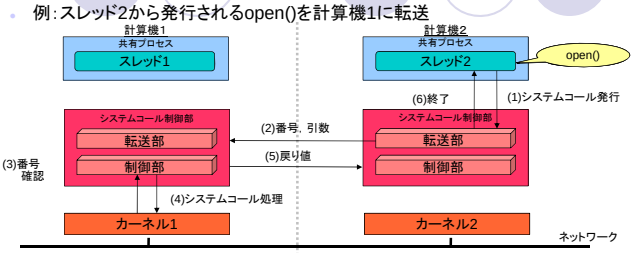


実行スレッドの生成



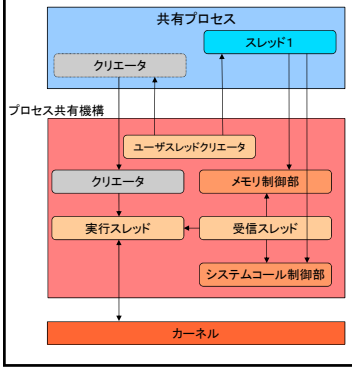
システムコール実行

- 例：スレッド2から発行されるopen()を計算機1に転送



(1)スレッド2がシステムコールを発行
(2)システムコール実行要求(スレッド番号，システムコール番号，引数)を計算機1に送信
　　open()　システムコール番号:5　引数(char *pathname, int flags)
(3)スレッド番号，システムコール番号の確認
(4)システムコールの処理(引数にアドレス空間の参照がある場合メモリ制御部により一貫性制御)
(5)システムコール終了通知(戻り値)を計算機2に送信
　　open()　戻り値(int)
(6)戻り値をレジスタに格納

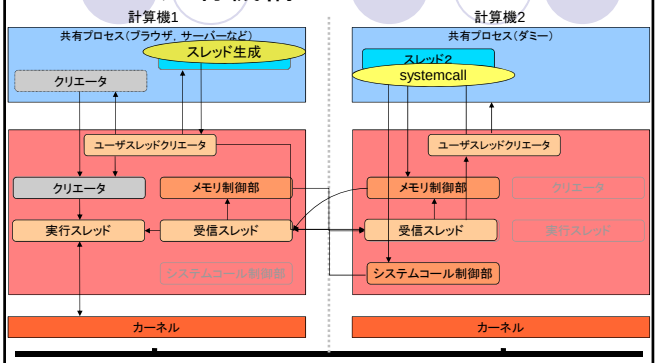
プロセス共有機構



- プロセス共有機構
  -Linux(x86アーキテクチャ)
  -カーネルモジュールとして実装

・メモリ制御部
　共有するプロセスのアドレス空間の制御
・システムコール制御部
　システムコールの転送
・ユーザスレッドクリエータ
　共有プロセス内にスレッドの生成
・実行スレッド
　システムコールの実行
・クリエータ
　実行スレッドの生成
・受信スレッド
　データの受信

プロセス共有機構



スレッド管理

- 分散するスレッドの管理
  - 分散されるスレッドの管理は，個々の計算機での管理だけでは足りない
  - 共有プロセスで生成されるスレッドは，システムコールを実行する計算機のスレッド管理テーブルで管理する

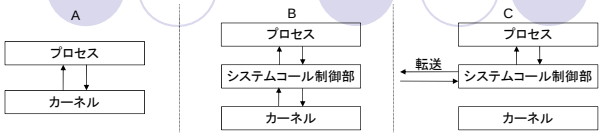
・スレッド管理テーブル
- システムコールを実行する計算機で管理
- 共有プロセスでスレッドが生成されたことを登録
- プロセス共有機構で管理するためのスレッド番号を割り当てる
- スレッドが動作している計算機のアドレスを登録
- 割り当てられているスレッド数が少ない計算機に分散

性能評価

<u>性能評価</u>
システムコールの転送の測定

<u>評価環境</u>
Linuxカーネル2.6.33.7にシステムコール制御部を実装
プロセッサ：Intel Core i7 2.8GHz
メモリ　：2GB
1000Mbpsのイーサネットで接続

性能評価



A:本機構を使用せず，システムコールを発行した計算機で実行
B:本機構を使用し，システムコールを発行した計算機で実行
C:本機構を使用し，システムコールを他の計算機に転送し実行

open以外　転送時間＝C-B　約0.020ms

システムコール	A	B	C
open	0.275ms	0.004 0.279ms	4.025 4.304ms
mkdir	1.355ms	0.010 1.365ms	0.020 1.385ms
rmdir	0.985ms	0.007 0.992ms	0.037 1.029ms
symlink	1.755ms	0.001 1.756ms	0.016 1.772ms

おわりに

単一のプロセスを複数の計算機にまたがって動作させる手法

- プロセス共有機構
  - メモリ制御部
    - メモリの一貫性制御を行う
  - システムコール制御部
    - 共有プロセスの発行するシステムコールを実行すべき計算機に転送し，処理する

位置透過にプロセス内のスレッドを分散実行可能

・処理速度の向上
・計算機資源の有効利用

分散センサデータ管理における性能向上のための クラスタを利用したプロアクティブデータの予測・配置手法

井邊 研吾 †† 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，我々はマイクロストレージを用いたセンサデータの分散管理システムである P2P データポット [1] の開発を進めている．P2P データポットは，観測対象の区域を複数の範囲に分割し，複数のセンサネットワークによる観測を行うことで，各センサノードの通信回数を削減し，長寿命化を図る．また，P2P データポットでは，ネットワーク構築や複数の拠点の管理を容易にするため，無線アドホックネットワークによる各拠点間の動的なネットワーク構成と，複数のセンサネットワークへの問合せを実現する統一的な問合せインターフェースを用いている．しかし，アドホックネットワークにおけるマルチホップ通信環境では，ホップ数の増加により，問合せの結果の取得に要する時間が増大し，応答性の低下が問題となる．

このため，我々はこれまでに P2P データポットにおけるセンサデータのプロアクティブデータ転送・検索機構の提案を行ってきた．本機構では，これまでに発行された問合せを基に，今後発行される問合せの予測を行い，事前に対応する問合せ結果の転送を行う．これにより，問合せ結果の転送に要する時間が短縮され，応答性の向上が可能となる．本研究では，この機構の予測精度を向上させるため，周辺の P2P データポットでクラスタを形成する手法を提案する．

2 P2P データポットによるセンサデータ管理

図 1 に P2P データポットを用いたセンサネットワークシステムの構成図を示す．本システムは，環境観測を行うセンサネットワーク，センサネットワークで得られたセンシングデータを蓄積する P2P データポット，P2P データポットネットワークに対してセンシングデータの問合せを行うアプリケーションで構成される．センサネットワークと P2P データポットは 1 対 1 に対応し，センサネットワークで取得したデータは，対応する P2P データポットに蓄積される．また，P2P データポット間は無線アドホックネットワークで接続され，アプリケーションから発行された問合せは，対応する P2P データポットに転送される．

任意の P2P データポットが，アプリケーションから問合せを受けることができる．問合せを受けた P2P データポットは対象となる各々の P2P データポットに対して

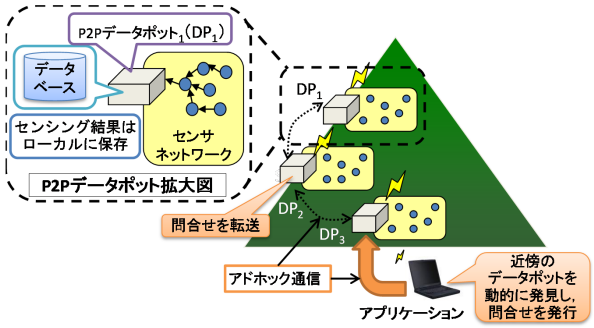


図 1 P2P データポットのシステム構成

問合せを行い，結果の集約を行う．この際，一連の通信処理は，無線アドホックネットワーク上で行われる．このため，遠方の P2P データポットに対する問合せが発生した場合，問合せや問合せ結果の転送回数が増加し，これによってアプリケーションが問合せ結果を得るまでの時間が増大し，応答性が低下することが問題となる．

3 プロアクティブデータ転送・検索機構

P2P データポットは，問合せの転送回数の増大による応答性の低下を改善するため，プロアクティブデータ転送・検索機構を持つ．本機構は，これまでに発行された問合せを基に，今後発行される問合せの予測を行い，事前に対応する結果の転送を行う．アプリケーションからの問合せが事前に転送したデータで対応可能な場合には，転送したデータを用いて応答することにより，転送回数が減少し応答性の向上が可能となる．予測された問合せに対応する結果となるデータをプロアクティブデータと呼ぶ．

本機構は，プロアクティブデータ転送機構と検索機構から構成される．プロアクティブデータ転送機構は，問合せの予測に必要となる問合せ履歴の保存，今後発行される問合せの予測，予測したデータを事前に転送するプロアクティブデータ転送の 3 つの機能から構成される．一方，プロアクティブデータ検索機構は，プロアクティブデータを検索対象に加え，アプリケーションから受けた問合せに応答する．プロアクティブデータの検索は，プロアクティブデータの配置情報をメタデータとして全 P2P データポットと共有することで実現する．

これまで，プロアクティブデータの転送・検索メカニズムについて研究を進めてきた．しかし，問合せの予測

方法とプロアクティブデータの配置方法が，検討されれておらず，効果を示すことができなかった．そこで，本稿では，性能向上のためのクラスタを利用した，プロアクティブデータの予測と配置手法について述べる．

4 クラスタを利用したプロアクティブデータの予測と配置手法

これまでの予測方法では，単体の P2P データポットに蓄積された履歴を用いて，繰り返し行われる問合せを検知し，その問合せが今後も発行されるものとしてきた．しかし，単体の P2P データポットの履歴では，アプリケーションからの問合せ先の P2P データポットが異なった場合や，要求されたセンサデータを持つ P2P データポットが異なった場合には，問合せ内容が同じ場合でも，異なる問合せとして認識されることになる．このため，繰り返し行われる問合せとして処理されず，今後も発行される問合せとして予測することができなかった．

例えば，登山者が天候を確認するようなアプリケーションでは，複数の登山者が個別に移動しながら，問合せを繰り返し発行すると考えられる．登山者 A は，山道の入口付近および入口付近から 3km ほど進んだ地点それぞれにおいて，現在地から移動方向に沿って 3km 先の地点の周辺気温情報を取得する問合せを発行するものとする．他の登山者も登山者 A と同一のアプリケーションを用いて行動する．この場合，単体の P2P データポットの履歴だけでは，全ての登山者が同じ P2P データポットに問合せを行い，同じ P2P データポットにあるセンサデータの取得を行わない限り，繰り返し発行される問合せを発見することができないことになる．そこで，ある程度問合せ発行位置が異なった場合でも予測結果を適用できるようにするため，必ずしも同一のデータポットでない場合でも，一定の範囲内に存在するものは同一視して扱うものとする．このような予測を実現するため，近隣の複数の P2P データポットをまとめてクラスタとして扱い，全ての P2P データポットのクエリ履歴を共有する方法を提案する．

5 プロアクティブデータの予測

図 2 にクラスタを用いた繰り返し行われる問合せの検知を示す．アプリケーション AP1 は時刻 $t = t_1$ においてデータポット DP1 に対し DP4 の気温データを要求する問合せ Q1(DP4) を発行する．その後 AP1 は移動し，時刻 $t = t_1 + k$ において DP7 の気温データを要求する問合せ Q1(DP7) を発行する．このとき，アプリケーションにデータを提供する P2P データポットの遷移履歴として，気温データを要求する問合せ Q1 がクラスタ β から γ に遷移したというクラスタ単位での情報が記録される．アプリケーション AP1 と同一種類のアプリケーショ

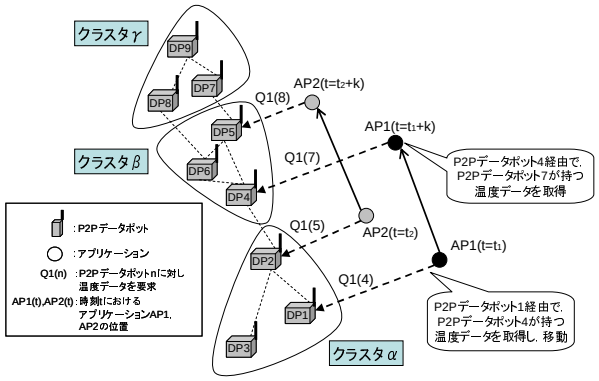


図 2　クラスタを用いた繰り返し行われる問合せの検知

ン AP2 も同様に，移動しながら $t = t_2$ および $t = t_2 + k$ において同じ内容の問合せ Q1(DP5) と Q1(DP8) を発行する．この場合も同様に，Q1 がクラスタ β から γ に遷移して発行されたという情報が記録される．このような問合せが繰り返された場合，今後もクラスタ α 内の P2P データポット経由でクラスタ β 内の P2P データポットから問合せ Q1 が発行され，時間 k 経過後にクラスタ β 内の P2P データポット経由でクラスタ γ 内の P2P データポットから問合せ Q1 が発行されると予測することが可能となる．

6 プロアクティブデータの配置

どのクラスタからどのクラスタのデータが要求されるかという，クラスタ単位での問合せの予測に対応し，プロアクティブデータの配置もクラスタ単位で行う．各 P2P データポットは全ての P2P データポットの問合せ履歴を共有することで，クラスタ単位で，繰り返し行われる問合せの検知が可能となる．図 3 に配置例を示す．図 3 では，AP1 が DP1 にアクセスし，DP5 の温度データを利用する．また，AP2 が DP3 にアクセスし，DP6 の温度データを利用する．この場合，アプリケーションがクラスタ α にアクセスし，クラスタ β の温度データを取得する問合せが発行されると予測される．このとき，クラスタ β に所属する DP4，5，6 の温度データはクラスタ α に所属する DP1，2，3 にコピーされる．これにより，アプリケーションがアクセスするデータポットが一定の範囲内で変化する場合や，データを要求するデータポットが一定の範囲内で変化する場合にも，プロアクティブデータを用いることで応答性の向上が見込める．

7 評価

本手法の評価としてプロアクティブデータ転送・検索機構を P2P データポットに搭載した場合のシステム評価と，予測・配置手法の妥当性評価を，シミュレータを用いて行う．システム評価は，Python によるシュミレーションパッケージである simpy[2] を用いて処理待ち時

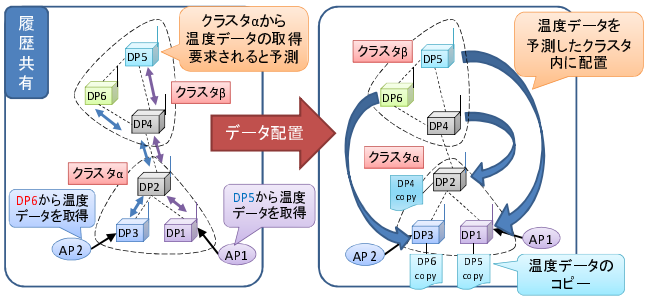


図 3　クラスタを用いたプロアクティブデータの配置

間，転送待ち時間，一連の問合せ処理の時間を計測し，本機構が与える影響について調査する．主な評価項目としては，既存の P2P データポットシステムと本機構を導入した P2P データポットシステムの，問合せ処理時間の比較が挙げられる．また，本機構の影響により通常の問合せ処理の時間が既存のシステムより増大するなどの性能の低下が考えられる．そこで，このシミュレーション結果を利用し，性能が低下しないデータの処理量の検討を行う．

一方，予測・配置手法の妥当性の評価は，クエリ履歴から予測を行いプロアクティブデータを配置することで，問合せ処理に要するホップ数の減少を評価するためのシミュレータを開発し実施する．このシミュレータは，擬似的にクエリ履歴を作成し，そのクエリ履歴を基に予測・配置手法を適用する．その結果，繰り返し行われる問合せの抽出と問合せ処理に要するホップ数の削減率を既存のシステムと比較し，評価を行う．また，予測・配置手法の適用時にかかるシステムの負荷の限度を，前述のシステム評価をもとに検討することで，システム負荷に関する考察を行う．

8　おわりに

本稿では，分散センサデータ管理における性能向上のためのクラスタを利用したプロアクティブデータ予測・配置手法について述べた．今後は，シミュレータの作成を進め，システム評価と提案手法の評価を行う予定である．

参考文献

[1] 藤崎友樹, 鈴来和久, 横田裕介, 大久保英嗣: “無線アドホック通信を利用したセンサネットワーク向け協調ストレージシステム”, 情報処理学会研究報告, 2008–MBL–44/2008–UBI–17, pp. 149–156, 2008 年 3 月.

[2] SimPy Simulation Package Homepage. http://simpy.sourceforge.net/

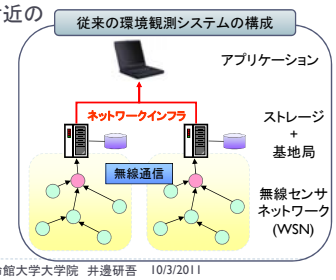
分散センサデータ管理における性能向上のためのクラスタを利用したプロアクティブデータ転送・検索手法

立命館大学大学院 理工学研究科　大久保・横田研究室
井邊研吾

目次

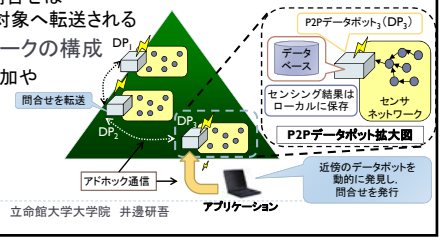
はじめに

- 従来の環境観測システム
  - 無線通信機能を持つ複数のノードから構成されるセンサネットワーク
  - センシングしたデータを基地局のストレージへ蓄積
  - 複数拠点を置き、基地局付近のノードの負荷軽減
- 複数拠点の課題点
  - ユーザは場所を意識して利用しなければならいない
  - ネットワークインフラの設定
- 改善方法
  - P2Pネットワークの利用
  - アドホック通信の利用



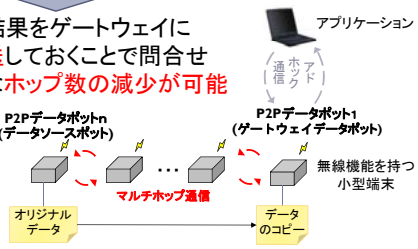
P2Pデータポットシステム

- 無線アドホック通信を利用した環境観測システム
  - 従来の基地局・ストレージを無線通信機能を持つ小型の計算機端末（P2Pデータポット）に置き換え
    インフラレスな観測システムを構成
  - どのデータポットからも同じ結果を得ることが可能
    - アプリからの問合せは
      内容に応じて対象へ転送される
  - 動的なネットワークの構成
    - 観測地域の追加や
      退去が容易



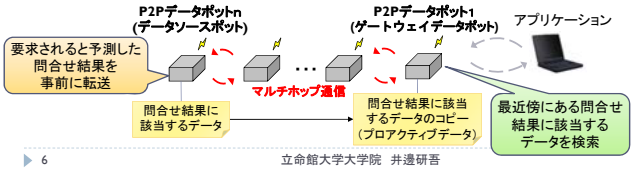
P2Pデータポットシステムにおける課題

- マルチホップ通信による応答性の低下
  - 遠方の端末に問合せを発行した場合
    問合せの転送に必要なホップ数が増加
- 事前に問合せ結果をゲートウェイに近い端末に転送しておくことで問合せ応答時に必要なホップ数の減少が可能



プロアクティブデータ転送・検索機構

- 目的：問合せ結果の取得時間を短縮し、応答性の向上
- プロアクティブデータ転送機構
  - どのP2Pデータポットがどのデータを要求するかを予測
  - 予測したデータを事前に当該P2Pデータポットの近隣に配置
- プロアクティブデータ検索機構
  - 問合せに対し、事前に転送したデータも検索対象
  - オリジナルと転送したデータの最近傍のデータを取得

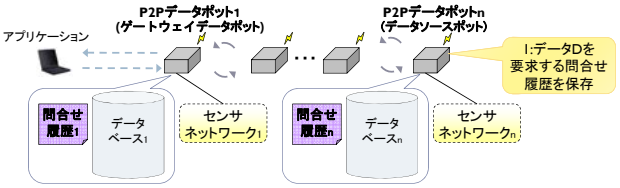


プロアクティブデータ転送機構の機能概要

1. 問合せの履歴保存
   - 問合せ履歴を予測に利用
2. 問合せの予測
   - 問合せ履歴から今後発行される問合せを予測
     - トラヒックの増加とストレージの消費を軽減
   - 予測した問合せ結果＝プロアクティブデータ
3. 予測した問合せ結果の転送・配置
   - プロアクティブデータを当該のP2Pデータポットに転送
4. プロアクティブデータの削除
   - 利用されないプロアクティブデータの削除

プロアクティブデータ転送機構の機能(1/4)

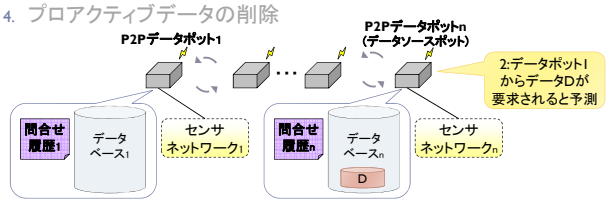
1. **問合せの履歴保存**
   - 受けた問合せ情報を保存
2. 問合せの予測
3. 予測したデータの転送・配置
4. プロアクティブデータの削除



プロアクティブデータ転送機構の機能(2/4)

1. 問合せの履歴保存
2. **問合せの予測**
   - 問合せ履歴を用いて今後発行される問合せの予測
3. 予測したデータの転送・配置
4. プロアクティブデータの削除

予測内容の例	
転送先データポット（プロアクティブデータポット）	DPI
問合せ内容	SELECT * FROM mts300

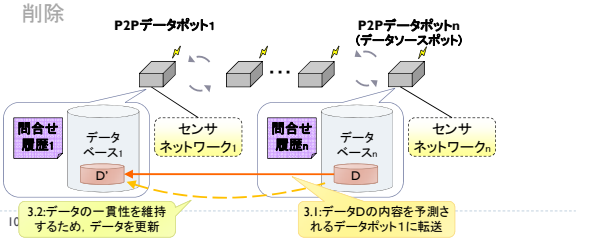


プロアクティブデータ転送機構の機能(3/4)

1. 問合せの履歴保存
2. 問合せの予測
3. **予測したデータの転送・配置**
4. プロアクティブデータの削除

3.1.データ転送
- 予測した問合せ結果をプロアクティブデータポットに転送

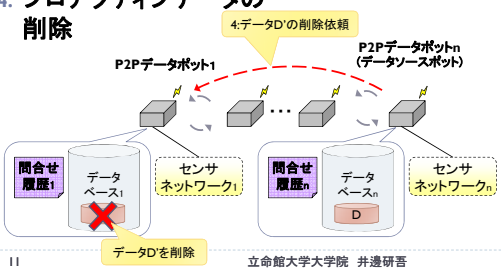
3.2.追加転送
- 問合せ結果に該当するデータを逐次転送
- 時間の経過により問合せ結果が異なる場合



プロアクティブデータ転送機構の機能(4/4)

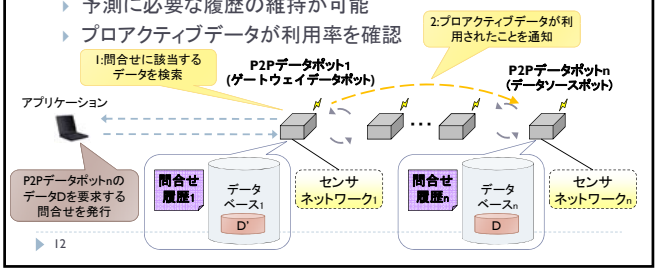
1. 問合せの履歴保存
2. 問合せの予測
3. 予測したデータの転送・配
4. **プロアクティブデータの削除**

- 利用されていないものを削除
  - 予測が失敗していた場合
  - 時間経過により利用されなくなる場合



プロアクティブデータ検索機構の機能

1. プロアクティブデータの検索
   - 問合せに応答できる最近傍のデータを検索
     - オリジナルデータとプロアクティブデータの両方を検索
2. プロアクティブデータの利用通知
   - 予測に必要な履歴の維持が可能
   - プロアクティブデータが利用率を確認



転送・検索機構における提案手法の位置づけ

- 予測手法:発行される問合せの予測方法
- 配置手法:予測した問合せ結果の配置方法
- 置換手法:プロアクティブデータの置換方法

プロアクティブデータ転送・検索機構

• 問合せの履歴保存 • 問合せの予測 ➢ **予測手法** • 予測したデータの転送・配置 ➢ **配置手法** ➢ **置換手法** • プロアクティブデータの削除	• プロアクティブデータの検索 • プロアクティブデータの利用通知

プロアクティブデータの予測

- アプリケーションからの問合せを予測
  - 利用されたデータから次に利用されるデータを予測することは困難
  - 問合せ結果の動的変化に対応可能
    - 時刻に依存して問合せ結果が変化する

P2PデータポットAが所持するデータ

ノードID	温度℃	日時
1	16.5	2010/11/30/10:30:15
2	15.7	2010/11/30/10:30:30
3	9.8	2010/11/30/10:30:45
1	16.4	2010/11/30/10:45:30
3	9.5	2010/11/30/10:45:50

15分経過後

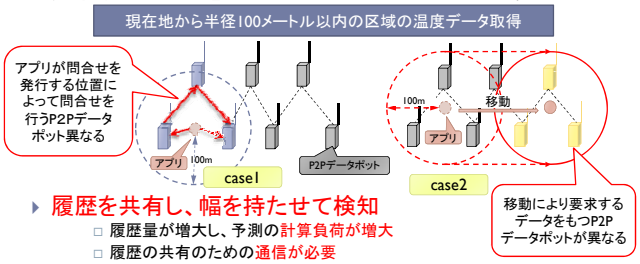
問合せ「**最近の3件のデータ**」の結果が変化

P2PデータポットAが所持するデータ

ノードID	温度℃	日時
1	16.5	2010/11/30/10:30:15
2	15.7	2010/11/30/10:30:30
3	9.8	2010/11/30/10:30:45
1	16.4	2010/11/30/10:45:30
3	9.5	2010/11/30/10:45:50
2	15.5	2010/11/30/10:50:35
3	9.1	2010/11/30/11:00:50

プロアクティブデータの予測手法概要(1/2)

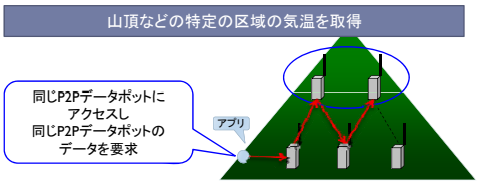
- アプリが問合せを発行するP2Pデータポットが異なる場合(case1)
- 要求するデータをもつP2Pデータポットが異なる場合(case2)



- 履歴を共有し、幅を持たせて検知
  - 履歴量が増大し、予測の計算負荷が増大
  - 履歴の共有のための通信が必要

プロアクティブデータの予測手法概要(2/2)

- アプリが同じP2Pデータポット経由でかつ同じデータを取得する場合(case3)



- 単体のP2Pデータポットの履歴で検知可能
  - 計算量が少ない
  - 履歴の共有のための通信が不要

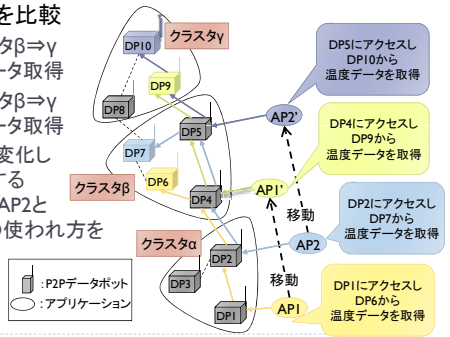
複数の履歴からの予測(case1,2)(1/2)

- 同じようなデータの使われ方を検知
  - 単体の履歴では
    - 問合せ先が変わると予測不可能
    - 利用するP2Pデータポットが変わると予測不可能
  - 完全に同じデータの使われ方でなくても検知が必要
    - 近隣の複数のデータポットでクラスタを形成
    - 履歴を共有し、クラスタ単位でのデータの使われ方を検知

複数の履歴からの予測(case1,2)(2/2)

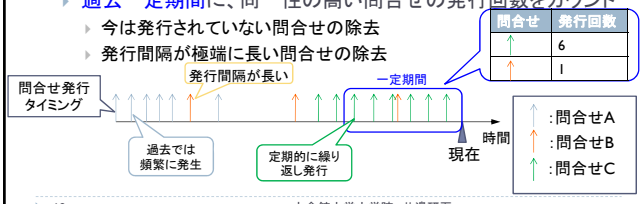
- 同じデータの使われ方をするアプリケーションのクラスタ移動を比較
  - API1はクラスタβ⇒γに変化しデータ取得
  - AP2はクラスタβ⇒γに変化しデータ取得

今後β⇒γに変化しデータ取得するようなAPI1やAP2と同じデータの使われ方をすると予測



単体の履歴からの予測(case3)

- 同じデータソースポットに対し
  繰り返し行われる問合せの検知
  - 繰り返し行われる問合せは、今後発行される可能性が高い
  - 単体のデータソースポットの履歴から検知可能
- 繰り返し発見方法
  - 過去一定期間に、同一性の高い問合せの発行回数をカウント
    - 今は発行されていない問合せの除去
    - 発行間隔が極端に長い問合せの除去



転送・検索機構における提案手法の位置づけ(再)

- 予測手法:発行される問合せの予測方法
- 配置手法:予測した問合せ結果の配置方法
- 置換手法:プロアクティブデータの置換方法

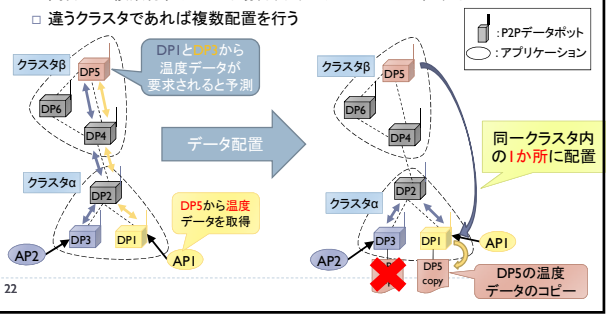
プロアクティブデータ転送・検索機構	
• 問合せの履歴保存 • 問合せの予測 ➢ **予測手法** • 予測したデータの転送・配置 ➢ **配置手法** ➢ **置換手法** • プロアクティブデータの削除	• プロアクティブデータの検索 • プロアクティブデータの利用通知

配置手法と置換手法の概要

- プロアクティブデータの配置手法
  - 単体のP2Pデータポットの履歴から検知
    - アプリケーションから問合せを受ける特定のP2Pデータポットへ転送
  - 複数のP2Pデータの履歴から検知
    - アプリケーションから問合せを受ける
      クラスタに所属するP2Pデータポットへ転送
- プロアクティブデータの置換手法
  - 配置するデータの選択が必要
    - ストレージ容量が有限
  - 選択する指標の決定
    - データの利用率(平均問合せ間隔)
    - データサイズ
    - オリジナルデータと転送先の距離

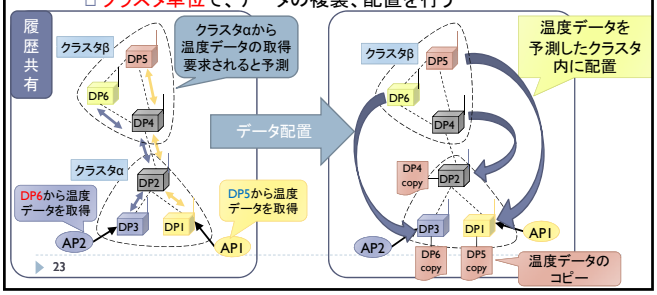
配置手法(1/2)

- 単体のP2Pデータポットの履歴から検知
  - アプリケーションから問合せを受けるP2Pデータポットに転送
    - 問合せが複数行われている場合、クラスタに1つだけ配置する
    - 違うクラスタであれば複数配置を行う



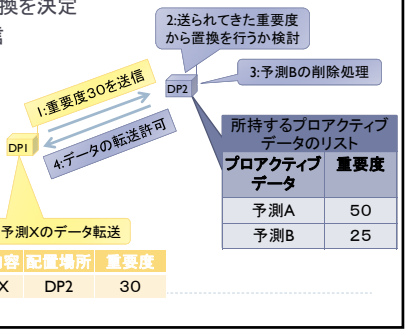
配置手法(2/2)

- 複数のP2Pデータポットの履歴から検知
  - アプリケーションから問合せを受ける
    クラスタに所属するP2PDPへ転送
    - クラスタ単位で、データの複製、配置を行う



置換手法

- 置換手法
  1. プロアクティブデータ転送前に、重要度を送信
  2. 重要度をもとに置換を決定
  3. 予測データの送信
- 重要度
  - データの利用率
    (平均問合せ間隔)
  - データサイズ
  - オリジナルデータと
    転送先の距離



評価方法

- システム性能評価
  - プロアクティブデータ転送・検索機構がシステムに与える影響を評価
    - 処理待ち時間や転送待ち時間
    - 一連の問合せ処理時間
  - simpyを利用したシミュレータを作成
    - simpy:プロセスベースの離散イベントシミュレーション言語
- 予測・配置手法の妥当性の評価
  - 提案手法によって、繰り返し行われる問合せの検知とその問合せに要するホップ数の減少率を評価
  - テスト用の問合せ履歴を作成し、提案手法を適応できるシミュレータを作成し評価する

おわりに

- 発表内容
  - 従来の環境観測システムの問題点
  - P2Pデータポット
    - 無線アドホック通信を利用した環境観測システム
  - プロアクティブデータ転送・検索機構
  - プロアクティブデータ予測・配置手法
    - クラスタを利用し、繰り返し行われる問合せの検知率を向上
- 今後の予定
  - システム評価用のシミュレータの作成と評価
  - 手法の妥当性を評価するシミュレータの作成と評価

天気予報に基づくソーラパネルを用いたセンサネットワークの永続的運用手法

大橋 一輝 ††　横田 裕介 †　大久保 英嗣 †
† 立命館大学情報理工学部　†† 立命館大学院理工学研究科

1　はじめに

センサノードと呼ばれる, センサと無線通信機能を持った小型計算機を用いることで温度などの環境データを取得することが可能である. このセンサノードを複数用いることで, 通信インフラのない場所において, 低コストで環境観測を実現するセンサネットワークを構築することができる. 一般的なセンサノードはバッテリで駆動しており, 電力資源に制約がある. そのため, センサネットワークの省電力化, 長寿命化のための研究が行われている. しかし, どれほど省電力化を行ったとしても, 消費する一方ではいずれ電力は枯渇してしまう. そのため, センサノードの電力をソーラパネル等を用いて回復させ, 電力資源を有効に利用する研究が行われている [4][5]. 本研究でも同様に, ソーラパネルを用い, ノードの電力消費の制御を行うことで, ユーザの要望に最大限こたえながら, ノードの永続的動作を実現することを目指す. 具体的には, ユーザが希望するサンプリングレートにできるだけ近い動作をさせることが目標である.

以下, 2 章で提案手法について述べる. そして, 3 章で初期的な評価について述べる. 最後に, 4 章で本稿をまとめる.

2　提案手法

提案手法のシステム構成を図 1 に示す. 基地局側では, 天気予報の取得や, 送られてくるセンサデータの処理を行う. また, ノード側では, 基地局から送られる天気予報を受信し, その情報をもとに, 電力消費スケジュールを決定する. そして, 電力消費スケジュールに応じて, センサデータを取得し, 基地局へ送信する. 提案手法は, 主にソーラパネルからの充電量の予測とサンプリングレートの調整からなる. 天気予報やセンサデータを用いて, 充電量を予測することでどの程度のエネルギーが利用できるかを予測することができる. また, 予測に応じてサンプリングレートを変更することで電力資源を効率的に利用することができる.

2.1　プログラム動作スケジュール

図 2 に各プログラムの機能が動作するタイミングを示す. 縦軸には, 動作内容, 動作タイミングに合わせて印がつけてある. 天気予報は同じものを何度も送る必要性がないため, 天気予報の更新間隔を考慮し, 6 時間毎に天気予報の取得, 送受信を行う. また, センサデータの取得, 送受信については, 後に記述する電力消費スケジュールによって決定された周期毎に行う. さらに, 電力消費スケジュールの決定については, 電力回復が行える時間帯とそうでない時間帯で動作間隔を変更する. これは, 天気の急な変化に対応するためである. そのため, 日の出から日没までは 30 分に 1 度電力消費スケジュールを決定する. それ以外の場合は, 天気予報の取得に合わせた 6 時間毎に行う.

2.2　電力利用スケジュールの決定

電力利用スケジュールの決定は 2 つの段階で成り立っている. 最初の段階は, 電力回復量の予測である. 1 週間先までの各日毎の電力回復量を 2 種類の方法によって予測する. 次の段階は, 日毎の電力利用量の調整である. ユーザの希望通りのまま動作をさせた際に, 電力が枯渇すると予想される場合, 電力を枯渇させないように調整する必要がある. 本手法では, 単純にサンプリング回数を減らすことで省電力化をはかる. すなわち, 決定された日毎の電力利用可能量より, サンプリング周期を決定する.

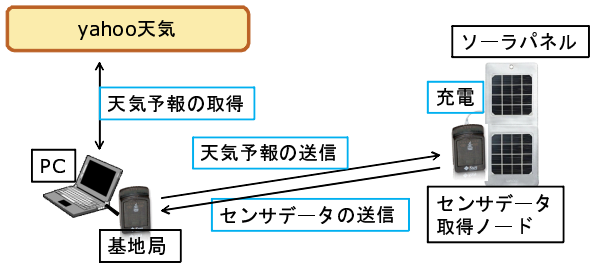


図 1　システム構成

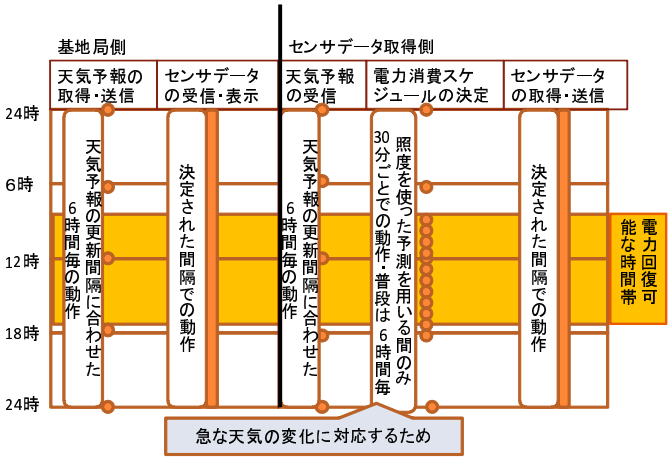


図 2　プログラム動作スケジュール

2.2.1　電力回復量の予測

電力回復量の予測は, 電力利用スケジュールの決定と同じタイミングで行われる. すなわち, 日の出～日没までの間では, 30 分に一度行われ, それ以外では 6 時間毎に行われる. 電力回復量の予測方法は期間によって 2 種類に分かれている. 当日の電力回復量の予測方法は短期予測である. 当日の電力回復量はセンサデータの照度の値から決定される. これは照度の値と電力発電量に非常に高い相関があり, 正確に電力回復量を予測できるためである. 現在の照度が日没まで続くとして, 電力回復量を予測する. また, 翌日以降の電力回復量は長期予測である. 長期予測での電力回復量の予測は, 天気予報に基づいて決定される. 天気毎に予め調査しておいた電力回復量の目安を使うことで, 予測が難しい将来の電力回復量についても目安をつけることができる.

2.2.2　電力利用量の調整

日毎の電力利用量の決定では, 基本的にユーザの希望した周期に従う. また, 長期間の環境観測では, ユーザの希望する周期が頻繁に変更されることは少ないと考えられるため, 前提として,

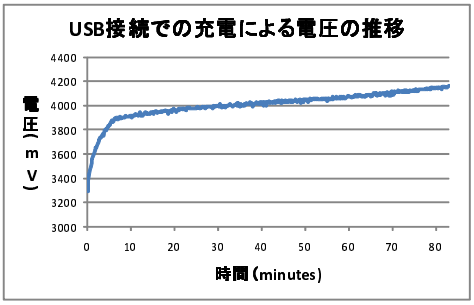


図 3　時間経過による電圧の推移

ユーザの希望する周期は一定とする. しかし, ユーザの希望した周期でサンプリングを行うと, 電力が枯渇すると予測される場合がある. そのため, 2 つの例外時のみ特別な処理をすることとする. まず 1 つ目は, 残電比率の推移予測が 100% を超えてしまう場合である. ここでいう残電比率とは満充電の際の動作寿命を 100% とする, センサノードにどの程度動作のための電力が残っているかの割合である. 残電比率は現在の電圧を調べることで把握することができる. 例えば,(100%,4150mV) , (50%,3885mV) , (0%,3250mV) といった具合である. この場合は, バッテリの許容量以上に電力をためておくことはできないので, 推移予測における次の日の残電比率を 100% 以上にならないように変更する. 2 つ目は, 残電比率の推移の予測が 50% を切る場合である. 事前実験より, 残電比率が 15% 以下の状態では, 電圧が変動しやすいという特徴がある. また, 図 3 より, 電力が 50% 以下の際には電力回復が速くなるという特徴がある.

そのため, より安定した動作を実現し, かつ効率よく電力を回復するために, 残電比率ができる限り 50% から 15% の間になるように電力利用スケジュールを立てる. その際の調整方法を以下に示す.

1. ユーザの希望周期による電力消費量のままで残電比率の推移を計算する.
   これにより, ユーザの希望通りに動作させた場合の残電比率の推移を予測できる.
2. N 日後の残電比率が 50% を切ると予想される場合に, N 日後まで各日の動作周期を減少させる.
   この動作により, 残電比率が少なくなりすぎるのを防ぐ. 変更する電力の利用量は,N 日での残電比率から 15% を引き (N+1) で割ったものにする.
3. 上記を最後まで繰り返す.

この調整方法を利用することで, 電力の枯渇を防止し, 各日の電力利用可能量を均等にするという電力利用ポリシーを満たすことができる.

3　評価

提案手法の初期段階の評価として, シミュレーションを行った. 今回のシミュレーションは事前実験で調べた単位時間毎の消費電力を利用している. 実際の手法との差異として, 照度と電力回復量の関連付けができていないため, 短期予測を用いて電力回復量の予測を行う部分の実装ができていない. そのため, 長期予測を当日の電力回復量の予測に利用し, 天気予報ベースで当日の電力の回復量予測を行っている. 用いたパラメータとして, 2011 年 4 月と 5 月に実際に予報された天気予報を用いた. また,

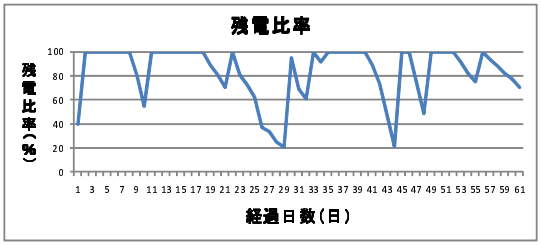


図 4　残電比率の推移のシミュレート結果

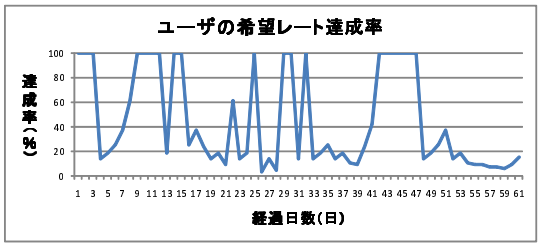


図 5　サンプリングレート達成率のシミュレート結果

期間中の実際の天気は気象庁の過去の天気のデータから判断した. 動作プログラムはユーザが 20 秒周期でのサンプリングを希望し, 照度, 温度, ソーラパネルからの発電量の値を送信する. シミュレーション結果を図 4, および図 5 に示す.

61 日間という環境観測としては短い期間であるが, 電力が枯渇することなく動作できていることが分かる. 特に 22 日〜28 日の期間において, 電力回復できる日があると予測されているが, 同期間の電力回復量は 0 である. このように天気予報が外れ, 電力回復量が予測よりも少なくなった場合にも電力が枯渇しないことが分かった.

4　おわりに

本稿では, 天気予報を用い, 効率的にユーザの希望する動作に応える手法について述べた. 今後は, 短期予測を実装し, シミュレーションの後, 実環境での動作実験を行う予定である. 現在の手法の問題点としては, 長期予測において, 同じ”晴れ”という天気であれば季節が関係なく一定の回復量としている点がある. 実際には日が出ている時間が違うため, 回復量は大幅に変わると思われる. そのため, 日照時間と天気の相関を求めることで, この問題を解決しようと考えている.

参考文献

[1] 畑田光毅, 永続的運用のためのソーラパネルを用いたセンサネットワークシステム, 立命館大学情報理工学部卒業論文, 2010.

[2] 中川重康, 天気概況からその日の晴天指数を概算する手法, 太陽エネルギー, vol.22, 1996, pp. 33–38.

[3] 田ノ岡正紀, 森山正和, 竹林英樹, 天気予報を利用した太陽光発電システムの効率的利用方法に関する研究, 神戸大学都市安全研究センター, 第 9 号, Japan, March 2005.

[4] Yong Yang, Lu Su, Yan Gao, Tarek F. Abdelzaher, Towards Reliable Data Delivery in Solar-powered Wireless Sensor Networks, Mobi-Com 2009 Poster , 2009.

[5] Roy Chaoming Hsu, Cheng-Ting Liu, Wei-Ming Lee, Reinforcement Learning-Based Dynamic Power Management for Energy Harvesting Wireless Sensor Network, IEA/AIE ’09 Proceedings of the 22nd International Conference on Industrial, pp399 - 408 , 2009.

天気予報に基づく ソーラパネルを用いた センサネットワークの永続的運用手法

立命館大学 大学院 M2
大橋 一輝

1

目次

- はじめに
- 天気予報に基づくソーラパネルを用いた センサネットワークの永続的運用手法
  - 電力回復量予測
  - 電力利用スケジュールの決定
- 評価
- おわりに

2

はじめに

- センサネットワークを用いた環境観測が期待されている
  - インフラのない場所においても安価で観測が可能
- センサノードの永続的運用には電力回復が必要
  - 省電力だけではいずれ電池がきれる
- 環境エネルギー(ソーラパネル)からの充電
  - 天候に左右されやすく不安定

電力回復量を予測することで
効率的にエネルギーを利用

3

天気予報に基づくソーラパネルを用いた センサネットワークの永続的運用手法

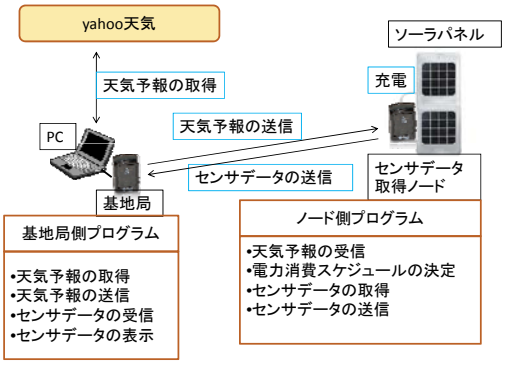
- 目的
  - 電力を枯渇させることなくできるだけ多くのデータを取得する
- 概要
  - 2種類の予測に基づき電力使用限界量を決める
  - 電力使用限界量に応じてセンサノードのサンプリングレートを変更
- 電力回復量予測
  - 短期予測(当日の電力回復量予測)
    - センサデータの照度から回復量を予測する
  - 長期予測(翌日以降の電力回復量予測)
    - 天気予報から回復量を予測する
- 電力使用限界量の決定
  - 電力使用限界量によってサンプリングレートを決定する

サンプリングレート低下により
消費電力も低下

計算によって導かれる
1日の電力利用の最大量

4

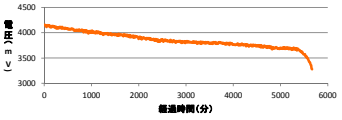
システム全体図



5

用語

- 残電比率
  - 満充電からの動作時間を100%とした際にどの程度動作時間が残っているかの割合
  - 残電比率は現在の電圧から求める
- 電力関係の単位が%表示
  - 残電比率と同様に電力の消費・回復量を%表示にしている
  - 現在の電圧によって電圧の降下速度が変わるため動作時間を基準に考える

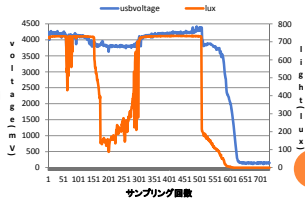


残電比率	残電圧
100%	4150
95%	4123.5
90%	4097
85%	4070.5
80%	4044
75%	4017.5
70%	3991
65%	3964.5
60%	3938
55%	3911.5
50%	3885
45%	3858.5
40%	3832
35%	3805.5
30%	3779
25%	3752.5
20%	3726
15%	3700
10%	3550
5%	3400
0%	3250

6

2種類の電力回復量予測-短期予測-

- 当日の電力回復量の予測
- センサノードの照度データより電力回復量を予測する
  - 現在の照度が日没まで続くと仮定し電力回復量を計算する
- 事前実験の結果
  - 照度が700LUX以上ある際に十分な電圧がかかり電力回復ができていた
  - 日照時間との関連がまだ調査できていない
- 予測方法のメリット
  - ソーラパネルの発電量と明るさにかなりの相関があり正確に回復量が予測できる
  - 今の照度を参照するため天気予報が外れた場合にも予測が可能



7

2種類の電力回復量予測-長期予測-

- 翌日以降の電力回復量の予測
- 基地局から送られてくる天気予報を基に電力回復量を予測する
- 予測方法のメリット
  - 予測しづらい将来の電力回復量についてもある程度予測をつけることができる

天気	晴れ	晴時々曇り	晴後曇・・・
電力回復量	200%	150%	100%・・・

- 事前実験より
  - 晴れの日1時間あたり約20%の電力を回復できる

正確さを重視した照度ベースの予測だけでは将来の照度が分からず
天気予報ベースの予測だけでは天気が外れた時に対応ができない
2種類の予測を利用することで正確に電力回復量を予測することができる

8

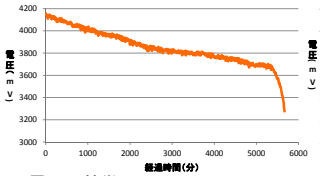
電力使用限界量の決定

- 仮定
  - ユーザの希望サンプリングレートは固定
- 限界量決定方法
  - ユーザの希望通りに動作させる
  - 例外処理
    - 50%を切る場合は残りの日数で15%を切らないように推移を計算
- 電力利用スケジューリングのポリシー
  - 電力が枯渇しない
  - 各日の電力利用量をできるだけ均一にする
    - 特定の日に全くサンプリングできない状況をなくすため

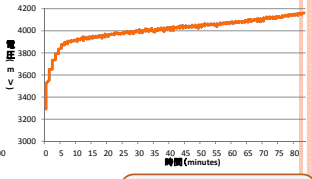
9

電力使用限界量の決定

10秒周期での電圧降下



USB接続での充電による電圧の推移



- 電圧の特徴
  - 残電比率15%(3700mV)以下では電圧の降下が激しい
  - 残電比率50%(3885mV)以下では電力回復が速い
- N日後の残電比率が50%を切ると予測される場合

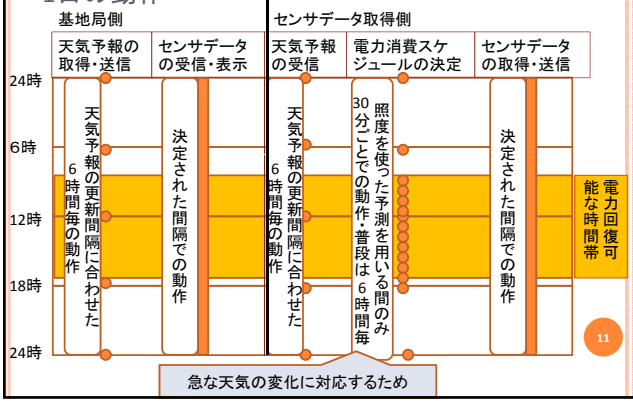
電力が不足しそうな日まで全ての日の動作を均一に減少させる

$$N\text{-}1\text{日までの各日の電力使用限界量} = \frac{N\text{日後の残電比率} - 15}{N+1}$$

$1 \le N \le 6$ 天気予報の週間予報が6日後までのため

10

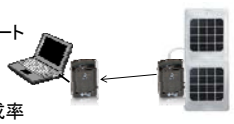
1日の動作



11

評価

- 仮定
  - ノードと基地局が1対1のシングルホップ通信
  - 短期予測が未実装のため天気予報の結果のみを利用
  - 電力回復量の予測には予報された天気を利用(2011年4月1日～2011年5月31日)
  - 電力回復量には気象庁の過去天気データの情報を利用(2011年4月1日～2011年5月31日)
  - ユーザの希望サンプリング周期は20秒
  - 初期残電比率は40%
  - 1日毎の推移をシミュレート
- 評価項目
  - 残電比率の推移
  - ユーザの希望レート達成率



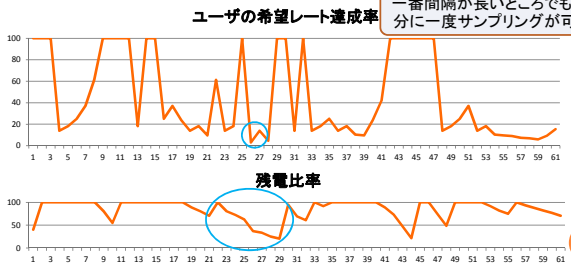
12

シミュレート結果

- 電力が枯渇しない
- 天気予報が外れても一定の動作を保証

22日～28日まで予測回復量が2回80%とあり電力回復量は期間中全て0と天気予報が外れているが電力が枯渇せずに動作できている

一番間隔が長いところでも10分に一度サンプリングが可能



13

今後の予定

- 短期予測の実装
  - 1日の日照時間と電力回復量の関係を調べる
- 長期予測の細かい電力回復量の決定
  - 同じ天気でも夏と冬では日照時間が違うため
- 実機による評価

14

おわりに

- 発表内容
  - 研究背景
  - 天気予報に基づくソーラパネルを用いたセンサネットワークの永続的運用手法
    - 短期予測
    - 長期予測
    - 電力使用限界量の決定
  - シミュレーションによる評価
  - 今後の展望
- まとめ
  - 状況に応じた電力回復量の予測やバッテリの性質を利用をすることで効率よく電力を利用することができる

15

Web アーキテクチャに基づく広域分散センサネットワークの管理機構

鳥居 隆弘 †† 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

地球環境の変化や自然災害の予測において，各地域における状況の変化を知るために広域な環境観測が不可欠となっている．また，近年のセンサ技術の発展により，無線デバイスの小型化や低価格化が進み，センサネットワークの大規模な構築が容易になっている．そこで，我々はマイクロストレージを用いたセンサデータの分散管理システムである P2P データポットの開発を進めている．P2P データポットは，無線通信機能を持った小型計算機で構成され，データポットに接続されているセンサネットワークが取得したセンサデータをデータポットのストレージに保存する．また，P2P データポットは他の P2P データポットと協調動作することが可能であり，これにより広範囲に及ぶセンサデータを分散管理することが可能である．しかし，現在の P2P データポットシステムでは，主に無線通信が可能な範囲で構成される単一の無線アドホックネットワーク内での運用を想定しており，離れた地域に存在する複数の無線アドホックネットワークを効率的に管理するための機能は十分であるとはいえない．また，アプリケーションソフトウェアが，データポットの構成するネットワークにアクセスし，データを利用するためには，基本的にはアプリケーションが動作するシステム自体に P2P データポットの機能を組み込むことが必要になる．そのため，性能や資源が限られたモバイル端末では，データポットの機能を組み込むことが困難な場合が出てくる．

本研究では，遠隔地にある複数の P2P データポットネットワークを統一的かつ効率的に管理するためのフレームワークとして，HTTP 通信とウェブブラウザの利用を前提としたデータポットの管理機構を実現することを目標とする．また，アクセスに特別なアプリケーションを必要としないことで，資源制約のある端末でのセンサデータ利用，データポット管理を可能とする．インターネットに接続され，ブラウザが使える環境であればセンシングデータを利用できるため，ユーザの利用範囲が拡大する．

本稿では，2 章で広域分散ネットワーク管理における要求について述べ，3 章でそれらの要求を実現するための手法について述べ，4 章で関連研究について述べる．

2 広域分散ネットワーク管理における要求

現在，P2P データポットは無線アドホックネットワーク通信で網羅できる範囲を 1 つの地域としており，各地域に分散された P2P データポットのネットワークはそれぞれ独立している状態である．より広範囲な環境センシングを容易に行うために，本研究では，分散された P2P データポットのネットワークを統一的に管理するとともに，センサネットワークの利用環境の拡大を実現する．本環境では，各 P2P データポットネットワーク内で 1 つ以上の P2P データポットがインターネットに接続されている環境を想定する．そこで，Web アーキテクチャを用いることで遠隔地に存在する複数の P2P データポットネットワークへのアクセスと管理を実現する．また，ブラウザからの制御を可能にすることで資源の制約のあるデバイスでも利用環境の拡大が可能となる．

本環境の実現において，2 つの課題が挙げられる．1 つは，HTTP 通信を用いた P2P データポットの制御である．現在の P2P データポットは P2P データポット間のやり取りを主に考えているが，HTTP 通信による制御を可能にするためには，データポットの機能を HTTP ベースで提供するための機構が必要となる．もう 1 つは，ユーザから P2P データポットへのアクセス手段である．HTTP 通信を行うに当たってアクセス先のデータポットの情報が必要となる．また，P2P データポットは分散データベースであるため必要なデータを持つデータポット，もしくは近隣のデータポットにアクセスし問い合わせを行うことで情報を取得する．そのため，目的のデータを持つデータポットの存在する地域と，近隣ノードへのアクセスを可能にすることが理想的である．

3 提案手法

本章では 2 章で述べた要求を実現するための手法について述べる．

本機構は，図 1 に示すように，Web サーバの機能を持つデータポット (以下，データポット Web サーバと呼ぶ)，とデータポット Web サーバの通信を行うための P2P ネットワークの機構からなる．ここでの P2P ネットワークはハイブリッド型 P2P ネットワークを検討している．データポット Web サーバは，設置された際にインデックスサーバに自身の情報を登録する．IP アドレス，ポート番号，センサの種類といったメタデータである．インデックスサーバの役割は，GoogleMap 等の Web

サービスとのマッシュアップによりセンシング環境を視覚化し提供することと，データポット Web サーバへのアクセスのための情報提供のみであり，センシングデータは管理しない．クライアントは，インデックスサーバにアクセスし，目的のセンシング範囲を選択する．インデックスサーバは最適なデータポット Web の情報を提供し，そこにクライアントをアクセスさせることで最適なデータポット Web サーバに割り振る．これによってクライアントはシームレスなセンシングデータの利用とデータポットの管理が可能になる．

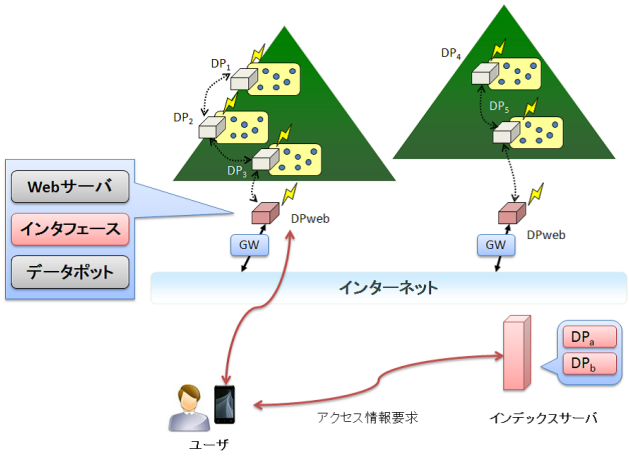


図 1　提案機構

3.1　データポットへのアクセス手段

P2P データポットを Web サーバとして利用するにあたり，全てのデータポットに固定アドレスを割り振るのは現実的ではないため，クライアントとデータポット Web サーバ間のアクセスを P2P 通信で実現する．構成を図 2 に示す．P2P データポットは JXTA の P2P フレームワークを用いてアドホックネットワークを構築しており，その上にさらにデータポット Web サーバとクライアント間の P2P ネットワークを構築するため，多段 P2P ネットワークとなる．データポット Web サーバは自身のメタデータをインデックスサーバに登録する．インデックスサーバも Web サーバとして働きクライアントからの HTTP アクセスによる問い合わせに対し，最適なデータポット Web サーバのインデックスを返す．クライアントは，そのインデックスを用いることでデータポット Web サーバへの HTTP アクセスが可能になる．

3.2　データポットの Web サーバ化

P2P データポットの制御をブラウザ上で実現するために，現在の P2P データポットに Web サーバとしての機能をもたせる．また，P2P データポットの機能をそのままブラウザ上で利用するために Web サーバと P2P データポットのメイン部分である問い合わせや制御機構である P2P データポット Core とのインタフェースを実装する必要がある．図 3 に示すように，データポット Web サーバを起動した際のインデックス登録の動作が必要となるため，P2P フレームワークの実装も必要となる．

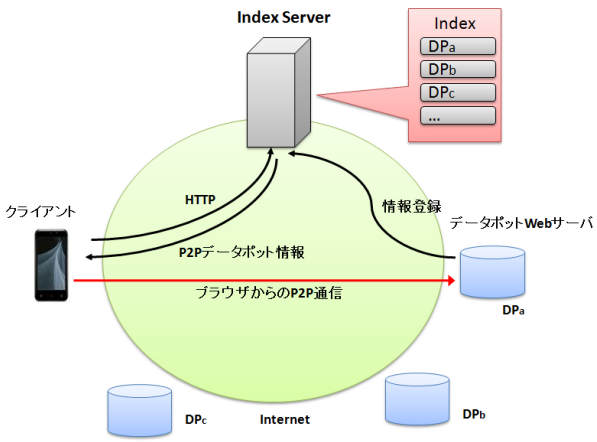


図 2　インデックスサーバによる P2P ネットワーク構築

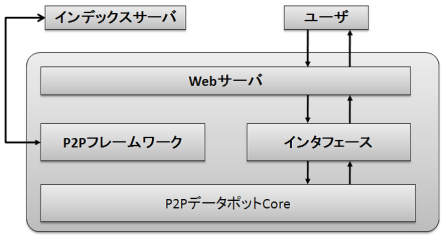


図 3　データポット Web のアーキテクチャ

4　関連研究

4.1　Pachube

Pachube(パッチベイ) は，センサに関する複数の実環境及び仮想空間をつなぐことを目的とするウェブサービスである．ユーザは Pachube のユーザ登録を行い任意のセンサを登録し，Pachube 独自の API を用いてセンサデータを配信する．センサデータは，Pachube サーバ上でマップとマッシュアップされ地図フォーマット上に印とメタデータの表示を行う．センサデータはグラフ化され表示される．また，独自の API での実装により，リアルタイムなデータの共有が可能となっている．本システムは，ネットワークインフラや電力のインフラが整っていることが前提となっている．

4.2　x-sensor

センサネットワークでは，センサノードの能力や無線通信の特性の制約が特に大きい，そのためシミュレー

ション上で実験するためのモデル化が非常に困難である．研究者が実ノードを用いた動作確認や実証実験を行う目的で大規模なセンサネットワークテストベッド環境である x-sensor が開発されている．x-sensor では，実ノードを用いた実験用拠点を複数設置し，それらを統合利用できる環境を提供している．これにより，研究者が提案した手法の動作確認や性能評価を、さまざまな環境下で行うことが可能となっている．また，実ノードが収集したリアルなデータを蓄積し・提供することで，センサデータ解析等の研究分野にも役立てる狙いがある．

5　おわりに

本稿では，広域分散ネットワーク管理の手法について述べた．今後は，インデックスサーバを用いた P2P ネットワークの構築のため，P2P フレームワークの調査と実装を行う予定である．

参考文献

[1] 藤崎友樹，鈴来和久，横田裕介，大久保英嗣：P2P データポット: センサーネットワーク向け分散型マイクロストレージアーキテクチャ，電子情報通信学会第 18 回データ工学ワークショップ (DEWS2007) (2007).

[2] Wilson, B.,　倉骨彰，佐野元之：JXTA のすべて－P2P Java プログラミング，日経 BP 社 (2003).

[3] Pachube,http://www.pachube.com/

[4] X-sensor,http://www-nishio.ist.osaka-u.ac.jp/research/index.php?

WEBアーキテクチャに基づく広域分散センサネットワークの管理機構

立命館大学　大久保槻田研究室
M1 島居 隆弘

1

目次

- はじめに
  - 環境センシングへの要求
- 関連研究(SensorWeb)
  - Pachube
  - X-sensor
- P2Pデータポット
  - 屋外環境を想定した環境センシングシステム
- 研究課題
  - センシング環境の拡大とセンサデータの汎用性の実現
- 提案手法
  - Webアーキテクチャを利用した分散ネットワークの管理
- おわりに

2

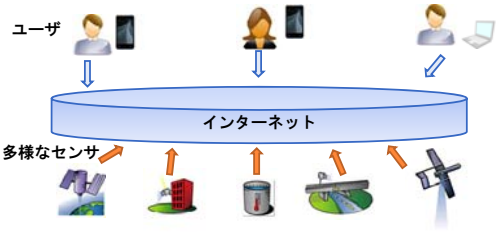
はじめに

- 様々な環境データの必要性
  - 地球環境観測
  - 自然災害の予測
- 様々な環境データの利用方法・利用環境
  - 環境観測 → 定期的にシステムからデータ取得
  - 防災 → 緊急時にデータを提供・利用できる環境
- いつでもどこにいても地球環境データを利用できる環境
- SensorWeb
  - 広域な環境観測技術

3

SENSORWEBとは

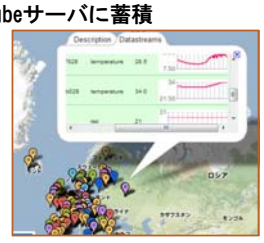
- センサデータ・センサをWeb上で利用・管理
- インターネットの利用できる環境であるため広域
- 異種センサの利用が容易
- Webを利用しているため汎用性に優れる



4

PACHUBE(パッチベイ)

- ユーザが好きなセンサを自由に設置登録
  - Pachube側がAPI・独自フォーマット提供
- 知りたいデータを検索(マップ・メタデータ)
- ユーザ間でリアルタイムなデータの共有
- センシングデータはPachubeサーバに蓄積
- センサをTCP/IPの利用できる環境に設置
- ネットワークインフラのある環境を前提



5

X-SENSOR

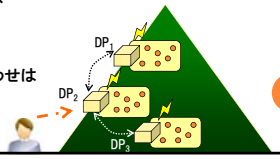
- 大阪大学　西尾研究室
- 制約の多いセンサネットワークではシミュレーションより実機での動作確認や実証実験が重要
- あらゆる環境にセンサネットワークの設置
- Web環境でのアクセスと管理
- 研究者が遠隔地から好きな環境で実機動作実験
  - トポロジ構成
  - データの収集

ブラウザ上でのネットワークトポロジの表示

6

P2Pデータポット

- 屋外環境センシングシステム
  - ネットワークインフラのない環境を想定
- 無線通信可能なストレージデバイス
- P2Pデータポットはストレージにセンサネットワークの取得するデータを保持
- ユーザは目的の環境のデータを近くのデータポットに要求するだけでデータの取得が可能
- データポット間はアドホック通信を行い協調動作をして目的のデータをユーザの元まで運ぶ
- 分散データベース
  - ユーザからは分散データベースであることは隠蔽される
- 汎用性が低い
  - システムとして閉じている
  - データポットに対する問い合わせはデータポットのみ可能



7

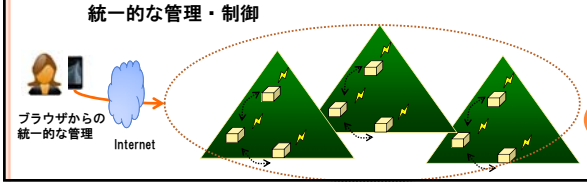
課題のまとめ

- 背景
  - いつでもどこにいても地球環境データを利用できる環境
- Sensor Web
  - 広域のセンシングを可能とする
  - HTTP通信を用いた汎用性の高いシステム
  - インフラのない環境ではセンシング困難
- P2Pデータポット
  - 屋外での環境センシングを目的
  - システムとして閉じているためユーザの利用環境が限られる

8

本研究の目標

- Webアーキテクチャを用いた分散センサネットワークの統一的な管理の実現
- 目的
  - センシング環境の拡大・HTTP通信による汎用性の実現
  - 広範囲のエリアからデータ取得
    - ネットワークインフラの整っていない環境も想定
  - 複数エリアのP2Pデータポットネットワークの統一的な管理・制御



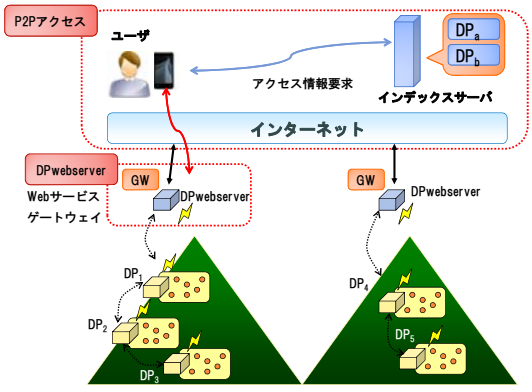
9

本環境における利点

- ブラウザベースでの管理の実現
  - PC・携帯など多様な端末からの管理が可能
    - 管理環境のユビキタス化
- HTTPの利用による汎用性の実現
  - HTTPベースでの管理システムの実現によりP2Pデータポットの複雑なネットワーク構成をユーザに対し隠蔽
  - 既存サービスとのマッシュアップによるUIの向上

10

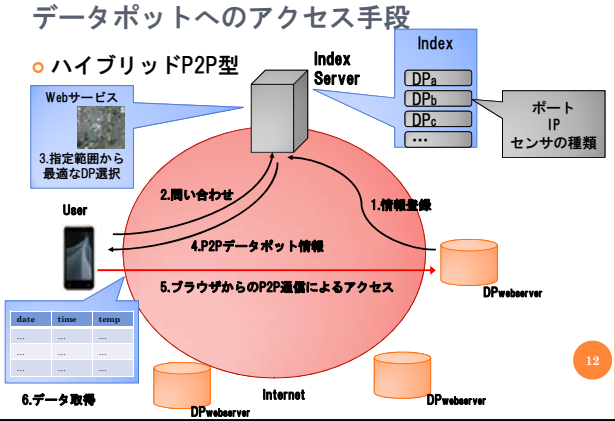
システム構成



11

データポットへのアクセス手段

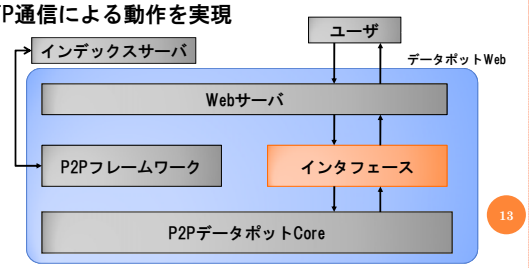
- ハイブリッドP2P型



12

データポットのWEBサーバ化

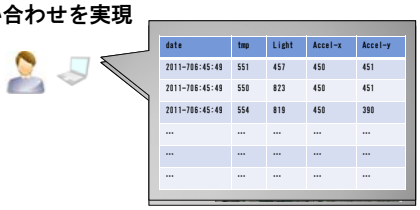
- Webサーバとして動作
- Webサーバとデータポット間のインタフェース実装
- HTTP通信による動作を実現



13

ユーザ利用方法

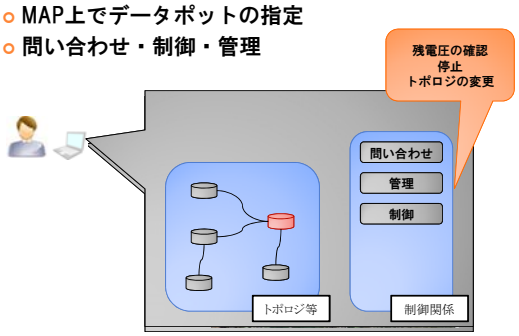
- インデックスサーバにブラウザからのアクセス
- MAP上でセンシング範囲の指定
- 問い合わせ(気温，照度)
- 問い合わせ結果取得(ブラウザ表示,APIからの提供)
- ネットワーク構成を意識しないシームレスな問い合わせを実現



14

管理者利用方法

- インデックスサーバにブラウザからのアクセス
- MAP上でデータポットの指定
- 問い合わせ・制御・管理



15

おわりに

- Webアーキテクチャに基づく分散ネットワーク管理
  - 各エリアのP2Pデータポットネットワークに対して Webアークテクチャによる管理を行い観測環境の 拡大とWebによる汎用性を実現する
- 今後の課題
  - データポットへのアクセス手段
    - P2Pネットワークのフレームワークの調査・実装
  - データポットのサーバ化
    - 既存機能をHTTP通信で利用可能にするインタフェース

16

編集後記

JSASS も今年で 12 回目を数えました．今年は龍谷大学を会場とし，合計 20 件の発表がありました．芝先生，ありがとうございました．今回はいつもの立命・龍谷・名工大・農工大に加え，拓殖大学からも参加がありました．先生方もいつもの面々で，と言いたいところでしたが，残念ながら並木先生が欠席され 2 日目には上原先生・毛利先生も欠席と，ちょっと例年よりも盛りあがりに欠ける印象を受けました．

話は変わりまして，今年は私も発表させていただきました．私の研究は無線ネットワークに関するものなのでちょっと場違いな感は否めませんでしたが，久しぶりの JSASS での発表を楽しむことができました．専門外の方々に説明するのは難しいなぁということを改めて感じ，来年はもっとわかりやすいプレゼンをしようと心に決めた次第です．

今年は近くでの開催でしたので，来年は旅行気分で・・・と今から期待しながら，また来年，JSASS で楽しい発表・聴講ができるのを楽しみにしています．

立命館大学　瀧本 栄二

今年も，農工大，名工大，拓殖大，龍谷大，立命館から 20 件もの発表があった. 瀧本さんが書かれていたように, 先生方が大学の業務等で忙しく，休みがちになっている中でこれだけの発表がなされたことに, 私は次の世代のがんばりを感じてる. とはいえ，昨年の JSASS の編集後記に書いたが，もっともっと JSASS が一般の研究会とは違った盛り上がりがあると良いと，今も思っている．そういうこともあって，今年は，手前味噌になってしまうが，毛利研究室の瀧本さんと樫山さんに発表を打診した．学生諸氏の発表はもちろん魅力的である．そこへさらに，研究を職とする者の発表をスパイスとして加えるといいのではという思いがあった．幸い，両氏とも快諾してくれた（私が打診しなくても発表されたかもしれない）．それ一つで全てががらっと変わるわけではないが，意義はあったと感じている．こういう，着実な試みをしていきたいと思っている．過去の JSASS では先生方も発表されている．今後はそういうものをまたやってみたり，デモンストレーションを前面に押し出したものを募集したり，招待講演を考えたり，さらには査読付きのものを取り入れたり，楽しく検討したい．また，議論の時間をさらにもう少し拡充するのも良いと考えている．「よそ行き」ではなく「本音」の議論を JSASS ならやってもいいのではないだろうか．JSASS の運営も落ち着いてきた．安定も良いことだが，そこからの進化・変化が研究へも波及し新たな流れを生み出すと確信している．

今回の JSASS 開催にご尽力頂いた，芝先生（龍谷大），横田先生（立命館），瀧本氏（立命館）に心から感謝したい．特に，プログラム作成からローカルアレンジまで広く担当された芝先生には，心から深く感謝を申し上げる．さらに，ご参加頂いた各氏，特にご発表頂いた方々と議論に参加して頂いた方々にも感謝したい．

本冊子の編集作業も終盤となったつい先ほど，ニュースが舞い込んできた．Apple の設立者の一人，取締役会長 Steve Jobs が 10 月 5 日に亡くなった．1955 年生まれ 56 才．冥福を祈る．

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 2011 (JSASS2011)

5 巻 1 号（通号 5 号）オンライン版　2011 年 10 月 28 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