先進的基盤ソフトウェア 6巻1号（通号6号）オンライン版 2012年10月29日発行
ISSN 1882-4196

先進的基盤ソフトウェア

Joint Symposium for Advanced System Software 2012
(JSASS2012)

2012年9月10日(月)・11日(火)

於 東京農工大学（東京都小金井市）

JSASS実行委員会

Joint Symposium for Advanced System Software 2012 (JSASS2012)

2012年9月10日(月)・11日(火)

東京農工大学（東京都小金井市）

プログラム

■ 9月10日(月)

○ オープニング: 13:30～13:40

○ セッション1: 13:40～15:40 分散システムと通信

1. 無線マルチホップネットワークにおけるTCPへのネットワークコーディングの適用
胡 懐穎（立命館大） ・・・・・・・・ 1

2. 組込みシステムを指向した分離Linuxプロセスロギング機構
Praween Amontamavut (拓殖大) ・・・・・・・・ 8

3. クラスタファイルシステムにおけるデータ配置の効率化
酒井 拓（龍谷大） ・・・・・・・・ 17

4. クラウドシステムにおける管理支援を目的とした可視化ツールの開発
落合 秀晴（拓殖大） ・・・・・・・・ 22

○ セッション2: 15:55～17:25 組込みシステム

5. 組込み向けマルチコアアクセラレータ用OpenCLライブラリと組込みOSの設計
坂本 龍一（農工大） ・・・・・・・・ 28

6. AndroidOSにおけるプロセス可視化環境の開発
中川 裕貴（拓殖大） ・・・・・・・・ 35

7. 複数のポリシ/メカニズムを搭載した学習向け組込みOSの実装
茂木 高宏（拓殖大） ・・・・・・・・ 41

○ 懇親会: 18:00～20:00

■ 9 月 11 日(火)

○ セッション 3: 10:30～12:00 セキュリティと信頼性

8. Alkanet: 仮想計算機モニタを用いたマルウェアトレーサ
大月 勇人（立命館大） ・・・・・・・・ 48

9. ソフトウェアによるメモリエラーの検出と訂正
若林 大晃（立命館大） ・・・・・・・・ 54

10. 管理 VM 監視のためのメモリアクセス通知機構の開発
猪飼 淳（名工大） ・・・・・・・・ 59

○ セッション 4: 13:00～14:30 オペレーティングシステム

11. ユーザ関数のカーネル内実行を可能とした
非同期カーネルサービスインタフェースの実現
安井 裕亮（名工大） ・・・・・・・・ 66

12. Cgroups による CPU 資源管理の性能評価
富樫 荘太（立命館大） ・・・・・・・・ 71

13. マルチコア・メニーコア混在型計算機における資源管理代行方式の試作と評価
深沢 豪（農工大） ・・・・・・・・ 78

○ クロージング: 14:30～14:40

無線マルチホップネットワークにおけるTCPへのネットワークコーディングの適用

胡　懐穎　6171110028-7　*ko@asl.cs.ritsumei.ac.jp*
立命館大学大学院研究科　毛利研究室

1　はじめに

近年，無線マルチホップネットワークにおけるネットワークコーディングが注目されている．

ネットワークコーディング [1] とは，中継ノードにおいて N 個のパケットを符号化によって M 個（ただし，N>M）のパケットを生成・送信し，受信側で復号化することでパケットの総送信数を減らす技術である．したがって，ネットワークコーディングを無線マルチホップネットワークに適用することで，中継ノードのパケット送信数を減らし，通信帯域幅を節約することが可能となる．

ネットワークコーディングは，その性質上，片方向通信には適用できず，また双方向通信に関しても，上りと下りのパケット数及びデータレートによって効果が変動する．このことから，上りと下りの各パケット数とパケット生成レートがほぼ同等である TCP に着目した．

TCP では，信頼性の高い通信を実現するため，DATA を受信すると ACK を返信する．したがって，DATA と ACK からなる双方向のパケットを符号化することで，ネットワークコーディングの適用が可能となり，それによるネットワーク性能の向上が期待できる．既存研究である PiggyCode[2] は，インタフェースキュー内にある TCP パケットを対象として符号化を行う．しかし，低送信レート時に，インタフェースキューに TCP パケットが溜まらないため，符号化が行われない．したがって，PiggyCode では，常にネットワークコーディングの効果を得られない．

そこで，本稿では，ネットワークコーディングを TCP 通信に適用した PiggyCode をベースとし、その性能を向上させるため，中継処理に待機時間を挿入する手法を提案する．以下，本稿では，2 章で PiggyCode について述べ，3 章で提案手法について述べる．そして，4 章で評価について述べ，5 章で今後の予定について述べる．

2　PiggyCode

2.1　概要

PiggyCode[2] は，同一フロー内の DATA と ACK に着目し，符号化を行うことでパケットの伝送回数を削減する．

図 1 は，PiggyCode を使用したマルチホップ通信の例を示している．中継ノードは，送信時に DATA と ACK を符号化する．符号化に用いるパケットは，復号化に必要なためバッファに一時保存する．DATA を受信した場合，インタフェースキューの中からもっとも古い ACK と符号化を行う．符号化したパケットは，符号化ヘッダを付与し送信する．また，ACK が存在しない場合は，通常の中継処理に従って送信する．ACK を受信した場合も同様である．

ノードは，受信したパケットに符号化ヘッダが付与されていた場合に復号化処理を行う．復号化処理では，符号化ヘッダに記録された符号化した元となるパケットの TCP シーケンス番号に基づき，バッファから対象のパケットを取り出し復号化を行う．復号化により，新たに得られたパケットは，上位レイヤに渡され，そのコピーが次の復号化のためにバッファに記憶する．

図 1　PiggyCode

2.2　PiggyCode の課題点

PiggyCode は，インタフェースキュー内で TCP パケットを符号化対象とする．PiggyCode には，以下の２つの課題が存在する．

- 低送信レート時では，中継ノードのインタフェースキューに TCP パケットが溜まりにくいため符号化が行われない．
- 高送信レート時では，インタフェースキューに待機するパケットは，DATA または ACK どちらか一種類が溜まる傾向にあるため効率的に符号化を行うことができない．

すなわち，PiggyCode には常にネットワークコーディングの効果を最大限に活用できない．符号化の効果を高めるためには，送信レートに影響されずに符号化率を向上させることが必要である．

3　提案手法

本章では，2.2 節で述べた課題を解決するためにパケットの中継処理に待機時間を挿入することにより，TCP パケットの符号化機会を増やしパケットの符号化率を向上させる手法を提案する．具体的には, インタフェースキューとは別に待機用のキューを設け，インタフェースキュー挿入前に当該キューで一定時間，パケットの送信を待機させる．待機中に符号化が可能となった場合は，符号化を行った後に即座にインタフェースキューに挿入する．符号化されずに待機時間を経過した場合は，そのままインタフェースキューに挿入する．これにより，PiggyCode のようにインタフェースキューの状態に左右されず，符号化の機会を増やすことができる．待機時間を挿入するパケットは，DATA のみとした．これは，一方のパケットに待機させ，一方のパケットの流れを妨げないことで，待機時間中に符号化できる確率を上げるためである．

4　評価

4.1　シミュレーション環境

提案手法の有効性を確認するため，シミュレーションによる評価を行った．ネットワークトポロジは，送信元から送信先ノードまで２〜４ホップのチェーントポロジとした．比較対象は，通常の TCP（NewReno）と PiggyCode とし，スループット，RTT および ACK の伝送回数の計測を行った．ACK の伝送回数は，すなわち，どれだけのパケットが符号化されなかったかを表す．評価は，３つのトポロジと，待機時間を 10ms から 90ms まで 20ms 単位で変化させた場合について行った．

4.2　シミュレーション結果と考察

3 ホップ（図２，図５）以下ではスループットが最大 10 %の向上を達成した．待機時間を挿入することにより，TCP パケッ

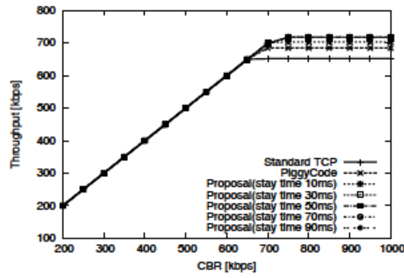


図２　スループット評価（2 ホップ）

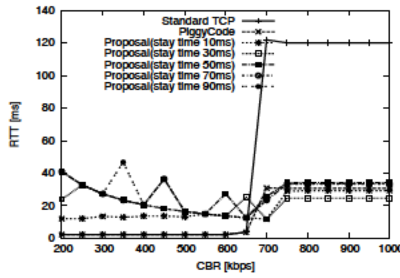


図３　RTT（2 ホップ）

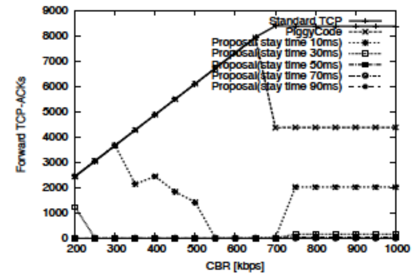


図４　ACK の伝送回数（2 ホップ）

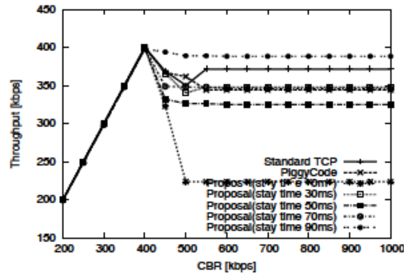


図５　スループット評価（3 ホップ）

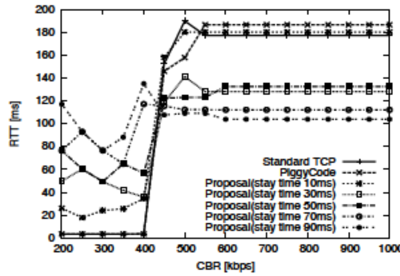


図６　RTT（3 ホップ）

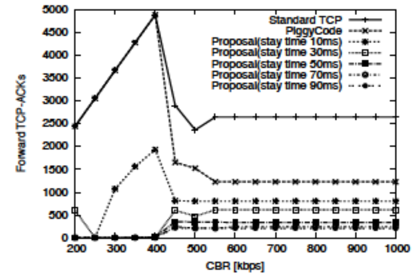


図７　ACK の伝送回数（3 ホップ）

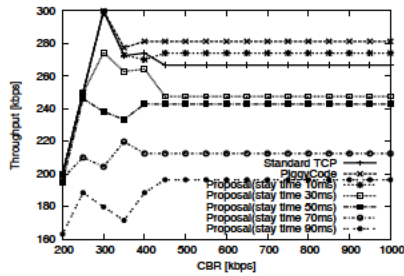


図８　スループット評価（4 ホップ）

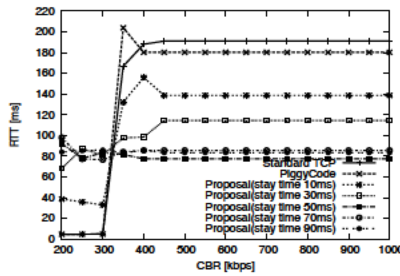


図９　RTT（4 ホップ）

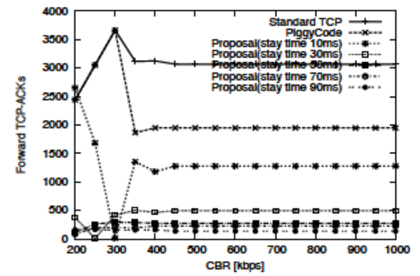


図 10　ACK の伝送回数（4 ホップ）

ト同士（図４，図７）が効率的に符号化できたため，利用できる帯域幅が増え，結果としてスループットの上昇につながった．スループットに関して，待機時間が長いほど高いスループットにつながったことを確認した．また，図３および図６より，低トラフィック時では，待機時間の挿入により RTT は増加した．しかし，高トラフィック時では，符号化した後即座に送信されることにより RTT は大幅に減少した．これにより，低トラフィック時では，待機時間を短く設定することにより，RTT への影響も少なくなると考えられる．

4 ホップ（図８）では，PiggyCode が有効であった原因として，プロミスキャス受信の失敗によりパケットロスが多くなったことと，待機時間分パケットの到達が遅くなったことが考えられる．スループットに関しては，ホップ数が 2 と 3 の場合と異なり待機時間が長いほど高い性能が悪化した．しかし，RTT（図９）およびパケットの送信回数（図１０）は改善している．

以上のことから，ホップ数が少ない場合では提案手法が特に有効である．しかし，ホップ数が多い場合では，待機時間を 0 とするのがよいと考えられる．また，トラフィック量が多いときに待機時間を短く，少ないときは長くすることが有効であると考えられる．以上から，本稿で提案する待機時間の挿入は有効であるが，その長さとホップ数，トラフィック量に応じて適切に調整することが，提案手法の有効性を高めるためには重要であることが明らかとなった．

5　おわりに

本稿では，PiggyCode の課題と，その課題を解決するための提案手法を述べ，実装し評価を行った．今後は，ホップ数と待機時間の関係の明確化し，動的に待機時間の調整方式について検討することともにさらに複雑な環境を想定し，提案手法の再評価を行う．

参考文献

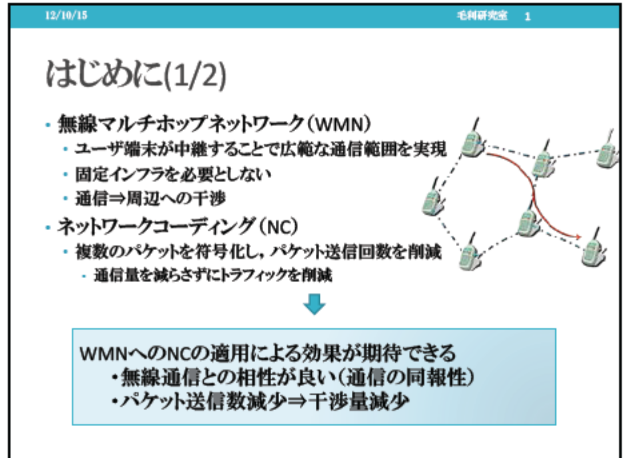
[1] S Katti, H Rahul, W J Hu: “XORs in the air: Practical wireless network coding” In Prof. of SIGCOMM 2006, 2006.

[2] Luca Scalia, Fabio Soldo, Mario Gerla: “Piggy- Code: A MAC Layer Network Coding Scheme to Improve TCP Performance OverWireless Networks，” In Prof. of GLOBECOM’07.pp.3672-3677，2007

12/10/15 毛利研究室 1

無線マルチホップネットワークにおけるTCPへのネットワークコーディングの適用

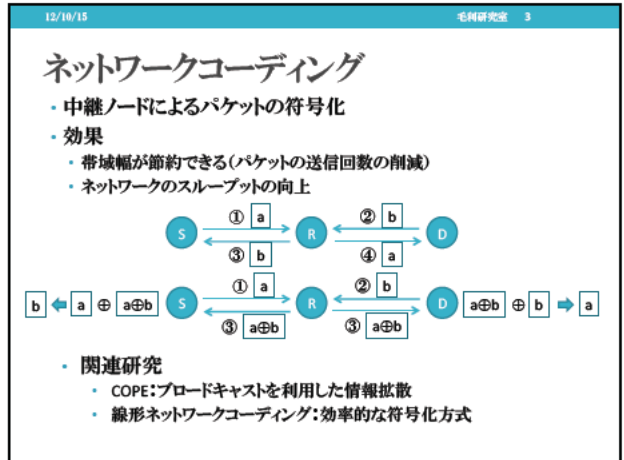
立命館大学大学院　毛利研究室
胡　懐穎

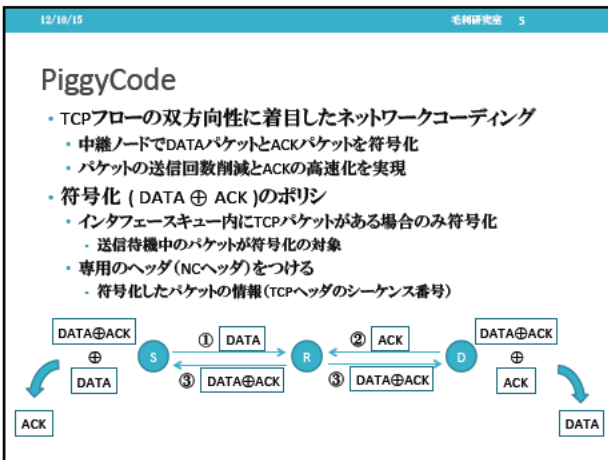


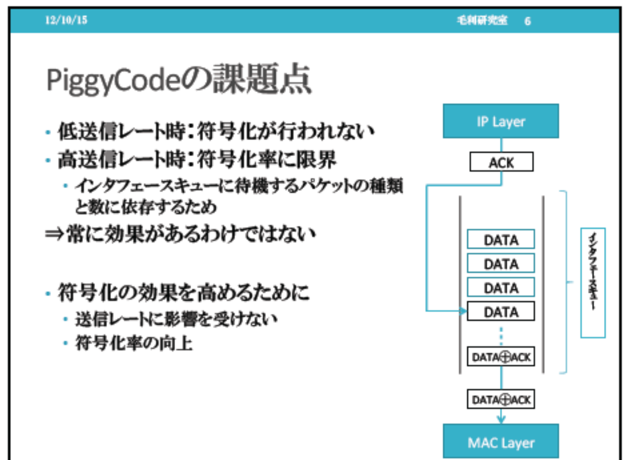
12/10/15 毛利研究室 2

はじめに(2/2)

- 既存研究(PiggyCode)
  - WMNにおけるTCP通信にNCを適用した研究
  - DATAとACKの双方向性に着目
  - DATAとACKを符号化し，パケットの送信回数を削減
  - 課題：送信レートが低い時，効果を得にくい
- 解決手法の提案
  - 中継時に待機時間を挿入することで、符号化機会を増やし、パケットの送信回数を削減する
  - 通信環境(ホップ数など)を考慮した待機時間の調整



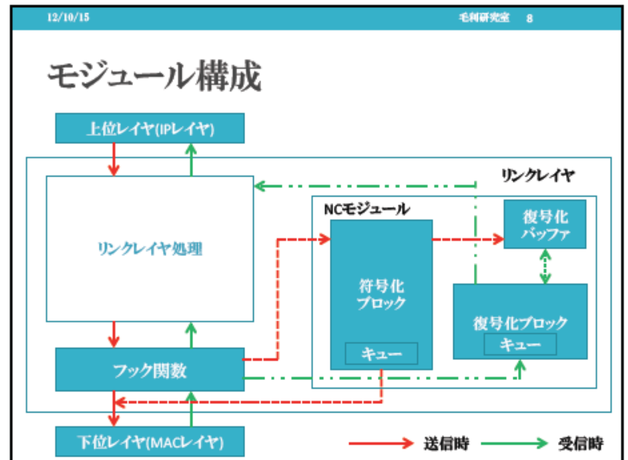


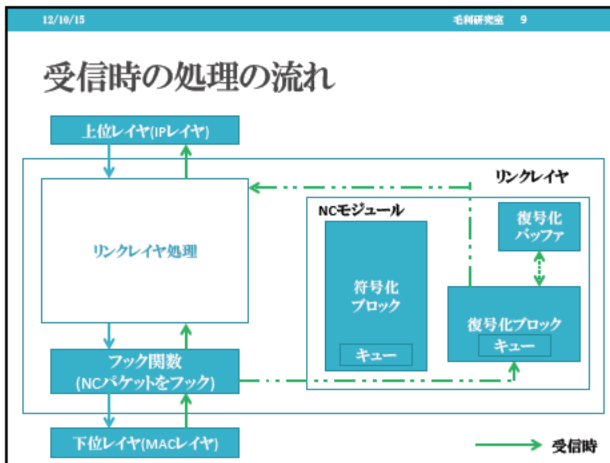


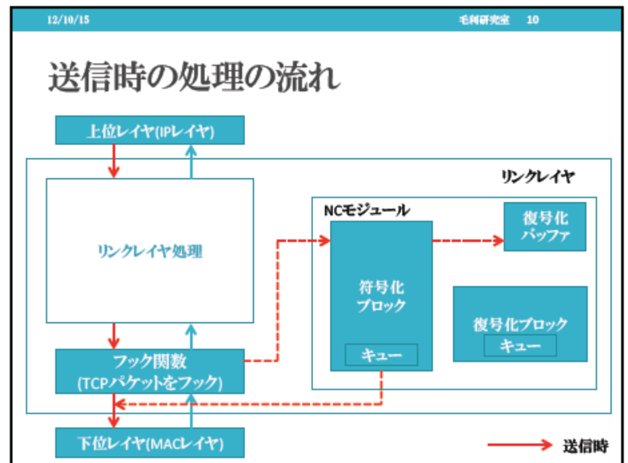
12/10/15　毛利研究室　7

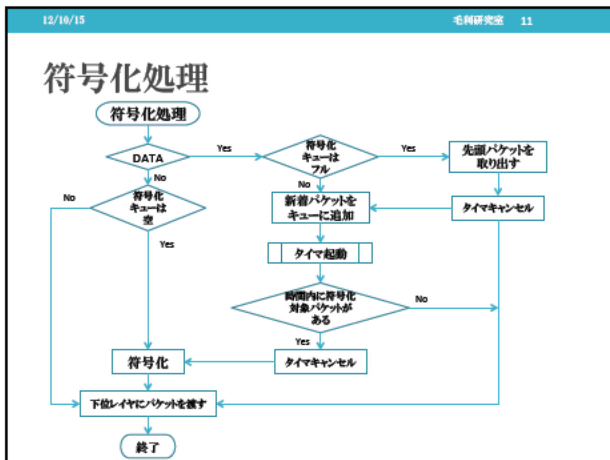
提案手法

- 中継時に待機時間を挿入することで符号化機会を増やす
  - 待機時間内に符号化できれば即座に送信
  - 待機時間経過後は符号化せずに送信
  - DATAパケットに待機時間を挿入
    - ACKへの挿入はしない
      - 待機による遅延の過剰な増加を避けるため
      - パケットを必要以上に停滞させないため
- 適応的な待機時間の調整









12/10/15　毛利研究室　12

シミュレーション環境

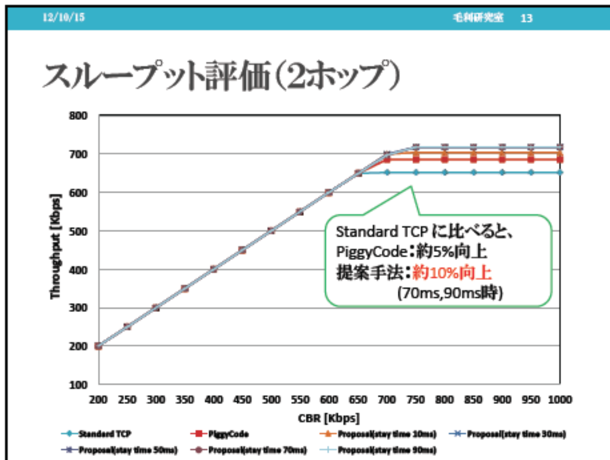
パラメータ	設定値
シミュレータ	NS-2 (Ver2.34)
伝搬モデル	2-rayモデル
シミュレーション時間	100秒
パケットサイズ	1024バイト
ノード間距離	200m
通信距離	250m
データレート	2Mbps
ベーシックレート	1Mbps
アプリケーション	CBR(200～1000Kbps)
TCP	TCP/NewReno
待機時間	10,30,50,70,90ms

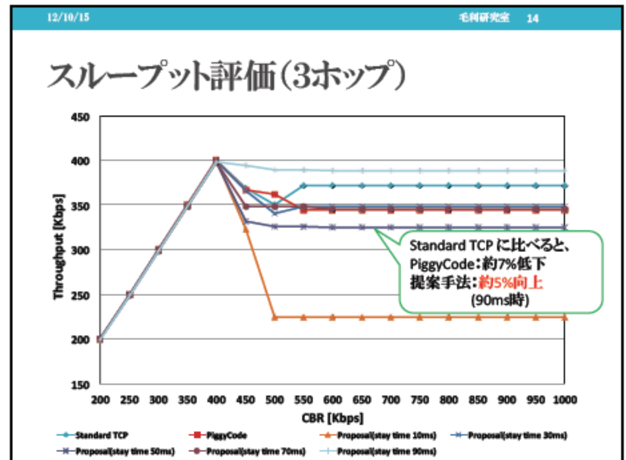
2ホップ　S — R — D

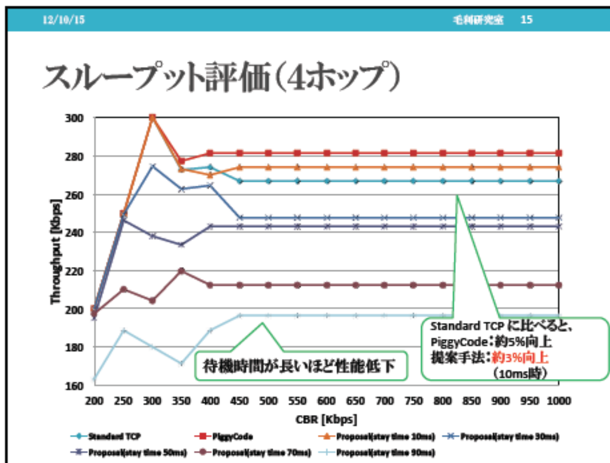
3ホップ　S — R1 — R2 — D

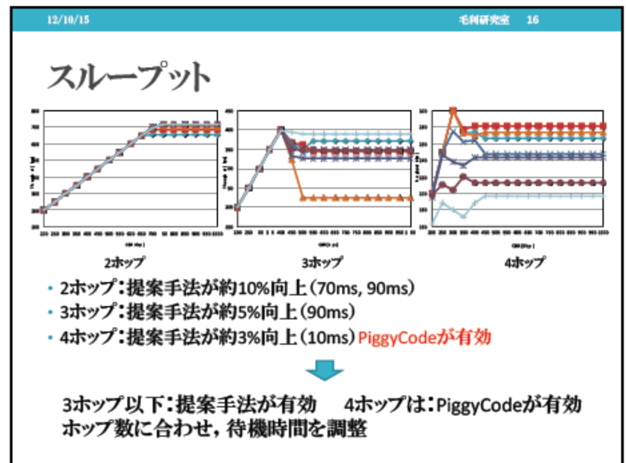
4ホップ　S — R1 — R2 — R3 — D

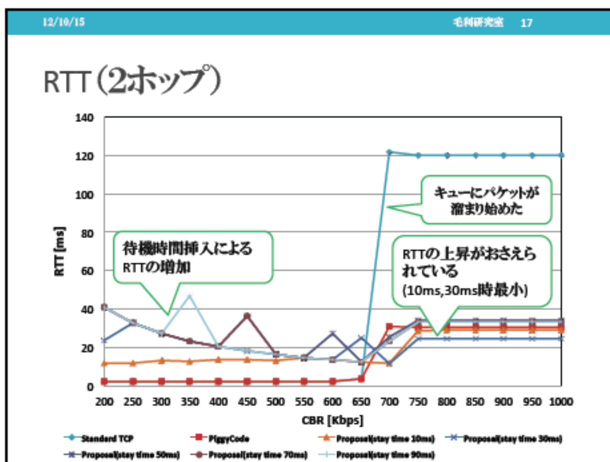
- 評価項目
  - スループット
    - 提案手法による性能向上
  - RTT
    - 待機時間挿入の影響
  - 中継ノードにおけるACKの伝送回数
    - 符号化率の変化
- これらの結果を待機時間をどのように調整するかの参考とする

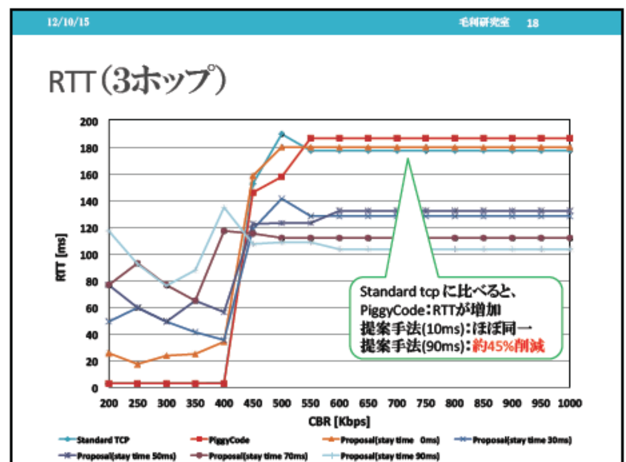

12/10/15
毛利研究室 13
スループット評価(2ホップ)
Throughput [Kbps]
CBR [Kbps]
Standard TCP に比べると、
PiggyCode:約5%向上
提案手法:約10%向上
(70ms,90ms時)
Standard TCP
PiggyCode
Proposal(stay time 10ms)
Proposal(stay time 30ms)
Proposal(stay time 50ms)
Proposal(stay time 70ms)
Proposal(stay time 90ms)

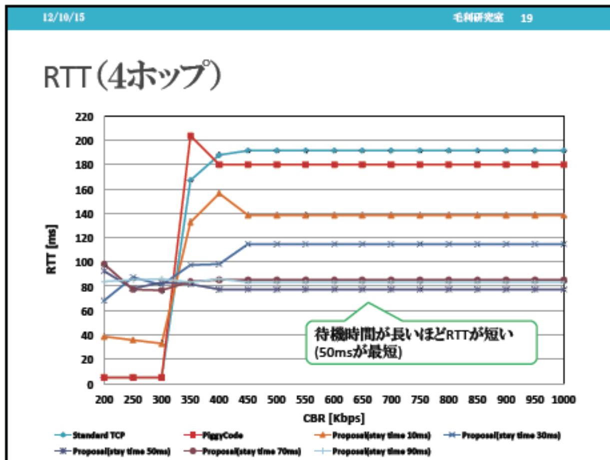

12/10/15
毛利研究室 14
スループット評価(3ホップ)
Throughput [Kbps]
CBR [Kbps]
Standard TCP に比べると、
PiggyCode:約7%低下
提案手法:約5%向上
(90ms時)
Standard TCP
PiggyCode
Proposal(stay time 10ms)
Proposal(stay time 30ms)
Proposal(stay time 50ms)
Proposal(stay time 70ms)
Proposal(stay time 90ms)

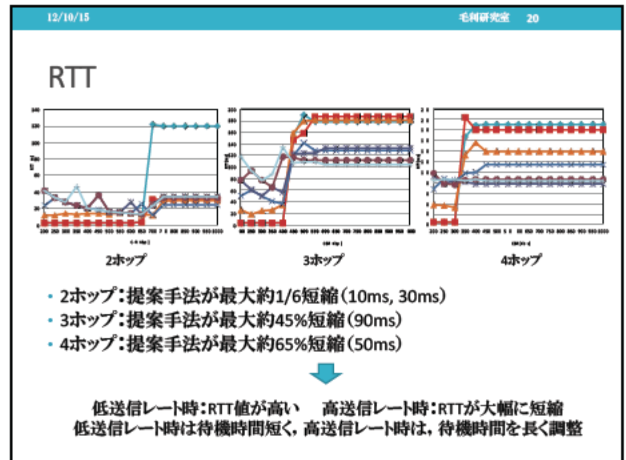

12/10/15
毛利研究室 15
スループット評価(4ホップ)
Throughput [Kbps]
CBR [Kbps]
待機時間が長いほど性能低下
Standard TCP に比べると、
PiggyCode:約5%向上
提案手法:約3%向上
(10ms時)
Standard TCP
PiggyCode
Proposal(stay time 10ms)
Proposal(stay time 30ms)
Proposal(stay time 50ms)
Proposal(stay time 70ms)
Proposal(stay time 90ms)

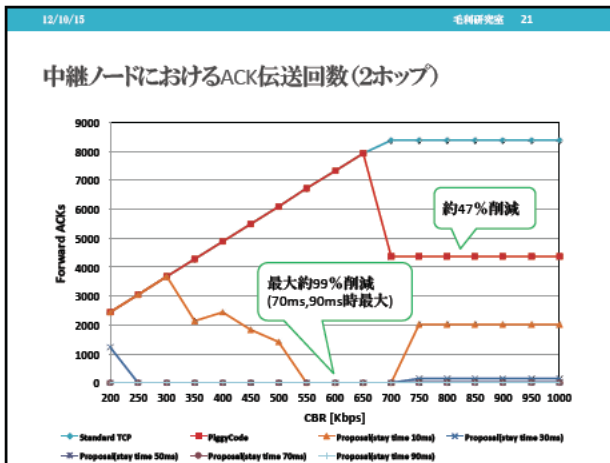

12/10/15
毛利研究室 16
スループット
2ホップ
3ホップ
4ホップ
2ホップ:提案手法が約10%向上(70ms, 90ms)
3ホップ:提案手法が約5%向上(90ms)
4ホップ:提案手法が約3%向上(10ms) PiggyCodeが有効
3ホップ以下:提案手法が有効 4ホップは:PiggyCodeが有効
ホップ数に合わせ,待機時間を調整

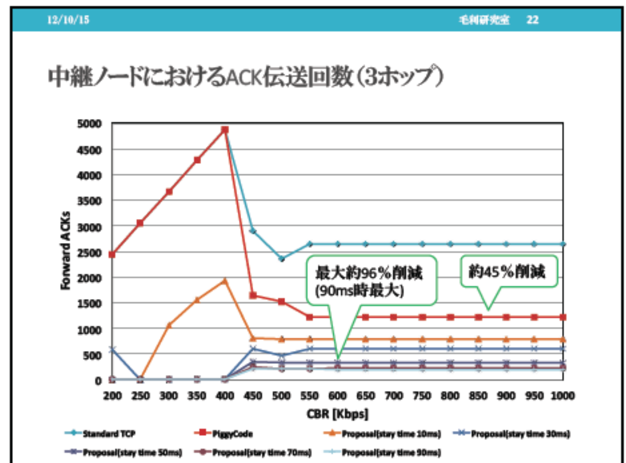

12/10/15
毛利研究室 17
RTT(2ホップ)
RTT [ms]
CBR [Kbps]
キューにパケットが
溜まり始めた
待機時間挿入による
RTTの増加
RTTの上昇がおさえら
れている
(10ms,30ms時最小)
Standard TCP
PiggyCode
Proposal(stay time 10ms)
Proposal(stay time 30ms)
Proposal(stay time 50ms)
Proposal(stay time 70ms)
Proposal(stay time 90ms)

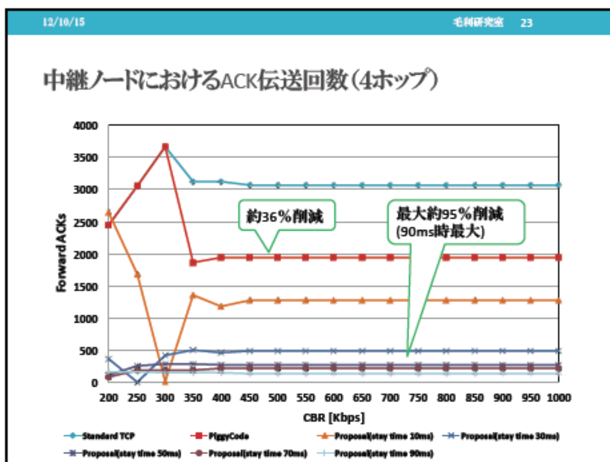

12/10/15
毛利研究室 18
RTT(3ホップ)
RTT [ms]
CBR [Kbps]
Standard tcp に比べると、
PiggyCode:RTTが増加
提案手法(10ms):ほぼ同一
提案手法(90ms):約45%削減
Standard TCP
PiggyCode
Proposal(stay time 0ms)
Proposal(stay time 30ms)
Proposal(stay time 50ms)
Proposal(stay time 70ms)
Proposal(stay time 90ms)

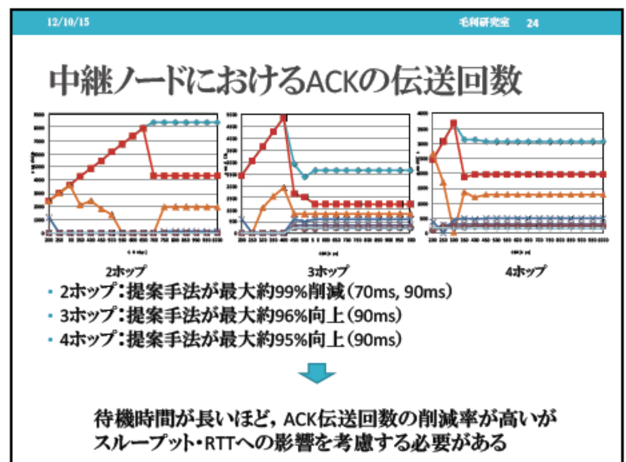

12/10/15
毛利研究室 19
RTT(4ホップ)
待機時間が長いほどRTTが短い
(50msが最短)
RTT [ms]
CBR [Kbps]
Standard TCP
PiggyCode
Proposal(stay time 10ms)
Proposal(stay time 30ms)
Proposal(stay time 50ms)
Proposal(stay time 70ms)
Proposal(stay time 90ms)


12/10/15
毛利研究室 20
RTT
2ホップ
3ホップ
4ホップ
・2ホップ:提案手法が最大約1/6短縮(10ms, 30ms)
・3ホップ:提案手法が最大約45%短縮(90ms)
・4ホップ:提案手法が最大約65%短縮(50ms)
低送信レート時:RTT値が高い　高送信レート時:RTTが大幅に短縮
低送信レート時は待機時間短く，高送信レート時は，待機時間を長く調整


12/10/15
毛利研究室 21
中継ノードにおけるACK伝送回数(2ホップ)
約47%削減
最大約99%削減
(70ms,90ms時最大)
Forward ACKs
CBR [Kbps]
Standard TCP
PiggyCode
Proposal(stay time 10ms)
Proposal(stay time 30ms)
Proposal(stay time 50ms)
Proposal(stay time 70ms)
Proposal(stay time 90ms)


12/10/15
毛利研究室 22
中継ノードにおけるACK伝送回数(3ホップ)
最大約96%削減
(90ms時最大)
約45%削減
Forward ACKs
CBR [Kbps]
Standard TCP
PiggyCode
Proposal(stay time 10ms)
Proposal(stay time 30ms)
Proposal(stay time 50ms)
Proposal(stay time 70ms)
Proposal(stay time 90ms)


12/10/15
毛利研究室 23
中継ノードにおけるACK伝送回数(4ホップ)
約36%削減
最大約95%削減
(90ms時最大)
Forward ACKs
CBR [Kbps]
Standard TCP
PiggyCode
Proposal(stay time 10ms)
Proposal(stay time 30ms)
Proposal(stay time 50ms)
Proposal(stay time 70ms)
Proposal(stay time 90ms)


12/10/15
毛利研究室 24
中継ノードにおけるACKの伝送回数
2ホップ
3ホップ
4ホップ
・2ホップ:提案手法が最大約99%削減(70ms, 90ms)
・3ホップ:提案手法が最大約96%向上(90ms)
・4ホップ:提案手法が最大約95%向上(90ms)
待機時間が長いほど，ACK伝送回数の削減率が高いが
スループット・RTTへの影響を考慮する必要がある

12/10/15 毛利研究室 25

まとめと考察

- 提案手法導入の効果
  - ホップ数が3ホップ以下でスループット最大10%の向上を達成
    - 待機時間が長いほど効果が大きい
    - 低トラフィック時のRTTは増加
      - 待機時間挿入による遅延
    - 高トラフィック時のRTTは大幅に減少
      - 符号化により即座に送信されるため待機時間短い
  - 4ホップではスループットが向上しなかった
    - プロミスキャス受信失敗によるパケットロスが多発
    - 待機時間が長いほど性能が悪化した
    - RTT，パケット送信回数は改善している

低トラフィック時に
待機時間小さい方が有効

現時点では，
- ホップ数が少ない場合特に有効である（待機時間100ms程度）
  - 低トラフィック時30ms，高トラフィック時90msがよい
- ホップ数が多い場合は待機時間を0とするのがよい

ただし、プロミスキャス通信の信頼性向上により改善が期待できる

12/10/15 毛利研究室 26

今後の予定

- ホップ数と待機時間の関係の明確化
  - ホップ数に合わせ、動的に制御する仕組みの構築
- プロミスキャス通信の信頼性向上
  - MACレイヤの再送機能を拡張
- 複雑な環境を想定した提案手法の再評価
  - 複数トラフィック、複雑なトポロジ

組込みシステムを指向した分離 Linux プロセスロギング機構

Praween Amontamavut†、中川 裕貴†、早川 栄一††

情報系の組込みシステムを対象とした省メモリのネットワーク上データ共有・遠隔ロギング可能な Linux プロセスのロギング機構の開発を行った．ロギングの高い効率とサイズが大きいログデータの共有・管理の2点に注目する。ロギングの効率を上げるため、最適化された圧縮フォーマットを提案しログエクスポート方向を変更した。サイズが大きいログデータの共有・管理を行うため、ロギング機構を「ロギング環境」と「ロギング監視環境」に分離し遠隔ロギング機構を実装した。

1. はじめに

1.1 背景

現在，タブレット PC やスマートフォンで起動する Android は多く使用されている．Android は Linux カーネルをベースとして多層構造的なプラットフォームで設計されており，組込みシステムで扱われている.[1]

組込みシステムにおけるリアルタイム性の検証や学習では，システムログが重要になっている．例えば，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，学習するプロセススケジューリングのデバッグでは，ログを元に分析や可視化が行われている.[2]

早川研究室では，カーネルプロセスおよ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.[3]

プロセススケジューリングのデバッグでは，一般にトレーサによるコンテキストスイッチログを扱う．プロセスの可視化も，コンテキストスイッチログを取得し，保存してから解析を行う．システムを利用しながらスケジューラを監視するには，両者とも長時間のログ取得が必要になる．しかし，生成されたログのサイズが大きく，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．

1.2 目的

本研究の目的は，組込みシステムに用いられる Android で使用される Linux カーネルに対して，プロセススケジューリングのログ取得機構を，より効率のより機構に改良する．具体的には，ログを生成するから表示するまでにオーバヘッドが少ない最適化したプロセスログ取得方法や機構改良方法を考え，実装，評価することを通して，Linux のプロセスログ取得機構について，提案することである．

† 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

2. 研究の概要

2.1. ftrace 機構とその問題点

本研究では Android に使用される Linux の ftrace を対象とする．ftrace は，図1のように非ブロッキング方式でデータをリングバッファに収集し, DebugFS を通じてログデータをリングバッファから Export Buffer へブロッキング方式で収集してユーザ空間に転送する．データは文字列化部により文字コードで表現されている．

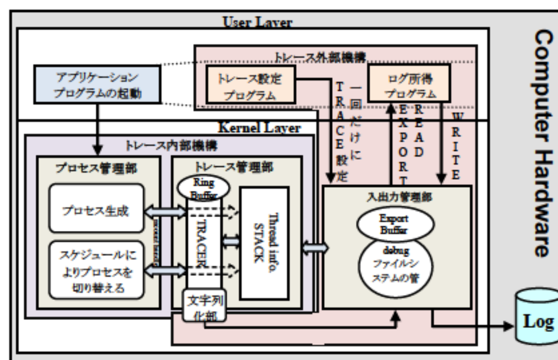


図1従来のトレース機構

この方式には次の問題点がある．

1） システムが提供する 1 ページ[4]の Export Buffer に格納されるコンテキストスイッチログは，文字列化するのでバッファ内に格納できるサイズが小さい．これにより, リングバッファから Export Buffer へのデータ書込みが間に合わず，タイマ割込みが発生する毎にエクスポートするコンテキストスイッチの情報が消失する問題が発生する可能性がある．

2） ログデータは Debug FS を介してユーザ空間にデータを提供する形になっているので，大量のログデータ転送においてオーバヘッドがあり，スケジューリングにも影響を与える． また，組込みシステムは機器内の Flash ROM 容量には制限があり，機器内に大容量のログデータを保存することは難しい．

2.2. システムの設計

ログを保持する容量を大きくして長時間のロギングが可能として遠隔ロギングが可能とするため，システムを2つの環境に分割する．ログを監視する環境(Log Monitoring Environment or LME)とロギング環境(Logging Environment or LE)である．

LME は遠隔ロギング制御命令や多様なログデータ形式に解凍命令ができる. LME は複数であればログデータはストリーミング通信で共有できる．

LE はログを収集する側(Log Collection Part or LCP)とログを生成する側(Log Generation Part or LGP)に分割する．設計の詳細は 2.3 に説明する．

LME と LE の間には HTTP の GET メソッドを利用する．このシステムモデルは遠距離であっても，ロギング制御やログデータを監視することができる．また，多様なログ形式を提供するので，多様なデバッガーツールにログを提供することができる．

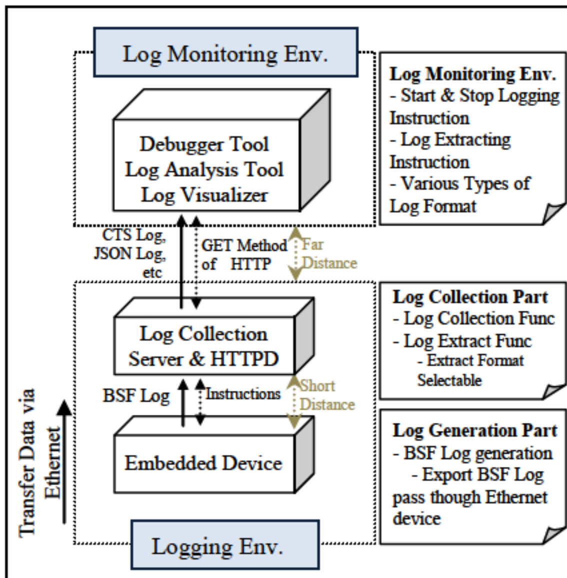


図 2 システムの全体構成モデル

2.3. ロギング環境の設計

図 3 にロギング環境(LE)の機構モデルを示す．

LE は記憶容量不足の問題を回避するため，LGP と LCP の二つから構成する．

LGP は組込みシステムから成り，従来のトレース機構を拡張した．入出力管理部には，トレース管理部，ユーザからのトレース設定，LCP と通信するトレースドライバを導入した．トレースドライバは Ethernet を通じて，LCP にログデータを送信する形にする．

また，バッファ溢れによるデータ消失を防ぐ，データ収集効果を上げるため，圧縮部を設計し，リングバッファと Export Buffer の間に元の文字列化部の代わりに追加した．

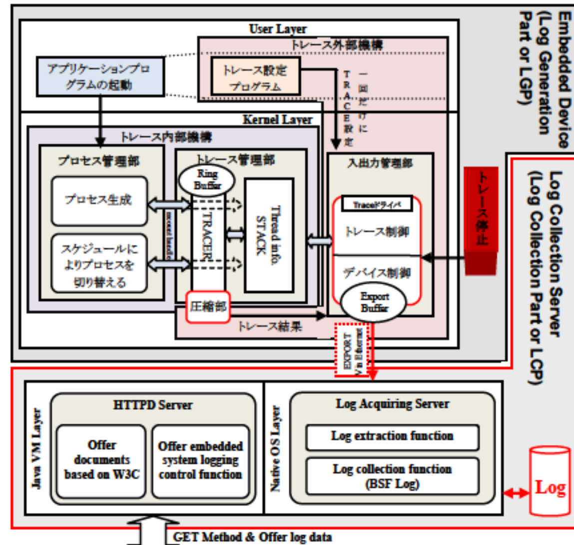


図 3 ロギング環境の機構モデル(LE)

LCP はログを収集するサーバから成り，三つの機能がある．

1) Ethernet を通じた圧縮されたログの収集をするログ取得機能
2) CTS 形式というオリジナルな形式のコンテキストスイッチログ形式，JSON 形式，XML 形式のログ形式選択可能なログ解凍機能
3) Web Server でログを提供する機能

2.4. 圧縮機能の概要と設計

リングバッファから出力したデータは, Export Buffer に蓄えられる．このため，リングバッファをログの一行を文字コードの文字列部とバイナリコードのバイナリ部に分割し，オリジナルのコンテキストスイッチログの特徴を分析し，圧縮するようにした．圧縮するフォーマットは mixed Binary and String Format である．(以下 BSF と呼ぶ) 圧縮した BSF Log は 図 4 に示す．

特徴は次の2点である．

1) 数値などの文字列ではないデータは，データの特徴により分類する．
    I. そのまま使用可能なバイナリデータ
    II. データ表現の範囲を縮められるデータ
2) 文字列のデータは同じデータの繰返し頻度による文字列可変長できるデータにする．

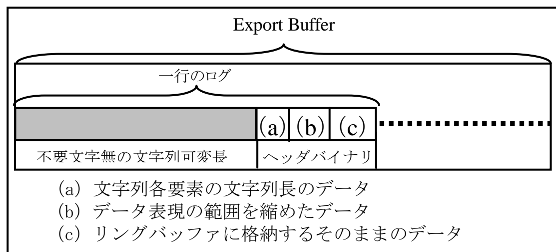


図 4 Export Buffer に格納する BSF Log

3. 実装

3.1. LGP の実装

LGP の実装は KZM－A9 評価ボードのハードウェアの下で Android2.2 が使用する Linux kernel 2.6.29 をトレースが使えるようにパッチする．パッチしたシステムのトレースの部分を本設計に従って LGP に改良する．

3.2. LCP の実装

LCP の実装は Ubuntu10.04 の環境の基にネティブで生成するログ解凍機能およびログ取得機能と Java VM で生成したログ提供機能から実装する．この三つの機能を一つにまとめるために，C 言語を直接に実行させる Java のインタフェースと言われる Java Native Interface(JNI)を使用する．

3.3. システムの実装

LGP と LCP のシステムを実装した上，これらの機能を動作させ，その状態を表示させるために，遠隔ロギング制御が可能な LME を実装した．LME の例は図 5 に示す. Document Object Model (DOM)技術を使用し本システムの LCP のログ提供機能に本システムのため拡張された HTTP Method である GET メソッドを渡し，システムを動作させ，ロギング制御したり，ログを表示したりする．

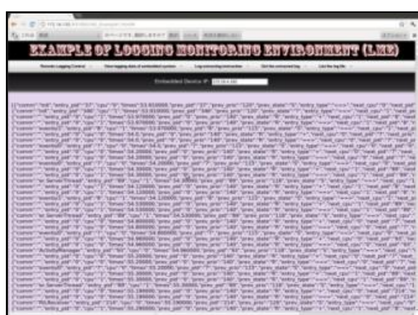


図 5 本システムの遠隔ロギング制御の実装図(LME)

4. システムの評価

4.1. 評価解析

評価する点としては 3 点がある. (1) ユーザ空間に通じないで，直接デバイスに転送するのはどれぐらい効果があるかという評価点, (2) BSF を用いた圧縮機能によりどれぐらいデータの収集効果を上げられるかという評価点, (3)圧縮機能によりどれぐらいオーバヘッドがあるかという評価点である．

(1)はプロセスとソレッドの数とコンテキストスイッチの数をデフォルト状態から測定する．プロセスとソレッド合計数は 352 である．この状態からプロセスを一個ずつ増やし、コンテキストスイッチ数を「/proc/stat」から測定する．測定する環境は圧縮付きトレース，圧縮無し既存のトレース，トレースせずにデフォルトの状態の 3 つである．

(2)と(3)は圧縮機能の評価であり，次の点で測定する．ログデータ収集の精度を表すデータ収集効果式(A)と，データを生成されたデータのサイズ当たりのシステム実行時間を表すデータ生成オーバヘッド式(B)の 2 点である．プログラム実行時間の単位 T は 10ms である．

$$\text{ログデータ収集効果} = \frac{\text{使用ログデータサイズ}}{\text{プログラム実行時間}} \quad \cdots\cdots \quad \text{(A)}$$

$$\text{データ生成オーバヘッド} = \frac{\text{プログラム実行時間}}{\text{生成ログデータサイズ}} \quad \cdots\cdots \quad \text{(B)}$$

4.2. 評価

図 6 は(1)の評価点を示す．このグラフから見るとユーザ空間を通じない圧縮付きトレースはユーザ空間に通じる圧縮無し既存トレースより傾きが 2 倍小さい．この結果はログデータを圧縮付きトレースはユーザ空間に対するコンテキストスイッチの影響をほぼ 2 倍に削減できる．

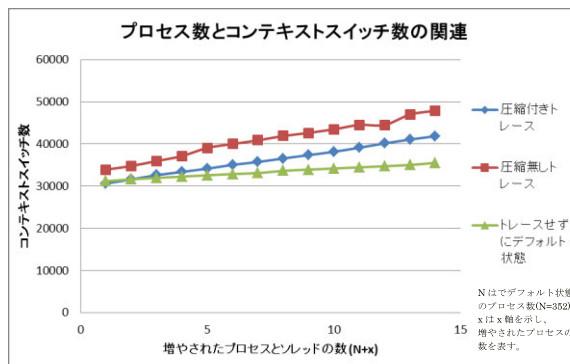


図 6 プロセス数とコンテキストスイッチ数の関連

図 7 と図 8 は(2)と(3)の評価点に示す．今回の評価は従来のトレースと本研究で実装した圧縮付きトレースを評価する．圧縮付きおよび圧縮無しのシステムに対して，Android をランダムな時間に実行し，生成ログデータサイズおよび使用ログデータサイズを取得した．これを元に式(A)と式(B)に従って計算し，グラフにプロッ

トした.

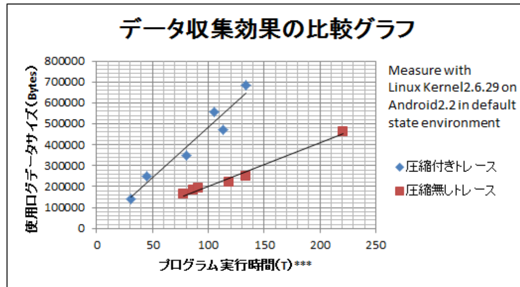


図 7 データ収集効果の比較

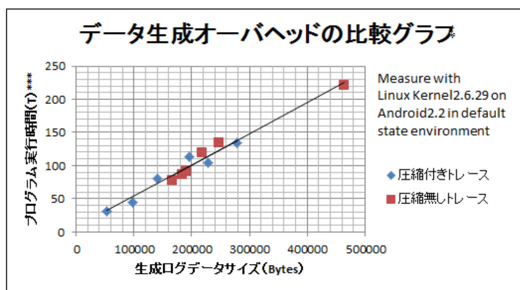


図 8 データ生成オーバーヘッドの比較

図 7 と図 8 のグラフから，圧縮機能付きトレースは圧縮機能なしの従来トレースよりも 2 倍ほどデータ収集効果が高くなったので，システム内に保持されるデータのサイズを減らすことができたことと，オーバヘッドも従来トレースとは変わらないことが明らかになった.

この結果から，圧縮機能付きのトレースはバッファが小さい環境の下であっても，データ収集効果を上げ，データ消失を防ぐことができ，コンテキストスイッチに対する影響を削減することができるということが明らかになった.

5. おわりに

本報告は, Linux の ftrace 機構に圧縮機構を文字列化部の代わりに追加し，ユーザ空間へコピーすることなく，ログデータを外部の装置に転送するトレース機構とその評価そして遠隔ロギング制御のシステムについて述べた.

今後の課題は LGP と LCP の間に通信するプロトコールをより軽いプロトコールを使用する．外部へデータ転送を評価する．LE を複数の組込み機器を並列にアクセスできるように改良して, LE の負荷を評価する.

参考文献

[1] “Android Platform Architecture”, Android, http://www.android.com/media/#platform-architecture#android-30

[2] “Development of Visualization Tool for Trace Log”, Junji GOTO, Shinya HONDA, Takuya NAGAO, Hiroaki TAKADA, IPSJ SIG Technical Report 2009-SLDM-139 & 2009-EMB-12

[3] “Android におけるプロセス可視化環境の開発”, Yuki Nakagawa, Takushoku University, Japan

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

組込みシステムを指向した
分離Linuxプロセスロギング機構

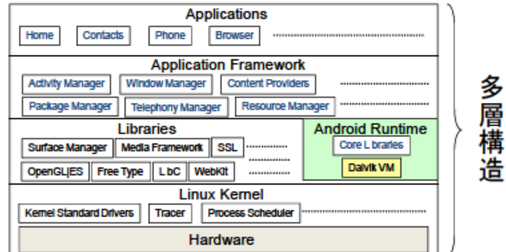
拓殖大学工学部情報工学科†

拓殖大学大学院電子情報工学専攻††

Praween Amontamavut††、中川裕貴††、
早川栄一†

背景

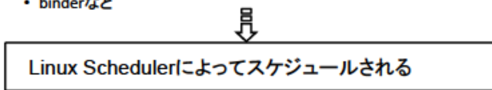
- Linuxカーネルをベースに設計された組込みシステムで用いられているAndroidの登場→ユーザの日常生活に偏るスマートホォンや家電製品など
- LinuxカーネルとAndroidとの関連性



多層構造

背景

- アプリケーション実行速度やHuman Interfaceの応答速度などの向上にはAndroid上で起動するプロセススケジューリングについての工夫
  - Dalvik仮想マシンで起動するプログラムのプロセス
    - Ibinderなど
  - JNIを用いて、LibCをはじめ直接Libraries層を呼び出すプログラムのプロセス
    - binderなど



- 組込みシステムのデバッグにより、Linuxカーネルのスケジューリングの学習や検証が必要

背景

- 組込みシステムのデバッグ
  - 動作の再現が難しい
  - システム環境自体による影響が大きい

アプリケーション及びシステムの開発・学習に応じるプロセススケジューリングの検証が難しい

⇩

実行の動作をロギングし、分析や可視化を行うのが有効な手段

要求

- 組込みシステムで、精度の効果が高いロギング機構が欲しい
  - よりオーバヘッドが低くて、システムにより少なく与えるロギング機構
  - 情報を正しく把握可能なロギング機構
  - 情報を細かく把握可能なロギング機構
- 組込みシステムで、大容量のログデータを保ちたい
  - 可視化などのシステムの
    - 微細なデバッグのため
    - 正確的な学習のため

目的

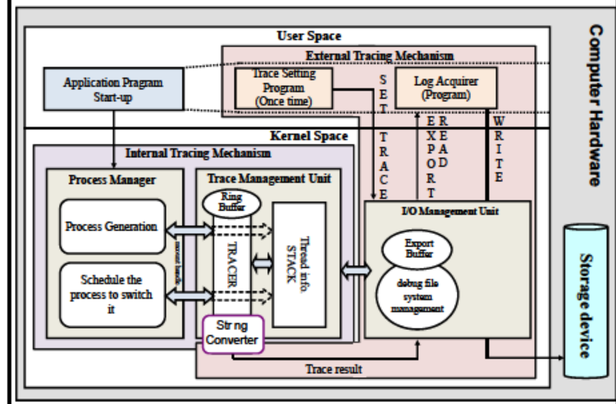
- 組込みシステムでも、精度・効果が高い、大容量のログデータを保てるLinuxロギング機構
  - 設計：既存Linuxのロギング機構を組込みシステムで利用したら発生する問題点を把握し、適用な解決方法を決定する。
  - 実装：KZM-A9評価ボードのハードウェア上に動作するLinux Kernel2.6.29のロギング機構について実装する。
    - 最適化された圧縮付きロギング機能の実装
    - ネットワークでログデータの通信の実装
  - 評価：実装したロギング機能内に発生したオーバヘッドを測定する。

従来のロギング構造

- Linuxプロセスロギングするのに必要なプロセスログ取得機構
  - Linuxカーネルに内蔵されたトレーサのftraceを使用する
  - コンテキストスイッチのログが出力される

```
sh-42   [000] 4154504421 591616   42 120 S   + [000]   42 120 S
sh-42   [000] 4154504421 591616   42 120 S ==> [000]    0 140 R
```

従来トレースの機構



問題

- コンテキストスイッチのログ
  - データを文字列化しエクスポートするので欠点がある
    - エクスポートログ容量は、他のユーザアプリケーション起動しないデフォルト状態で平均*一秒当たりの12833バイト*で、サイズが*大きい*
      - 他のユーザアプリケーションを動作しながらログをとると、これの2~3倍になる
- 組込みシステム
  - 機器内のFlash ROM
    - eMMCの4GBの容量は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

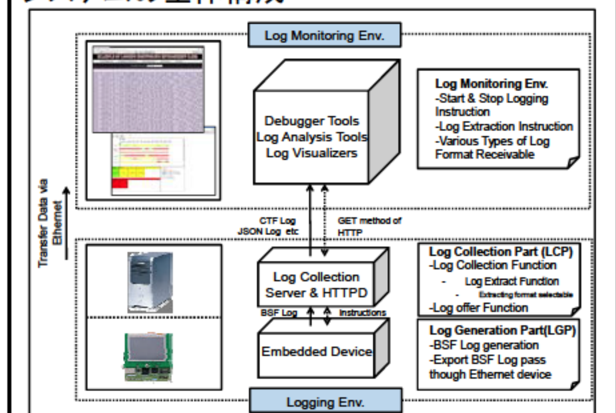
問題に対するシステムへの影響

- 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有限であるため速度差を吸収できない可能性がある
      - データが消失する可能性がある
  - 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

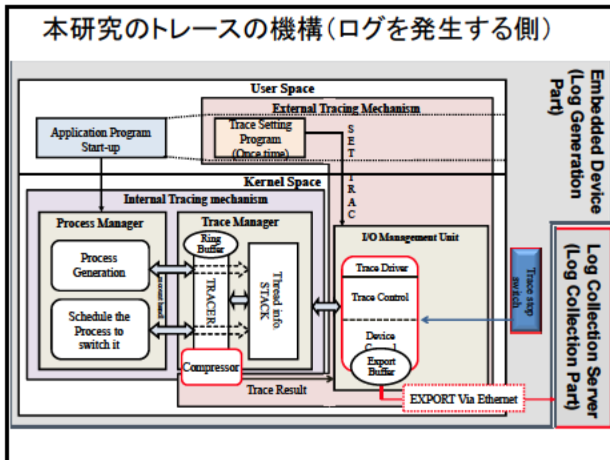
目標

- 機器内のFlash ROMの容量不足問題を回避する
  - 「Logging Monitoring Environment(LME)」+「Logging Environment(LE)」
  - LEは「ログを発生する側」+「ログを取得する側」
    - 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
- エクスポートバッファへのデータ転送速度がリングバッファへのデータ生成・収集速度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
  - エクスポートバッファに情報を多く保持できるように、文字列化部の代わりに*低オーバヘッドの圧縮機能*を追加する

システムの全体構成



本研究のトレースの機構（ログを発生する側）



圧縮機能

- 機能設計
  - 出力したログの形による特徴をベースに注目し、設計する
- 機能の概念
  - 冗長な文字データは出力しない
    - 空白、区別字、改行
  - 二回以上同じデータが繰り返すなら、できれば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

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

コンテキストスイッチの特徴の分類

1)データの範囲も表現も広くて、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

2) [illegible]範囲が狭くて表現が広いデータ:
- 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

3)データ表現が元のデータ範囲に最適となっているデータ:
- そのままバイナリデータを出すれば良い

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	

■データの範囲も表現も広くて、同じデータの繰り返し頻度が高いデータ
◎データの範囲が狭くて表現が広いデータ
●データ表現が元のデータ範囲に最適となっているデータ

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」にした

- データ容量減少程度は同じデータ繰り返しの頻度に正比例する
  ・プロセス名、時間などのデータ
- 従来トレースによるコンテキストスイッチログの特徴は繰り返し頻度が高いため、圧縮性能が高い

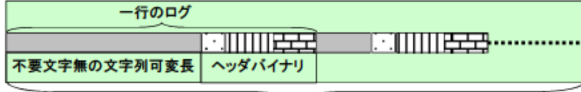
* 文字列の長さをまとめるためのオーバヘッドや、文字列に変換するオーバヘッドなどがある

（2） データの範囲が狭くて表現が広いデータ:
- その範囲に相当する領域にすれば、繰り返す頻度を無視しても良い
  - 文字列変換しなく、低オーバヘッドで通常より少領域でデータを把握できる
    ・前後状態、エントリー種類などのデータ
  - （1）よりオーバヘッドが低いため、その分は吸収できる

（3） データ表現が元のデータ範囲に最適となっているデータ:
- そのままバイナリデータを出すれば良い
  - 文字列変換しなく、（1）より低オーバヘッドでデータを把握できる
    ・プロセスID、前後の優先度などのデータ
  - （1）よりオーバヘッドが低いため、その分は吸収できる

エクスポートバッファに書き込むデータ

- 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、ヘッダを後方にする
  - 文字列の長さをまとめるためのオーバヘッドを減らすため



エクスポートバッファ

- 文字列各要素の文字列長のデータ
- データ表現の範囲を縮めたデータ
- リングバッファに格納するそのままのデータ

- このデータの形式を「mixed Binary and String Format(**BSF**)」と呼ぶ

評価

システムの評価の環境

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 Kernel2.6.29上に動作するAndroid2.2	

評価

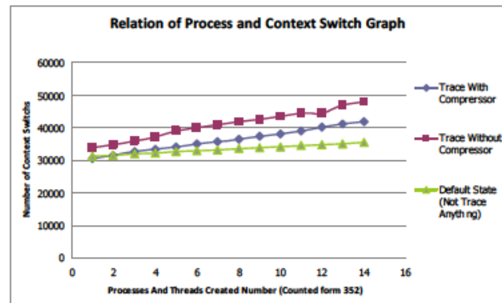
– 使用データ
- 使用ログデータサイズ(Bytes): エクスポートされたデータを実際に使用する全体的なデータのサイズを表す
- 生成ログデータサイズ(Bytes): エクスポートバッファに格納して、エクスポートした全体的なデータのサイズを表す
- プログラム実行時間(T*): リングバッファからエクスポートバッファまでのデータ転送の必要なCPUの時刻の間隔の合計時間を表す

– 圧縮付きと圧縮無しのトレースを評価する
- **データ収集効果** ＝使用ログデータサイズ / プログラム実行時間
  - → 時間当たりにどれぐらい情報を把握できるか
- **データ生成オーバヘッド**＝プログラム実行時間 / 生成ログデータサイズ
  - → データを生成するためにどれぐらい時間がかかるか

– 圧縮率
- **圧縮率**＝1−（生成ログデータサイズ / 使用ログデータサイズ）

(*) jiffies: HZ=100, T = 10 ms

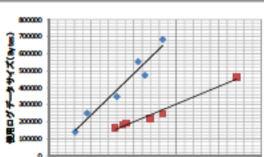
評価



評価

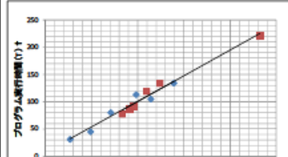
Measure with Linux kernel2 6 29 on android 2 2 in default state environment

◆ 圧縮付きトレース　■ 圧縮無しトレース



データ収集効果の比較

- データ収集効果がほぼ二倍上がる
- 圧縮付きトレースは分散が高いですが、これは第一の特徴で、データ繰り返しの頻度によって変動する



データ生成オーバヘッドの比較

- トレードオフがある特徴を分類した
- 適切にオーバヘッドを吸収できた
  - データ生成オーバヘッドは圧縮付きトレースと圧縮無しトレースは変わらない

関連研究

- Online Log Analysis System for Real-time System's Faults
  - ART-Linuxの使用
  - パフォーマンス、拡張性という要求に対して、ロボットを始め、組込みシステムの故障や障害を検出可能なロギング機構とログ解析機構のプロトタイプの実装
- 本研究
  - ➤ Androidが用いているLinux Kernelの使用
  - ➤ プロセススケジューリングのデバッグが可能で、よりオーバヘッドが低い、情報をより正しく、細かく、多く把握したいという要求に対してのロギング機構のプロトタイプの実装

おわりに

- 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、割込みによりコンテキストスイッチのデータを含めたデータの消失を避けることができる
- システムの環境を分離した
  - ロギングする際にプロセスに影響を与えない
  - 容量が大きい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
- 今後の課題
  - LGPとLCPの間により軽いプロトコールを使用し、より効果なロギング機構を実装
  - 外部へのデータ転送の評価（ネットワークの送信状態）
  - LEのサーバを複数の機器を並列にアクセスできるように改良する

クラスタファイルシステムにおけるデータ配置の効率化

酒井 拓[†]　芝 公仁[†]

[†] 龍谷大学理工学部

1 はじめに

クラスタファイルシステムの特徴として，ノードの追加，削除によって記憶容量を増減できることがある．このとき，ノードの追加．削除に伴い格納ファイルの移動が必要になる．移動するファイルの数が多いほどファイル移動の処理時間が増加し，ファイルにアクセス出来ない時間も増加する．本稿では，ノードの追加，削除時のファイル移動数を減らすための新たなノードへのファイルの振り分け方法について述べる．

2 GlusterFS

本研究では，オープンソースのクラスタファイルシステムである GlusterFS を用いる．GlusterFS は，Red Hat, Inc. が開発しているクラスタファイルシステムであり，brick と呼ばれる各ノードの記憶領域を束ね，1 つのファイルシステムを形成する．GlusterFS には以下の特徴がある．

- 無停止で brick の追加（＝容量拡張)
- ゼロ・シングルポイント障害
- レプリケーション（複製）機能

GlusterFS では，他の多くのクラスタファイルシステムに存在する中央サーバを使用しない．代わりに，ファイル名から 32 ビットのハッシュ値を求め，そのハッシュ値でファイルの管理を行っている．まず，各 brick にハッシュ値の担当範囲を均等に割り当て，その担当範囲に従ってファイルを格納していく．brick の追加，削除を行ったときには，全ての brick で改めて担当範囲を均等に割り当て直す．担当範囲が変更されるため，変更前の格納ファイルを変更後の担当範囲に合わせて移動させる必要がある．この処理をリバランスという．

図 1 のように GlusterFS で使用されている均等割り当てでは，全ての brick の担当範囲が変更されるため，ファイル移動が起こりやすくなる．移動するファイルの数が増加すれば，リバランスに要する時間も増加してしまう．そのため，リバランス時におけるファイル移動を減少させる必要がある．

3 提案手法

リバランス時のファイル移動を最小化するための以下の手法を提案する．

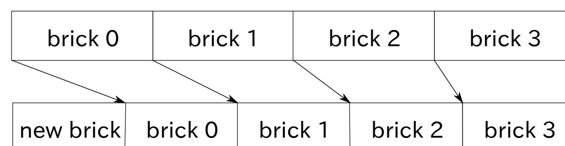


図 1　均等割り当て

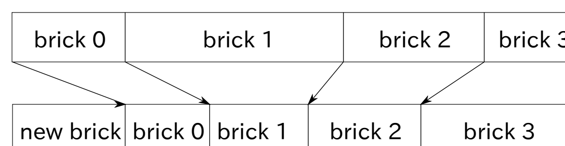


図 2　ランダム割り当て

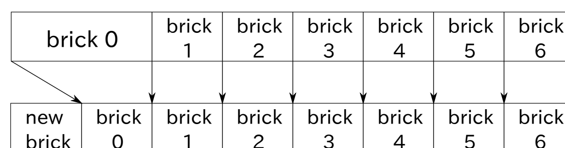


図 3　2 分割割り当て

- ランダム割り当て
- 2 分割割り当て
- ファイル数均等化割り当て
- フリー範囲探索割り当て

ランダム割り当ては，図 2 のように全ての brick の担当範囲の大きさをランダムで設定する．全ての担当範囲の合計がハッシュ値の最大値を越えないようにするために，担当範囲の大きさは 1 から (0xFFFFFFFF ÷ brick の数) とする．最後尾の brick には残りの範囲を全て割り当てる．

2 分割割り当ては，図 3 のように最も担当範囲の広い brick の半分を新しい brick に割り当てる．この方法により 1 個の brick のみが変更となり，他の全ての brick では変更が行われないため，ファイル移動数の削減に繋がると考えられる．

ファイル数均等化割り当ては，図 4 のように各 brick で格納するファイルの数が同じになるように担当範囲を割り当てる．格納ファイルが均一に分けられるため，運用時に各 brick の負荷が均一になると考えられる．

フリー範囲探索割り当ては，図 5 のように各 brick の境界に近いファイル間の範囲が最も広い範囲を新たな brick に割り当てる．格納ファイルの無い範囲を割り当てるため，ファイルの移動が起こらない．

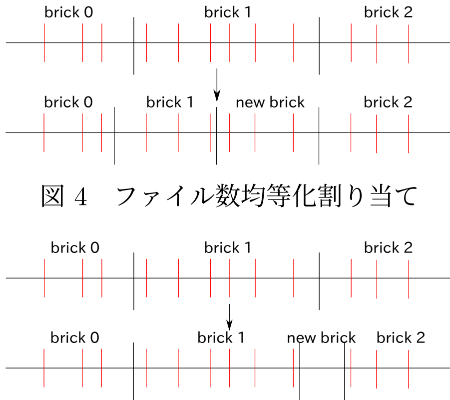


図 4　ファイル数均等化割り当て

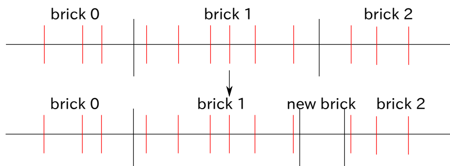


図 5　フリー範囲探索割り当て

4　実験

それぞれの割り当て手法でのリバランス時のファイル移動数，その処理にかかる時間の測定を行う．

以下の条件で実験を行う．

- 単一の計算機で行う
- ランダムなファイル名をつけた空ファイルを 1000 個用いる
- brick を 1 個作成し，その brick に全ファイルを格納する
- brick を 1 個ずつ追加し，追加するごとにリバランスを実行する
- brick が 10 個になるまで行う

以下では，均等割り当て，ランダム割り当ての実験を行う．

4.1　均等割り当て

均等割り当てでのリバランス時のファイル移動数，処理時間を図 6，図 7 に示す．図 6 と図 7 を比較した際，ほぼ同じ形のグラフであると言える．そのため，ファイル移動数と処理時間が密接に関係していることが分かる．

図 6 から brick を追加するごとにファイル移動数が減少していることが分かる．これは本実験ではファイル数を 1000 個に固定しているため，brick を増加すれば brick1 個あたりの格納ファイル数が減少するためと考えられる．また，3 個目の brick を追加した際に全てのファイル移動が行われている．これは brick が 1 個目，2 個目と並んでいるところを，2 個目，3 個目，1 個目というように担当範囲の大きな変更が行われたためである．この点は問題であると考えられる．

4.2　ランダム割り当て

ランダム割り当てでのリバランス時のファイル移動数，処理時間を図 8，図 9 に示す．図 8 からファイル移動数が不規則であることが分かる．brick が 4 個，5 個，9 個のときにはファイルの移動が行われなかったが，これは追加した brick の担当範囲の大きさが 1 であったためである．この場合ファイルの移動自体は起こらなかったものの，その後の運用に問題がある．

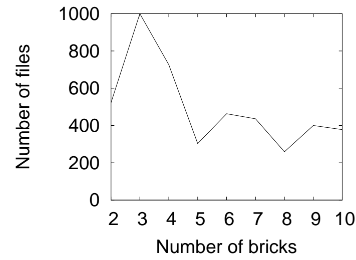


図 6　均等割り当てのファイル移動数

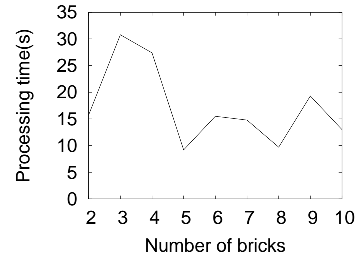


図 7　均等割り当ての処理時間

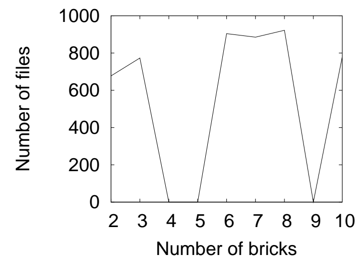


図 8　ランダム割り当てのファイル移動数

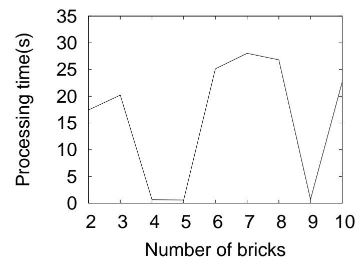


図 9　ランダム割り当ての処理時間

5　おわりに

本稿では，GlusterFS におけるファイル割り当てについて述べ，ファイル移動数を減らすための手法を提案をした．均等割り当てとランダム割り当てのリバランス時のファイル移動数の処理時間について測定した．結果よりファイル移動数の減少が処理時間の短縮に繋がることを確認できた．

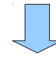
クラスタファイルシステムにおける データ配置の効率化

龍谷大学
理工学部
酒井拓

1

背景

- インターネットの普及により, 様々なサービスを利用
  - クラウド
  - 動画配信
  - etc...
- 技術の進歩によりファイルサイズ, ファイル数の増加

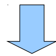


- クラスタファイルシステムの発達

2

目的

- クラスタファイルシステムはノードの追加, 削除が可能
- ノードの変更に伴いファイルの移動が発生



ノード間で起こるファイル移動の最小化

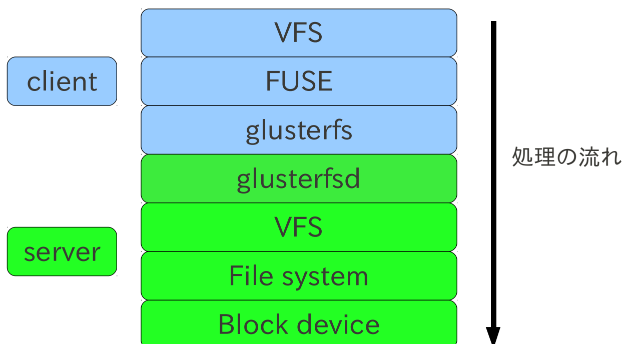
3

GlusterFS

- Red Hat, Inc.の開発しているオープンソースのクラスタファイルシステム
- 無停止でノードの追加（＝容量拡張）が可能
- 多くのクラスタファイルシステムに存在する中央サーバがない（ゼロ・シングルポイント障害）
- 導入，操作が容易

4

階層



5

Elastic Hash Algorithm

- brickと呼ぶノードの記憶領域を束ね1つのファイルシステムを作成
- ディレクトリを除いたベースファイル名から32ビットのハッシュ値を求め, その値により格納するbrickを決定
- brickは以下のように受け持つハッシュ値の範囲を均等に分割

brick 0	brick 1	brick 2	brick 3
0x00000000 〜 0x3FFFFFFF	0x40000000 〜 0x7FFFFFFF	0x80000000 〜 0xAFFFFFFF	0xA0000000 〜 0xFFFFFFFF

6

Elastic Hash Algorithm

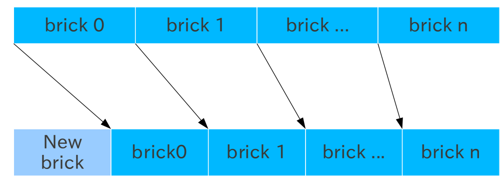
- brickの追加, 削除したとき, 改めて均等に担当範囲を設定
- それに従うようファイルを移動(リバランス)

brick 0	brick 1	brick 2	brick 3
0x00000000 〜 0x3FFFFFFF	0x40000000 〜 0x7FFFFFFF	0x80000000 〜 0xAFFFFFFF	0xA0000000 〜 0xFFFFFFFF

＋ New brick

brick 0	brick 1	brick 2	brick 3	New brick
0x00000000 〜 0x33333333	0x33333334 〜 0x66666666	0x66666667 〜 0x99999999	0x9999999A 〜 0xCCCCCCCC	0xCCCCCCCD 〜 0xFFFFFFFF

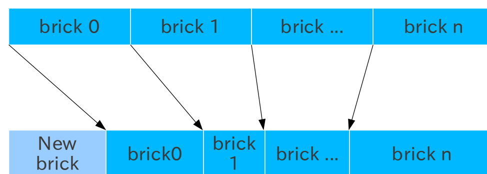
均等割り当て



- 全てのbrickの担当範囲を均一化
- New brickから離れるごとに各brickの担当範囲の変化は減少

8

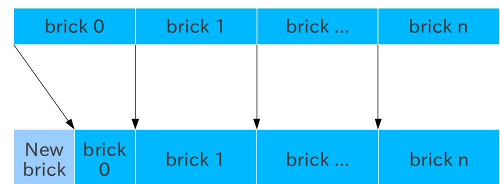
ランダム割り当て



- 担当範囲を個別にランダムで割り当て
- 乱数値は 1 〜（0xFFFFFFFF / brick の数）

9

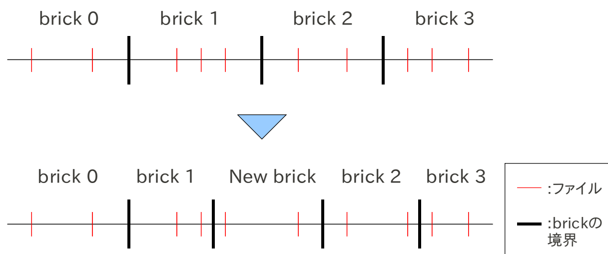
2分割割り当て



- 最も大きな担当範囲を持つbrickの半分を割り当て
- その他のbrickは変わらない

10

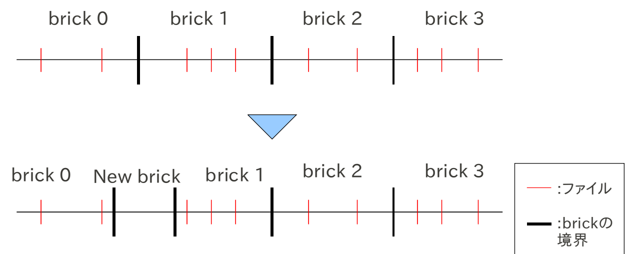
ファイル数均等化割り当て



- 各brickの保持ファイルを均一にするよう全体の担当範囲を振り分ける

11

フリー範囲探索割り当て



- 各brickの境界にあるファイル間の範囲が最も広い範囲にbrickを追加
- その他のbrickは変わらない

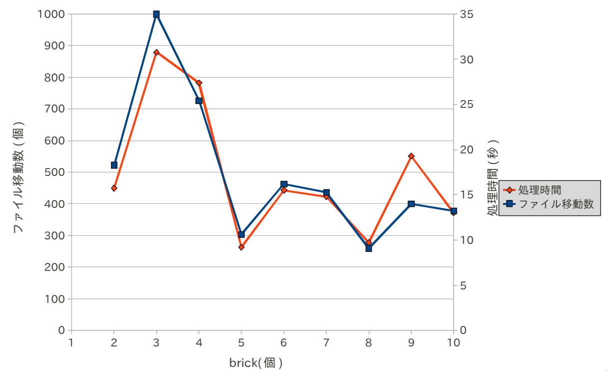
12

実験

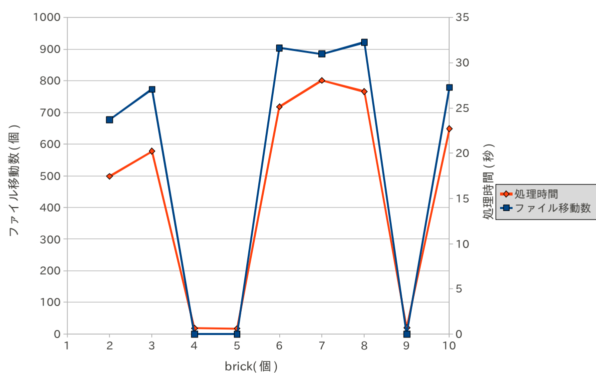
- 均等割り当て，ランダム割り当てでのリバランス時のファイル移動数，それにかかる処理時間の測定
- 単一の計算機で行う
- 1個のbrickにランダムなファイル名をもつファイル1000個を格納
- 1個のbrickを追加するごとにリバランス実行
- brickが最大10個になるまで行う

13

均等割り当て



ランダム割り当て



まとめ

- Glusterfsで起こるbrick間でのファイル移動を減少させるための手法を提案した．
- ファイル移動数と処理時間とのグラフの形が酷似していることにより，ファイル移動数の減少が処理の軽減となることがわかった．

16

クラウドシステムにおける管理支援を目的とした可視化ツールの開発

落合　秀晴[†]　　　早川　栄一[‡]

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[‡]

1.　背景と目的

現在クラウドのサービスが急激に増加している.各企業や組織においてパブリッククラウドサービスとの互換性のある環境の構築を目的とするため,あるいはハードウェアリソースの有効活用などを目的として IaaS 型のプライベートクラウドを導入するケースがある.IaaS 型のプライベートクラウド，または，ハイブリッドクラウドを管理[1]するうえで問題となることは，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，仮想マシンがどの物理マシン上で動作しているかを確認するため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[2]や Ganglia[3]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.

そこで本研究では,上記の問題を解決するため,クラウドシステムを構成する物理マシンと仮想マシンの各リソースの使用状況と対応関係の直観的な把握が可能な可視化ツールの開発を行った.

なお,本研究では IaaS 型のプライベートクラウドの環境として Eucalyptus[4]を用いた.

2.　問題分析と設計方針

2.1　問題分析

本研究で使用している Eucalyptus によるプライベートクラウドの構成は，仮想マシンを稼働させる計算ノードとしての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フロントエンドとしての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 種類のコンポーネントで構成されている.

ここで問題となるのは,上記コンポーネントの対応関係と NC と仮想マシンとの対応関係の把握とシステム内のマシンそれぞれの状態の把握である.従来の MRTG 等の監視ツールでは物理マシンを対象とした監視を行うため仮想マシンの状態とそれを稼働させている物理マシンとの対応関係が把握できない.また画面構成がシステム全体のリソース表示か各マシンのリソースの状態を表示する構成となっているため,それぞれのマシンの状態を把握するときには手間がかかる.

そこで本研究では上記のコンポーネント及び仮想マシンのリソースの使用状況と各コンポーネントと仮想マシンの対応関係について直観的な把握が可能な可視化画面を開発することで問題の解決を図る.

またプライベートクラウドが各企業や組織などの限られたリソースによって構築されるものとした場合,規模としては 1000 台未満が妥当と考え,この規模を想定して可視化画面の開発を行った.

2.2　可視化方法

本研究における直観的な可視化の手法として,クラウドシステムを構築しているマシンの状態をコンポーネント単位で把握可能な状態グラフによる可視化と,仮想マシンと物理マシンとの対応関係をツリー状に表示する関係グラフによる方法を提案するものとした.

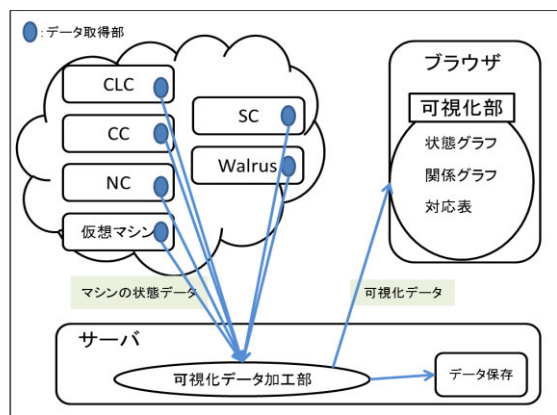


図 1. 全体構成

状態グラフの表示方法は棒グラフを用いて縦軸に台数横軸に使用率を取ることで複数台の状態を可視化することとした.

関係グラフの表示方法はツリーグラフによるものとした.CLC をルートノードとし,その下の階層に CC,SC,Warlus,そのの下の階層に NC,その下に仮想マシンとしたツリー状のグラフ表示で対応関係を可視化する.これによりどの NC 上で何の仮想マシンが動作しているかが視覚的に把握できるようになると考えた.

2.3　可視化情報

可視化するために各物理マシンと仮想マシンから取得する情報は,マシンの識別情報として IP アドレス,ドメイン名,所属クラスタの情報を利用することとした(仮想マシンに関してはこの他に Eucalyptus から割り振られている ID,動作状態,自身を生成している NC の情報も加える).また,マシンの状態の情報として,マシンごとの各リソース(CPU・メモリ・ディスクなど)の使用状況,ネットワークの状態などの情報を可視化に利用することとした.

2.4　ユーザによるマシンの状態通知設定

プライベートクラウドの利用目的ごとにマシンのリソースの使用状態が違うと考えられる.そこで,マシンのリソースの状態,ネットワークの利用状態から正常な状態の範囲をユーザが定義し,それをもとに正常,異常の状態を通知するものとした.

3.　設計

3.1　全体構成

図 3 に全体構成を示す.

構成はクラウドの各コンポーネントからマシンの情報とマシンの状態情報を取得するデータ取得部,取得したデータを可視化情報に変換し,保存するデータ加工部,可視化情報をブラウザ上に表示するための可視化部により構成するものとした.

3.2　データ取得部

データ取得部から CPU，メモリ，データを取得した時間,IP などのマシン情報と状態の情報を取得し，取得し

たデータをデータ加工部に一定時間ごとに送る.

3.3　データ加工部

データ加工部は Eucalyptus 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.

3.4　可視化部

可視化画面の構成はトップページに全体の状態グラフ,各コンポーネントの詳細ページにコンポーネントごとの状態グラフ,関係グラフのページに関係グラフ,それに加えて関係グラフだけでは表現しきれないそれぞれのマシンの情報を表示するフォームとしての対応表により構成するものとした.

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，ブラウザ上に状態グラフや関係グラフの描画を行うものとした.

また,マシンの正常状態をユーザが任意に決定できるようにフォームを設定する.このフォームからユーザは任意のリソース負荷に応じたマシンの状態を設定が可能にする.この正常状態を超えたマシンが存在する場合,状態グラフで該当する部分を異常として赤で表示する.

状態グラフにおける JSON データの参照は JavaScript の Ajax 機能を用いることにした.

4.　実装

4.1 データ取得部

データ取得部及びデータ加工部の実装には EucalyptusAPI が利用可能なライブラリが豊富な Python を用いた.

データ取得部はエージェントとしてすべての可視化対象とするコンポーネントに設置する.今回実装したデータ取得部が取得するデータは次のとおりである.

・ドメイン名
・IP アドレス
・リソースの使用状況のデータ（CPU,メモリ,ネットワークトラフィック,ディスク)
・取得時間

各コンポーネントの各マシンに設置したデータ取得用

図 2.状態グラフの画面構成

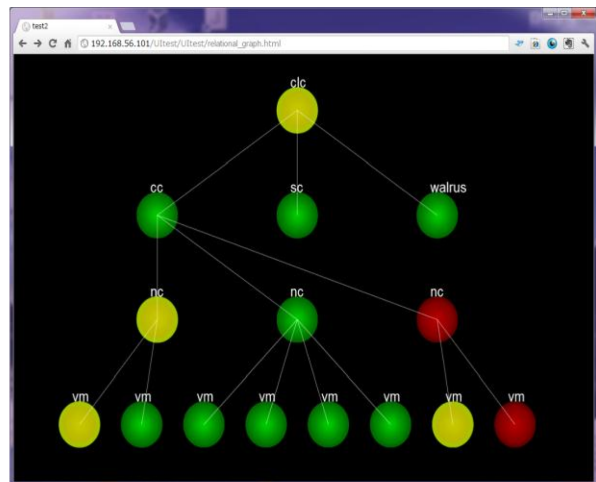


図 3.関係グラフの画面構成

のスクリプトから一定時間ごとに Linux の/proc ファイルからマシンのリソースに関するデータを取得し,データ加工部に送信する.

4.2 データ加工部

データ加工部は EucalyptusAPI から取得したデータを元に可視化データの配列を生成する.EucalyptusAPI から取得するデータは次のとおりである.

・コンポーネントと仮想マシンの IP アドレス
・それぞれの所属しているクラスタ名
・状態(仮想マシン)
・ID(仮想マシン)
・所属している計算ノード名(仮想マシン)

これらの取得した情報を元にデータ取得部から送られてきた各マシンの状態データを可視化データの配列に挿入し JSON データとしてまとめる.

4.3 可視化部

図 2 に状態グラフの画面構成を,図 3 に関係グラフの画面構成を示す.可視化部の状態グラフ及び関係グラフの実装は JavaScript 及び HTML5 の canvas を用いた.

5.　まとめ

クラウドシステムの管理支援を目的とした，状態グラフを用いたサーバの状態把握と，関係グラフ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. これによりクラウドのコンポーネントの対応関係がより直観的に把握することが可能となった.

今後の課題はデータ加工部の保存機能の実装及びツールの評価実験である.

参　考　文　献

[1]　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
[2]　MRTG.jp:http://www.mrtg.jp/
[3]　Ganglia Monitoring System: http://ganglia.sourceforge.net/
[4]　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

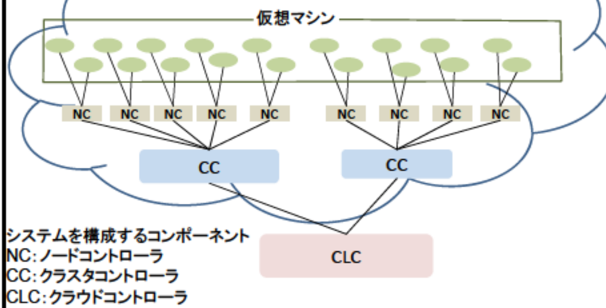
クラウドシステムにおける管理支援を目的とした可視化ツールの開発

†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

プライベートクラウドの構成



背景(2)

- 管理が煩雑
  - 物理マシンと仮想マシンの稼働状態の把握
  - 仮想マシンを生成している物理マシンの特定が必要

・スケールアウトによりサーバが増加
・仮想化により仮想マシンが増加

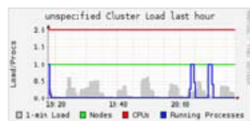
問題点

- 負荷のばらつきの把握しづらい
- マシンの特定に手間がかかる

GangliaやMRTGなどのRRDtoolを用いた表示形式

・システム全体のリソースをひとまとめにした表示形式
・単一のマシンだけの表示形式

・CPU



・メモリ

問題点(2)

- 対応関係の把握の手間
  - 物理マシン間の対応関係の把握
  - 物理マシンと仮想マシンの対応関係の把握

CUIからコマンドの入力が必要

複数のマシンの関係を把握する場合には手間がかかる

目的

- 管理者がシステムの監視する際の手間を軽減
  - サーバ稼働状態を可視化することで直観的に把握
  - 個別のマシンの素早い特定
  - 仮想マシンと物理マシンの関係を視覚的に把握

可視化ツールの特徴

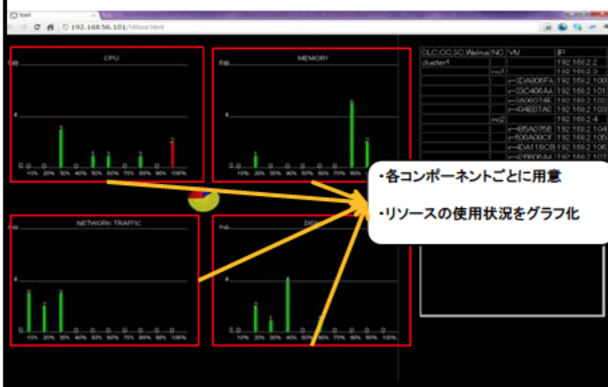
- 状態グラフ
  - システム内のリソースの使用状況を表すグラフ

各コンポーネントのリソースを使用率ごとに棒グラフ化

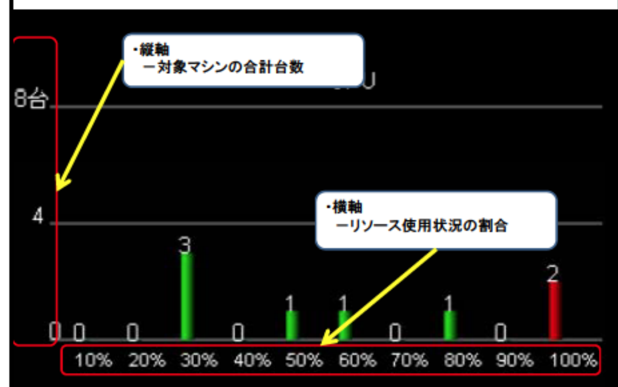
↓

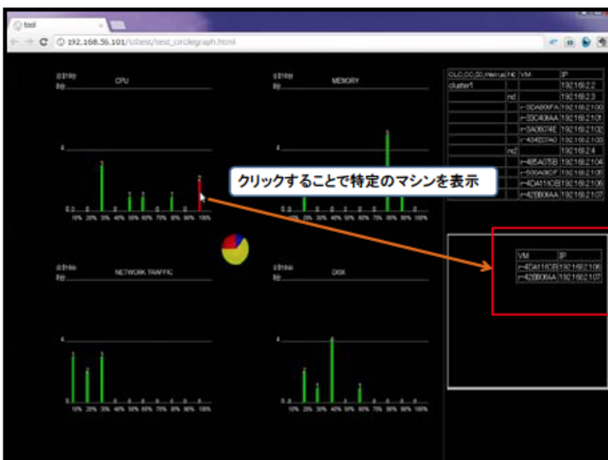
従来のツールではわかりにくかった負荷のばらつきが見える

状態グラフ



状態グラフ(2)





可視化ツールの特徴(2)

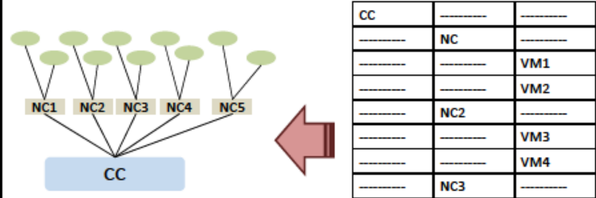
- 関係グラフ
  - 各コンポーネントや仮想マシンの関連を表すグラフ

各コンポーネントマシンをツリーグラフで表示

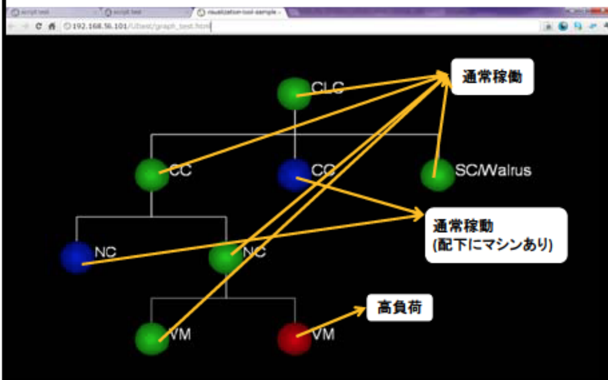
コマンドラインから把握しづらかったコンポーネント間の対応
関係を視覚化

関係グラフ

- 物理マシン,仮想マシンの対応
  - 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

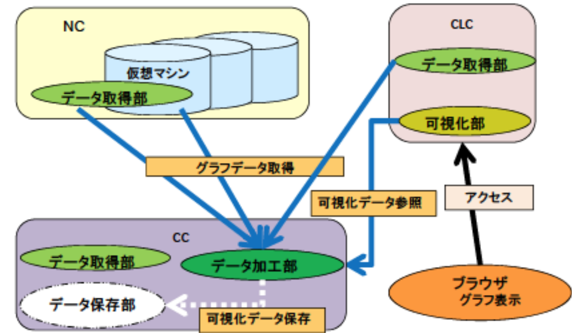


関係グラフ



構成

例)データ取得部を全体に,データ加工部をCCに,可視化部をCLCのマシンに設置した場合の構成



状態グラフに用いるデータ

- 可視化対象から取得するリソース情報
  - CPU負荷
  - メモリ負荷
  - ネットワーク負荷
  - ディスクの空き容量

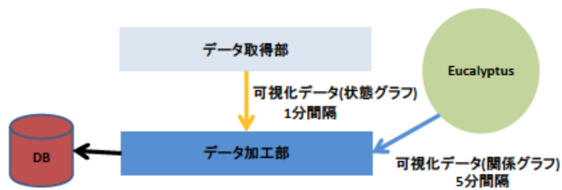
- 一定時間ごとにリソース情報を収集

関係グラフに用いるデータ

- EucalyptusAPIから取得する情報
  - マシン間の関係に関する情報
    - 名前
    - IP
    - ID
    - マシンの役割(CLC,CCなど)
    - 所属しているクラスタの名前

実装

- データ取得部・データ加工部
  - 通信用のクライアント及びサーバ
    - Python



実装(2)

- 可視化部
  - 各グラフの描画
    - HTML
    - canvas
    - JavaScript
  - グラフの更新
    - jQuery
      - Ajax
- Webサーバ・DB関連
  - Django

プライベートクラウド環境

- サーバ
  - DELL PowerEdgeR410
- OS
  - CentOS5.6
- クラウド構築ソフト
  - Eucalyptus2.0
- 仮想マシンマネージャ
  - Xen

まとめ

- 研究の成果
  - 棒グラフを用いた負荷のばらつきの可視化
  - 関係をグラフ化することによる仮想マシンと対応したサーバの把握
- 今後の課題
  - 過去状態の可視化方法の検討
  - より直感的な関係性の把握手法の考案

組込み向けマルチコアアクセラレータ用 OpenCL ライブラリと組込み OS の設計

坂本 龍一†

東京農工大学大学院†

1 はじめに

近年スマートフォンやタブレッドの普及により組込み向け SoC の高性能化，省電力化が大きな課題となっている．これらの SoC は汎用プロセッサと演算処理に特化したアクセラレータから構成されている．我々もこれらの問題を解決するために，組込み向けのマルチコアアクセラレータ Cube の研究，開発を行っている．Cube プロセッサは汎用の Geyser-Cube と CMA-Cube から構成されている．しかし，これらのプロセッサを有効に利用するためには，煩雑な並列プログラミングや，効率よくアクセラレータを制御する必要がある．そこで本研究では，マルチコアアクセラレータ Cube を対象として OpenCL ライブラリの提供を行い，CMA-Cube アクセラレータの隠蔽を行う．さらに，これらの OpenCL ライブラリと協調する CubeOS を提案する．これらによって，効率よい実行環境の提供を目指す．

2 Cube プロセッサ

Cube プロセッサは汎用プロセッサチップ(Geyser-Cube)と，アクセラレータチップ(CMA-Cube)から構成されるマルチコアアクセラレータである．これらのチップは 3 次元方向に積層される構造となっている．CMA-Cube は演算器アレイベースの超省電力な再構成可能なアクセラレータである．図 1 に Cube プロセッサの概要を示す．

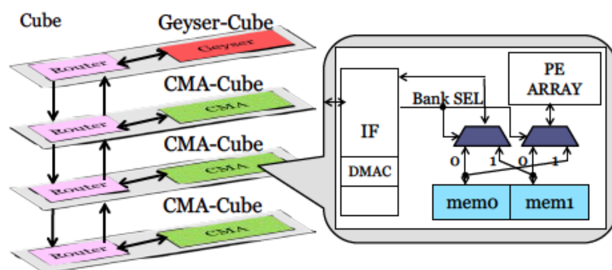


図 1 Cube プロセッサ

Cube プロセッサは，積層する CMA-Cube の枚数を増やすことで性能とコストのトレードオフに対応できる特徴を持っている．コストが優先される場合は CMA-Cube を 1 枚とし，性能が優先される場合は CMA-Cube の枚数を増やすことで対応が可能である．また，Cube プロセッサはパイプライン処理において特に高い性能を得ることができるように設計され，CMA-Cube 間でデータフローを行う機構が備わっている．具体的には，CMA-Cube には DMAC，ダブルバッファを用いたローカルメモリがある．これらを利用してパイプライン処理を行うことで CMA-Cube を効率よく利用できる．

Designing OpenCL library and embedded OS for multi-core accelerator.
†Ryuichi SAKAMOTO
Tokyo University of Agriculture and Technology

3 問題点

Cube プロセッサはマルチメディア処理に代表されるパイプライン処理を効率よく実行可能である．CMA-Cube にパイプラインを構成する処理を割り当て，演算データを CMA-Cube 間でコピーすることで効率よい実行が可能である．図 2 に 2 つの CMA-Cube を用いたパイプライン処理の例を示す．ここでは，入力のデータに対して，step0, step1 を行う例を示す．

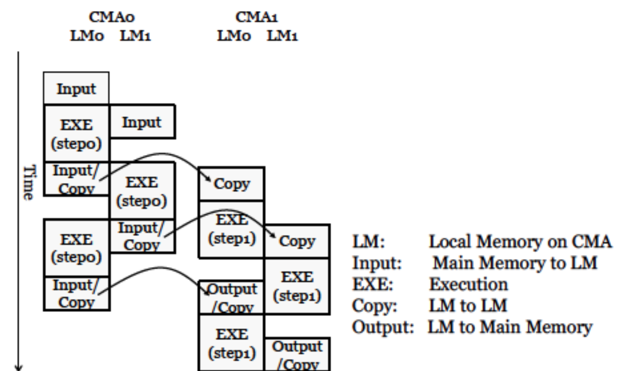


図 2 パイプライン処理の例

図 2 のように CMA0 に step0 を，CMA1 に step1 を割り当て，step0 の演算を行った結果を CAM1 に転送することでパイプライン処理を行うことができる．また，CMA-Cube 内部のダブルバッファリングの機能を利用することでデータ転送を，演算時間中に隠蔽することが可能となり，CMA-Cube の利用効率を上げることが可能である．しかし，主記憶から CMA-Cube のローカルメモリへデータコピー，CMA-Cube 間でのデータコピーのための DMAC 設定，ダブルバッファの制御，CMA-

Cube の演算開始の指示を Geyser-Cube から制御する必要がある．また，これらの処理は CMA-Cube ごとに並列に動作し，同期処理も，Geyser-Cube で管理する必要がある．これらのパイプライン制御はプログラマが指示する必要がある．しかし，これらの制御は，非常に煩雑のものとなる．この場合，平行に動作するスレッドを CMA-Cube の数分用意し，それらのスレッド間にで，同期をとりながら動作させる必要がある．また，積層する CMA-Cube 枚数を増やすことで性能を上げることが可能となる特徴を持つが，制御スレッドの数も増え，スレッド間での同期の管理オーバヘッドが増大することが考えられる．これらのオーバヘッドは CMA-Cube のスケールアウトによる性能向上を阻害してしまう．

4 研究目標

Cube プロセッサはパイプライン処理を行うことで高い演算性能を得ることが可能である．しかし，CMA-Cube の制御が複雑である課題がある．そこで，本研究では CMA-Cube 向けの OpenCL 環境を提供する．具体的には，OpenCL ライブラリで，CMA-Cube へのデータ転送，CMA-Cube 間の同期処理，非同期処理を隠蔽する．これによって，複雑な CMA-Cube の制御を OpenCL API から制御できるようにする．また，CMA-Cube の積層の枚数による計算性能のスケールアウトに対応するシステムソフトウェアが必要となる．そこで，OpenCL ライブラリと協調する CubeOS の提供を行う．

5 システム構成

本研究では，効率よい CMA-Cube 利用のために OpenCL ライブラリと CubeOS によってこれらを実現する．図 3 に全体構成を示す．

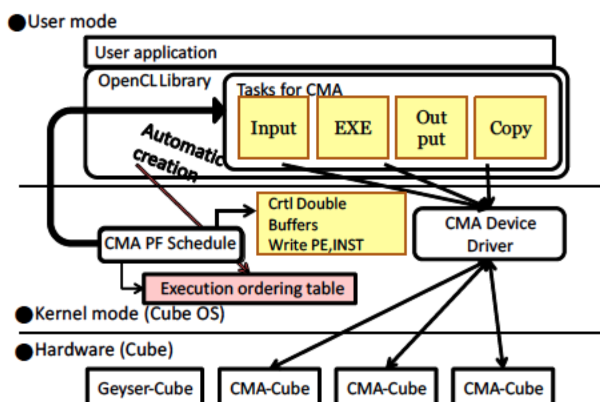


図 3　全体構成

5.1 CMA-Primitive Function (CMA-PF)

CMA-Cube のパイプライン処理の実行単位を CMA-PF として定義する．CMA-PF は Input，EXE，Output，Copy，Ctrl Double Buffer，Write PE，Write INST からなる．表 1 に CMA-PF の役割を示す．

表 1　CMA-PF が行う CMA-Cube の操作

CMA-PF 名	CMA-PF が行う CMA-Cube 操作
Input	主記憶から CMA-Cube のローカルメモリへデータのコピー
Output	CMA-Cube のローカルメモリから主記憶へデータのコピー
Copy	CMA-Cube のローカルメモリ間でのデータのコピー
EXE	CMA-Cube での演算の開始を制御
Ctrl Double Buffer	CMA-Cube のダブルバッファの制御
Write PE	CMA-Cube の演算器アレイの構成情報の書き込み
Write INST	CMA-Cube のマイクロコントローラの命令メモリの書き込み

CMA-PF で CMA-Cube 独自の機能を隠蔽する．これらの CMA-PF のうちのブロッキングを含む，Input，EXE，Output，Copy の CMA-PF は CubeOS が提供するタスクとして実行することで，並列に実行できるようにする．このように CMA-PF を定義することで，CMA-PF 間での同期を OS が管理できるようになり，パイプライン処理を行うことができる．

5.2 OpenCL ライブラリ

OpenCL ライブラリは CMA-Cube で行われるパイプライン処理のデータ転送，演算の実行，ダブルバッファ処理を OpenCL API に隠蔽する．具体的には OpenCL API 中に複数の CMA-PF を隠蔽する．また，OS との連携のために，ライブラリ中で CMA PF の実行順序を CubeOS へ通知する．

5.3 CubeOS

CubeOS では，CMA-Cube でパイプライン処理を行うために，CMA-PF を制御する．また，CMA-Cube スケールアウトに伴う，スレッドの制御オーバヘッドの削減を行うために，CMA-PF 間の同期情報，各 CMA-PF の実行用のアドレスなどのパラメータを CubeOS 内部に保持する．このようにして，CMA-PF の同期処理を OS 内部で行うことで，パイプライン処理における同期処理オーバヘッドを削減する．CubeOS と OpenCL ライブラリの連携について図 4 に示す．CubeOS は OpenCL ライブラリから CMA-PF の実行のために情報をもらい OS 内部で管理を行い，それによって CMA-PF を実行する．このようにすることで，同期を OS 内部で処理することができ，オーバヘッドを削減できる．

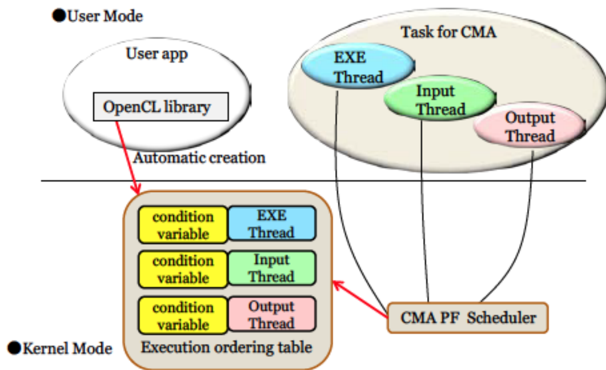


図 4 OpenCL ライブラリと CubeOS の連携

5.4 無閉路有向グラフ(DAG)による CMA PF 同期管理

CubeOS は CMA PF の同期の管理を行い，パイプライン処理を実現する．その際，CMA PF の同期の管理を DAG を用いて管理する．図 5 に DAG グラフの例を示す．

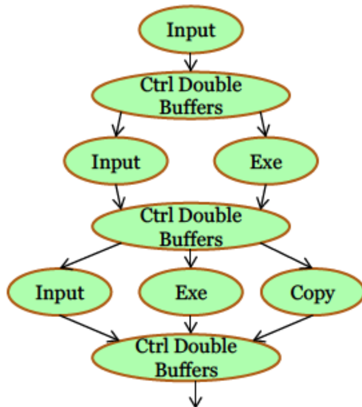


図 5 DAG を用いた CMA PF 間の同期管理

ノードは CMA PF を表し，エッジは依存関係のあるデータフローを示す．本 DAG グラフは OpenCL ライブラリによって生成され，CubeOS が参照し CMA PF の実行順序を制御する．

5.5 OpenCL API と連携用システムコール

OpenCL API では CMA-Cube での処理を抽象化してユーザに提供を行う．表 2 に Cube 用 OpenCL API を示す．

表 2 OpenCL API と CMA-Cube 操作

OpenCL API	CMA-Cube での操作
clEnqueueWriteBuffer	主記憶からローカルメモリへのデータのコピー
clEnqueueReadBuffer	ローカルメモリから主記憶へのデータのコピー
clEnqueueCopyBuffer	ローカルメモリ間でのデータのコピー
clEnqueueTask	CMA-Cube の実行

これらの OpenCL API 中では，CMA-Cube の操作を抽象化した CMA-PF の実行要求を CubeOS に対して行う．これらは，次の表 3 に示す連携用システムコールを用いて要求する．

表 3 連携用システムコール

システムコール	機能
cam_task_create	CMA-PF の Input, EXE, Output, Copy を実行するタスクの生成
add_order_table	CMA-PF の追加を OS に依頼

cma_task_create システムコールでは，CMA-PF を実行するためのタスクの生成を CubeOS に対して行う．必要数の CMA-Cube を指定することで，タスクにかかるオーバヘッドとメモリ使用量の削減を行う．また，add_order_table システムコールは，CubeOS に対して CMA-PF の追加を指示する．

6 関連研究

Jaejin Lee らの研究[1]では，ローカルメモリを持つ Cell プロセッサを対象とした OpenCL 実装を行っている．彼らは POSIX スレッドを利用し，ユーザープログラム側で同期制御を行っているのに対して，本研究では OS 内部で，同期処理を完結させるようにしている．また，Juan らの研究では，リコンフィギャラブルプロセッサを対象とした OS の研究[2]を行っている．彼らは，リコンフィギャラブルデバイスを操作するのに独自の API を利用しているのに対し，本研究では OpenCL API を利用してアクセラレータの制御を行っている．

7 おわりに

本研究では組込み向けのマルチコアアクセラレータ用の OpenCL ランタイムライブラリについて示した．OpenCL によってアクセラレータの操作を隠蔽し，ライブラリと OS が連携し軽量にアクセラレータの制御を行う方法を示した．今後はライブラリ，OS の実装を行い，評価を行う．

参 考 文 献

[1] Jaejin Lee, Jungwon Kim, Sangmin Seo, Seungkyun Kim, Jung-Ho Park, Honggyu Kim, Thanh Tuan Dao, Yongjin Cho, Sung Jong Seo, Seung Hak Lee, Seung Mo Cho, Hyo Jung Song, Sang-Bum Suh, and Jong-Deok Choi, An opencl framework for heterogeneous multicores with local memory. In Valentina Salapura, Michael Gschwind, and Jens Knoop, editors, PACT, pp. 193-204. ACM, 2010.

[2] Juan Antonio Clemente, Carlos Gonzalez, Javier Resano, and Daniel Mozos, A task graph execution manager for reconfigurable multi-tasking systems. Microprocess. Microsyst., Vol. 34, No.2-4, pp. 73-83, March 2010.

1

組込み向けマルチコアアクセラレータ用 OpenCLライブラリと組込みOSの設計

2012/09/10
並木研究室
D1　坂本　龍一

2

目次

1. 研究背景
2. 組込み向けマルチコアアクセラレータ(Cube)
3. 課題
4. 目標
5. 全体構成
6. OpenCL API
7. 設計
8. まとめ

3

研究背景

- スマートフォン，タブレットの普及により組込み向けSoCの高性能化，省電力化が課題
- マルチコアアクセラレータの普及
  - 汎用プロセッサと複数のアクセラレータから構成
- ◆Cubeプロセッサ
  - 汎用のGeyser-Cubeプロセッサと3つのCMA-Cubeアクセラレータ

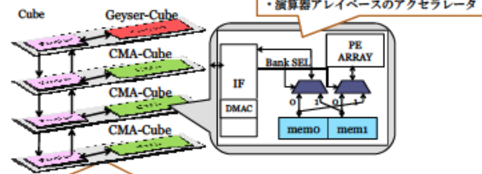
Cube用OpenCL環境
- OpenCLによってCMA-Cubeを隠蔽
- OpenCLライブラリとCubeOSは協調動作し，同期制御，データ転送を軽量に実行

4

Cubeプロセッサ

- 汎用プロセッサチップGeyser-CubeとアクセラレータチップCMA-Cubeから構成
- チップは3次元方向に積層
  - 積層するCMA-Cubeチップの枚数を変更可能
  →演算性能，コストのトレードオフに柔軟に対応

・超省電力再構成可能なデバイス
・演算器アレイベースのアクセラレータ



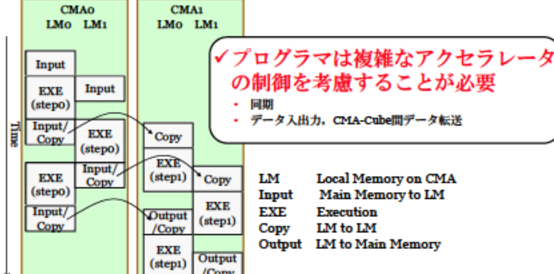
チップはリングバスにて接続され，CMA-Cube間で効率よくデータ転送を行えるようにダブルバッファを搭載

→データフローによるパイプライン処理

5

パイプライン処理記述の課題

2つのCMA-Cubeを用いたパイプライン処理の例
(Input data -> step0 -> step1 -> output)



✓プログラマは複雑なアクセラレータの制御を考慮することが必要
- 同期
- データ入出力，CMA-Cube間データ転送

LM	Local Memory on CMA
Input	Main Memory to LM
EXE	Execution
Copy	LM to LM
Output	LM to Main Memory

- Data copy are abstracted using double buffers.
  - Two LMs in a CMA-Cube for pipelining.
  - One LM is copied data, and the other LM is used by calculation.

6

CMA-Cubeスケールアウトによるパイプライン制御オーバヘッド増加の課題

- ➢ CMA-Cubeの実行制御
  - CMA-Cubeごとに並行して動作するRead, Write, Exeスレッドを生成
  - 複数スレッド間で条件同期を行いパイプライン処理
- ➢ CMA-Cubeのスケールアウト
  - 性能，コストに応じてCMA-Cubeの数を増大

✓CMA-Cubeの利用効率の低下
パイプライン制御オーバーヘッド増加
➢ 制御スレッドの増加に伴う条件同期コストの増加

研究目標

効率よいCMA-Cubeのパイプライン処理を目指し，CMA-Cube用OpenCLの実行環境の提供

✓OpenCL ライブラリ
- パイプライン処理が複雑
- 効率的にパイプラインをプログラミングする環境が必要
  →**OpenCLライブラリでCMA-Cube間の同期，非同期処理を隠蔽**
  プログラマはOpenCL APIを用いて簡単にCMA-Cubeを制御

✓Cube OS
- CMA-Cubeに対する非同期な処理，データ転送を軽量に実行
- CMA-Cubeチップの増加に対する制御コスト増大に対応
  →**CubeOSはOpenCLライブラリと協調しCMA-Cubeを効率的に制御**

全体構成

●User mode
User application
OpenCL Library
Tasks for CMA: Input, EXE, Out put, Copy
Automatic creation
CMA PF Schedule
Ctrl Double Buffers, Write PE/INST
CMA Device Driver
Execution ordering table
●Kernel mode (Cube OS)
●Hardware (Cube)
Geyser-Cube, CMA-Cube, CMA-Cube, CMA-Cube

CMA Primitive Function (CMA PF): Input, EXE, Output, Copy, Crtl Double Buffer, Write PE, Write INST
パイプライン処理の実行単位

✓OpenCL ライブラリ
- OpenCL APIを用いてCMA PFを隠蔽
- CMA PFの実行順序情報をCubeOSへ通知
- 自動的にexecution order tableとCMAタスクを自動生成

✓Cube OS
- CMA-Cubeでパイプライン処理ができるようにCMA PFを制御
- CMA PFの実行情報をOS内部にexecution order tableとして保存

CubeOSとOpenCLライブラリの協調動作

基本アイディア
- OS内部で実行順序を決定
  - ライブラリと協調し制御

→スレッド間の条件同期オーバーヘッドを減らしアクセラレータの利用効率を向上

●User Mode
User app
OpenCL library
Automatic creation
Task for CMA: EXE Thread, Input Thread, Output Thread
●Kernel Mode
condition variable / EXE Thread
condition variable / Input Thread
condition variable / Output Thread
Execution ordering table
CMA PF Scheduler

CMA PF専用スケジューラ
→CMA PFの実行順序制御を行いパイプライン処理

- OSはスレッド(タスク)を直接制御
- OpenCLライブラリは実行順序を通知

✓低レベルで煩雑な同期をOSが行うことでユーザアプリに対して同期処理を隠蔽
✓条件同期オーバヘッドを削減しCMA-Cubeのスケールアウトに対応

CMA PFを用いたパイプライン処理

✓CMA PFはCMA-Cubeのパイプライン処理の実行単位.
- CMA PFは並行して動作
- CMA PF間の同期はOSで管理

OSがCMA PFの操作を行うことでパイプライン処理を実行可能に

→OSが効率よくCMA PFを制御することでCMA-Cubeの実行効率を上げることが可能

CMA0 LM0 LM1 / CMA1 LM0 LM1
Time
Kernel mode / User Mode Task
Input, CMA Scheduler, Ctrl Double Buffers
EXE (step0), Input
Input/Copy, EXE (step0), Copy
EXE (step0), Input/Copy, EXE (step1), Copy
Input/Copy, Output/Copy, EXE (step1)
EXE (step1), Output/Copy

入出力，実行のブロッキングの隠蔽

- CMA PFは2つの種類に分類可能
  - ブロッキングするもの
  - ブロッキングしないもの
- ✓ブロッキングを伴うものは並列実行を可能とするために非同期に実行
  - OSが提供するタスクの中で実行

CMA primitive function	Execution type
• Ctrl Double Buffers : change Bank SEL	Execute in Kernel • parallel • not take time
• Write PE : write pe const mem • Write INST : write CMA instruction mem	Execute in Kernel • not parallel
• Input : write local memory • Output : read local memory • Copy : copy between local memory • EXE : execute cma	User Mode Task • parallel • take time

✓CMA PFスケジューラ内部で実行

✓OSが提供するタスクの中で実行
✓タスクのでブロッキングの処理を実行

入出力，実行のブロッキング処理をCMA PFの中に隠蔽
- OpenCL APIではブロッキング処理の考慮が不要
- CMA PFの同期を無閉路有向グラフ(DAG)として管理可能

CMA PF間での同期管理

1. OpenCL APIごとにCMA PFに分解
2. OpenCLのイベント変数（APIの引数）をもとにCMA PFの条件変数へ変換
3. これらの情報をsyscallを用いて通知

- 同期をDAGとして管理
  ノード：CMA PF
  エッジ：データフロー
- これらをOS内部に保持

Input → Ctrl Double Buffers → Input, Exe → Ctrl Double Buffers → Input, Exe, Copy → Ctrl Double Buffers

EXE, Output, Input
Execution ordering table
CMA Scheduler, Update, execution
Ctrl Double Buffers, Write PE, Write INST
Cube OS ●Kernel mode

1. 次のCMA PFを検索
2. 次のCMA PFを実行
3. CMA PFの実行が終了したらstate情報をアップデート

13

Execution ordering table

- ステートトランジションテーブル
  - CMA PF間の条件同期の管理
  - 各CMA PFのパラメータの管理

CMA primitive function id	wait for signal	State (state)	CMA primitive functio_info
0	NULL	Done	
1	state[0]	Running	
2	state[1]	Blocked	
3	state[1]	Blocked	
4	state[2] and state[3]	Blocked	

CMA PFの情報の構造体
・CMA PFの種類
・アドレス
・オフセット
・ダブルバッファの設定情報
等を保持

- Execuiton order table 用システムコール
  - add_order_table()
    - CMA PFを追加

14

OpenCL APIs

OpenCL API vs CMA function

OpenCL API	CMA function	Abstracted CMA PFs
clEnqueueWriteBuffer	Writing local memory	Crtl Double Buffers Input
clEnqueueReadBuffer	Reading local memory	Crtl Double Buffers Output
clEnqueueCopyBuffer	copy between local memories	Ctrl Double Buffers Copy
clEnqueueTask	Executing CMA	Write PE Write INST Crtl Double Buffers EXE

- これらのOpenCL API によってCMA-Cubeの制御を隠蔽
  - CMA-Cubeの制御をCMA PFに隠蔽
  - CMA PFをOSが制御することでパイプライン処理を実行

15

各レーヤごとの役割のまとめ

	役割	同期情報
OpenCL API	CMA-Cubeに対するデータ転送実行をAPIに隠蔽	OpenCL が提供するイベント変数でデータ転送，実行の同期を指定
OpenCL ライブラリ	CMA-Cubeでパイプライン処理ができるようにAPI中にCMA PFを隠蔽	複数のCMA PF間の同期情報をOpenCL APIから与えられたイベント変数より生成
CMA PF	ブロッキングが発生するCMA PFをOSが提供するタスクとして実行することでブロッキングをタスクに隠蔽 低レベルのレジスタ操作もCMA PFに隠蔽	CMA PFをノード，データフローをエッジとするDAGにて同期管理
CubeOS	CMA PFを実行制御することで，CMA-Cubeのパイプライン処理を隠蔽	CMA PFの実行状態を条件変数とみなし，同期管理

16

ライブラリOS協調用システムコール

- CubeOSはライブラリとの協調動作のために2つのシステムコールを提供

1．cma_task_create()
- CMA操作タスクの生成
- CMA-Cubeごとに初期化時に実行
- REAT_TASK, WRITE_TASK, EXE_TASK, DMA_TASKを生成

2．add_order_table()
- clEnqueueXXの中で利用
- CMA-PFの内容をCubeOSへ通知
- CMA-PFに必要なCMA番号，データのアドレスをcma_pf_info構造体に格納し引数に渡す．
- CMA PFのIDと，実行待ちを行うCMA PFを指定

cma_pf_info構造体
```
struct job_info{
 int task name;
 int cma_number;
 int src_addr;
 int src_offset;
 int dist_addr;
 int dist_offset;
 int data_size;
 int bsel_n;
}
```

17

clCreateBuffer()

- OpenCLのメモリ確保の関数でCMA操作タスクの生成
- CMA-Cubeの予約を指示
- CMA-Cubeごとに実行

```
clCreateBuffer(){
  //CMA操作タスクを生成
   cma_task_create();
}
```

- プログラマが使用するCMA-Cubeの数を決定
  - CMA操作タスクによるオーバヘッドを削減
    - メモリ使用量
    - タスク生成時間

18

clEnqueueWriteBuffer

- OpenCLの条件同期変数のイベント情報をCMA-PFの同期情報へ変換
- CMA-PF (Ctrl Double Buffers ,Write)の追加

```
clEnqueueWriteBuffer(){
  1．引数の必要な情報をCMA_PF_info構造体のメンバに代入
  2．引数で受け取ったイベントをCMA PF 間の同期に変換
  //3．Ctrl Double BuffersのCMA PFを追加
   add_order_table();
  //4．WriteのCMA PFを追加
   add_order_table();
}
```

clEnqueueWriteBuffer, clEnqueueReadBuffer, clEnqueueTaskでは，ダブルバッファ処理を最初に追加することで，OpenCLのコードからダブルバッファ処理を隠蔽することが可能

clEnqueueTask

- CMA-PF (Ctrl Double Buffers ,Exe)の追加
- OpenCLの条件同期変数のイベント情報をCMA-PFの同期情報へ変換

```
clEnqueueTask(){
  1．引数の必要な情報をCMA_PF_info構造体のメンバに代入
  2．引数で受け取ったイベントをCMA PF 間の同期に変換
 //３． Ctrl Double BuffersのCMA PFを追加
   add_order_table();
 //４．Write PEのCMA PFを追加
   add_order_table();
 // 5. Write INSTのCMA PFを追加
   add_order_table();
 // 6. ExeのCMA PFを追加
   add_order_table();
}
```

CMA-Cubeへの命令の書き込み等を隠蔽
Write PE, Write INSTは初回のみ実行
→初回か，2回目以降であるかを管理し，Write PE, Write INSTを発行することで，無駄な命令書き込みを削減

関連研究

[1]ローカルメモリを持つプロセッサ用のOpenCL実装
- Cellプロセッサを利用
- POSIXスレッドを利用 ⇔ 軽量な実行制御

[2]リコンフィギャラブルプロセッサを対象としたシステムソフトウェア
- データフローを考慮しアクセラレータの再構成
- 独自のAPIにてアクセラレータを操作 ⇔ OpenCL APIから制御

[1] An opencl framework for heterogeneous multicores with local memory.
Jaejin Lee, Jungwon Kim, Sangmin Seo, Seungkyun Kim, Jung-Ho Park, Honggyu Kim, Thanh Tuan Dao, Yongjin Cho, Sung Jong Seo, Seung Hak Lee, Seung Mo Cho, Hyo Jung Song, Sang-Bum Suh, and Jong-Deok Choi. In Valentina Salapura, Michael Gschwind, and Jens Knoop, editors, PACT, pp. 193-204. ACM, 2010.

[2] A task graph execution manager for reconfigurable multi-tasking systems.
Juan Antonio Clemente, Carlos Gonzalez, Javier Resano, and Daniel Mozos. Microprocess. Microsyst., Vol. 34, No.2-4, pp. 73?83, March 2010.

現状と今後の課題

- ハードウェア
  - ✓ Geyser-Cubeチップ上でのLinuxの動作
  - ✓ Geyser-Cubeチップ上での仮想メモリを搭載した独自OSの動作
  - ✓ CMA-Cubeチップ上での様々なアプリケーションの動作
  - ■ Geyser-CubeチップとCMA-Cubeチップを接続しテスト
- OpenCL library
  - ✓ 設計
  - ■ 実装
- Cube OS
  - ✓ 設計
  - ■ 実装
- 実チップでの評価
  - ■ 実チップ用評価環境の構築
  - ■ CMA-Cube上で実用的なアプリを動かし，CMA-Cubeの実行時間の算出

まとめ

- 組込み向けマルチコアアクセラレータ用のOpenCLランタイムライブラリを提案
  - OpenCLによってCMA-Cubeの制御を隠蔽
  - CMA-Cubeの制御をCMA PFに隠蔽
  - CMA PFをOSが制御することでパイプライン処理を実行
  - OSとライブラリが協調しCMA PFを軽量実行

AndroidOS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、それら四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、アニメーション表示を行う際に使用するアニメーション API の四種類を作成した。

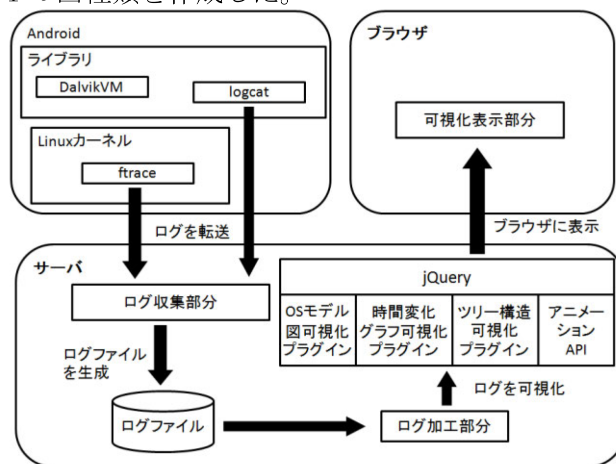


図 1　全体構成

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。また、アニメーションは表示速度が調整可能なので、利用者の見やすいスピードのアニメーションを表示でき

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

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

実装結果を図 2 に示す。左下が 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。また、待ち状態には 6 つの状態があり、それらの状態も色分けをして表示している。

左上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、正確な時間を知りたい場合はマウスカーソルを知りたい時間のグラフ上に置くとポップアップで表示する。

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。

右上にプロセス選択画面を表示している。可視化表示させたいプロセスをチェックすることでチェックしたプロセスのみを表示することができる。

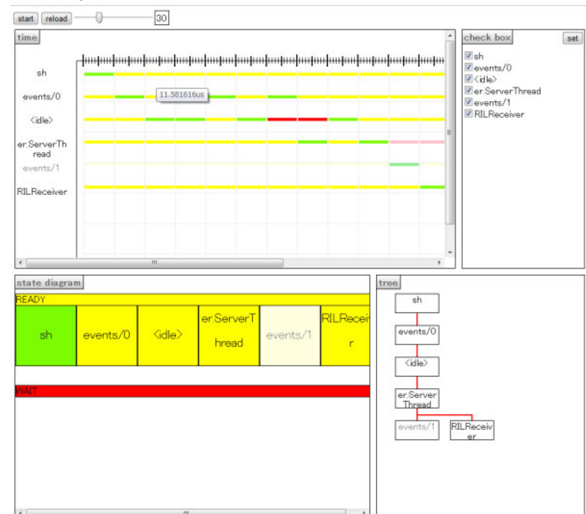


図 2　実装結果

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

AndroidOS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
††東京都立　八王子桑志高等学校
†††拓殖大学　工学部　情報工学科

背景

- Androidは組込みOSに利用する機会の増加
  - 携帯端末やタブレットで利用
- 利用者の増加に伴い、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

- LinuxカーネルとDalvikVMとのプロセスの対応
  - Androidは2つの実行環境で動作
  - 2つの実行環境の間でどのような対応をしているか
- 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によってはアプリケーションエラーが起きる

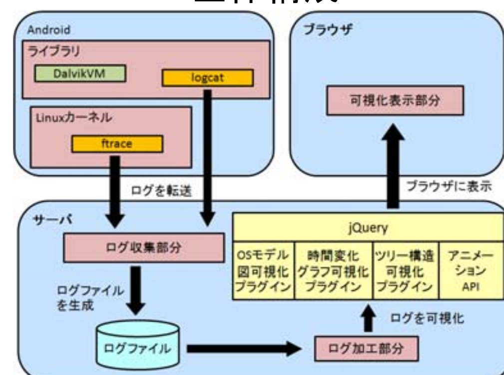
目的

- Androidのプロセスを可視化、表示
  - プロセス管理を可視化
    - OSの基礎を理解できる
    - 複数あるプロセスの動作を理解
    - プロセス同士の関係性を理解
  - Androidの2つの実行環境を可視化
    - Android内の2つの実行環境でどのように動作しているか理解できる
- 学習者の対象としては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
  - 実行画面の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
  - 利用者により理解してもらうためにアニメーション
- ログファイルをサーバ内に保存
  - サーバに送ることにより、Android内の保存領域を使用せずに済む
  - サイズの大きいログファイルも保存可能

全体構成



可視化内容

- Androidの可視化内容
  - OSモデル図
    - 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
- プロセスを選択し、選択したプロセスだけを表示
  - 利用者が知りたいプロセスの情報だけを見ることが可能
- 可視化の表示をコンポーネント化
  - 可視化表示部分のプログラムを分割
  - 分割されたプラグラムを変えることで様々な可視化表示が可能

アニメーション表示

- アニメーションにはjQueryを使用
  - jQueryとはJavaScriptライブラリの1つ
  - 様々なプラグインで描画
- プラグインを可視化表示に利用
  - 少ないプログラムコードで表示可能
  - プラグインにはWebAPIを使用し、可視化表示が容易
- アニメーションの表示速度は調整が可能
  - 好みの速さで可視化を見ることができる

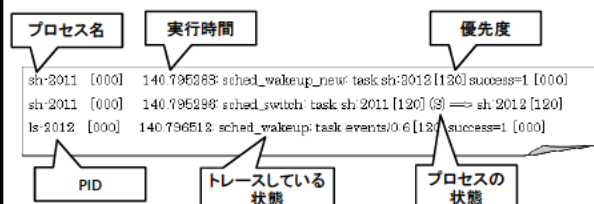
プラグイン

- OSモデル図可視化プラグイン
  - プロセスの状態変化を表示
- 時間変化グラフ可視化プラグイン
  - 時間変化による状態の移り変わりを表示
- ツリー構造可視化プラグイン
  - 親子関係などを表示
- アニメーションAPI
  - 可視化表示に必要なアニメーションをAPI

可視化情報

- ログ取得にはftraceとlogcatを使用
  - ftraceはLinuxカーネル内にあるトレーサ
    - プロセスの情報を取得
  - logcatはAndroid内にあるデバック用ツール
    - ftraceのログファイル情報と合わせて可視化

ftraceのログ



logcatのログ

ログの種類
PID
説明

D/dalvikvm(　46)　creating instr width table
I/AudioFlinger(　47)　AudioFlinger's thread 0x1beb0 ready to run
I/SamplingProfilerIntegration(　46)　Profiler is disabled

Tag

可視化画面

時間変化グラフ
プロセス選択画面
ツリー構造
OSモデル図

時間変化グラフ

実行時間
正確な時間を表示
プロセス名
状態遷移のタイミングを表示

時間変化グラフ

- 各プロセスの状態遷移を棒グラフで表示
  - どのタイミングで状態が遷移するか理解できる
- 正確な時間を表示
  - 調べたい時間にマウスを置くと時間がポップアップで表示
- プロセスが2つ実行環境で動作している場合
  - プロセスの名前と状態の色を薄くして表示

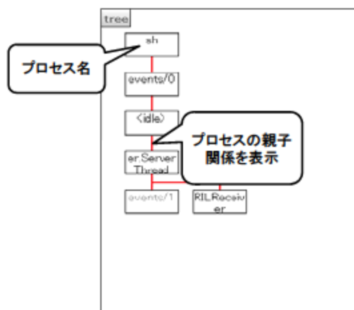
OSモデル図

3つの状態に分けて表示
プロセス名

OSモデル図

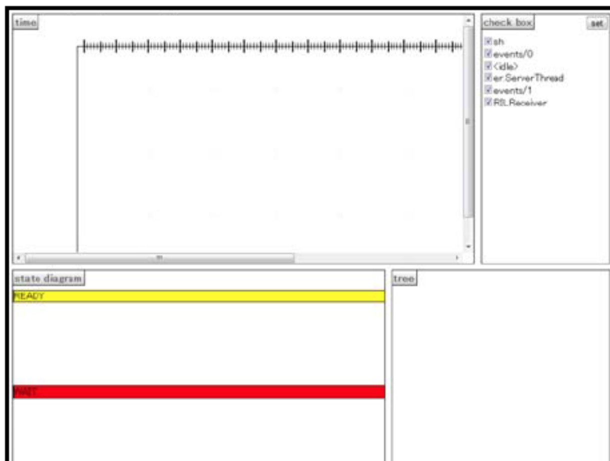
- プロセスの状態を色で識別
  - 実行状態⇒緑色
  - 実行可能状態⇒黄色
  - 待ち状態⇒赤色
- 待ち状態には6つの状態がある
  - 6色で6つの状態を表示
- 状態の遷移をアニメーション表示
  - 静的な表示よりも動的な表示の方が理解しやすい

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表示
  - 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るか把握できる
- アニメーションの表示
  - プロセスの生成と消滅が把握できる



おわりに

- 研究成果
  - 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
  - 可視化のアニメーション表示の速度が調整可能
  - 可視化するプロセスを選択可能
- 今後の予定
  - 可視化の実行画面のUIを改良
  - 可視化するログファイルの指定ができるようにする

複数のポリシ/メカニズムを搭載した学習向け組込みOSの実装

茂木　高宏[†]　　岩澤　京子[‡]　　早川　栄一[‡]

拓殖大学大学院博士前期過程 電子情報工学専攻[†]

拓殖大学工学部情報工学科[‡]

1　はじめに

現在，組込みOSの設計として，複数のポリシ/メカニズムが考えられている．組込みアプリケーション技術者では，実現したいアプリケーション特性に適応したポリシ/メカニズムを搭載する組込みOSの選択が行なわれている．そのため，複数のポリシ/メカニズムに対する知識と動作の理解が求められている．

ポリシの問題点として，組込みOSでは，メカニズムからポリシの完全分離が困難である．そのため，ポリシに対する柔軟性が低下し，複数のポリシの動作学習が難しい．メカニズムの問題点として，メカニズムは，組込みOSの特徴に応じて変化する．よって，単一環境下から，OSの特徴を無視した複数のメカニズムの学習が難しい．さらに，OSの記述方式が複雑化し，内部構造の理解が困難，及び可読性の低下があげられる．

問題解決法として，複数のポリシ/メカニズムを搭載した組込みOSを自作する事を本研究の目的とする．また，本研究OSの機能及び複数のポリシ/メカニズムを学習させたいサンプルをタスクセットとして実装し，ライブラリとして提供する．これにより，学習者はポリシやメカニズムの変更がタスクの動作に与える影響について，単一の環境下で学習する事が可能となる．

2　研究方針

研究方針は，OS設計方針及びOS記述方針，学習方針とし，以下の(1)～(3)に示す．

(1)　OS設計方針

実装対象とする複数のメカニズムとポリシを(1-1)と(1-2)に示す．

(1-1)　複数のメカニズム

システムコールの方式を2通りの方法で実装する事にした．具体的には，トラップを使用してOSの特権パーティションを切替える方式とトラップを使用しないサブルーチンコールの方式である．

(1-2)　複数のポリシ

OSの機能において重要なタスクスケジューリングを13種実装する事にした．さらに，優先度逆転防止プロトコルを6種実装する事にした．

(1-1)と(1-2)の動作学習をするために，OSの基本的な機能を提供する事にした．OSの基本的な機能の仕様は，広く採用されている[1]を参考にする事にした．

(2)　OS記述方針

OSの記述は，C言語，アセンブラ，リンカスクリプトにする事にした．OSソースコード可読性の考慮として，コンフィギュレーションを行なわず,すべて[1]の動的API仕様とする．さらに，MISRA C2004([2])を参考にコーディングを行い，コメント比率とアセンブリ比率の減少を行なう．

(3)　学習方針

OSの動作はターゲット上とし，対象CPUはH8とARMとする．学習の方針として，(3-1)と(3-2)に示す．

(3-1)　直感な動作学習

同研究室で開発されたブラウザベース可視化ツールを使用する．

この学習では，複数のポリシ(タスクスケジューリング)がタスクに与える動作の学習が可能となる．

(3-2)　詳細な動作学習

本研究OSのサンプルタスクセットライブラリのタスクソースコードと，ターゲット上で動作させた時のレスポンスを突き合わせる方法とした．

この学習では，複数のポリシにおけるタスク切替え事象やカーネルオブジェクトの学習が可能である．

3　組込みシステム全体機能

3.1　組込みシステムの全体構成

組込みシステムの全体構成を図1にまとめた．サンプルタスクセットライブラリは本研究OSの機能を網羅したサンプルタスクが格納されている．

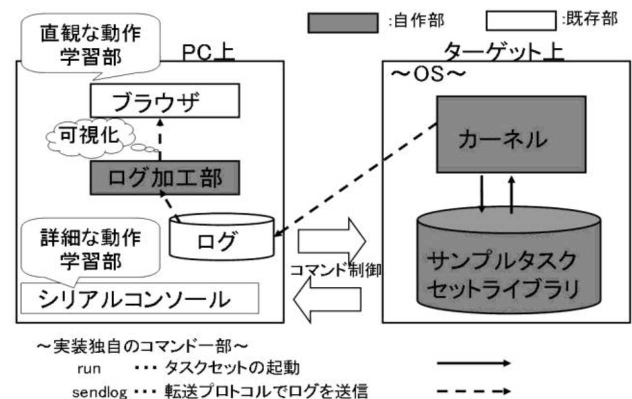


図1 組込みシステム全体構成

上図の実装独自のコマンドであるrunとsendlogの詳細を以下の(1)と(2)に示す．

(1)　runコマンド

シリアルコンソール上から，runコマンドでサンプルタスクセットを選択すると，カーネルが指定されたサンプルタスクセットを起動し，レスポンスをシリアルコンソール上に返却する．

(2)　sendlogコマンド

シリアルコンソール上から，sendlogコマンド選択すると，カーネルはログを転送プロトコルでPC上に送る．このログは，直感な学習を行う可視化ツールで使用する．

3.2　カーネル全体構成

機能からみたカーネルの全体構成を図2にまとめた．これらの機能をタスクから使用するために，60程度の動的API([1])の実装を行なった．

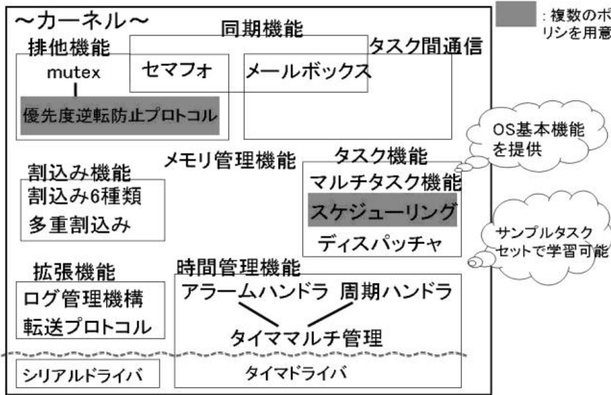


図2 カーネルの機能

4　複数のポリシがタスクに与える影響

本研究OSは6種の優先度逆転防止プロトコルの実装をし,それぞれの優先度逆転プロトコルに対して発生する3つの主作用(デッドロックの誘発現象，推移的優先度継承現象，連続ブロッキング現象)と1つの副作用(デッドロックの予防)をサンプルタスクセットライブラリで学習する事が可能である．さらに，これらは本研究OSに実装した13種のスケジューリング環境下と待ちタスクのレディー返却アルゴリズムによって，レスポンスが変化する．これらの関係を図3にまとめた．

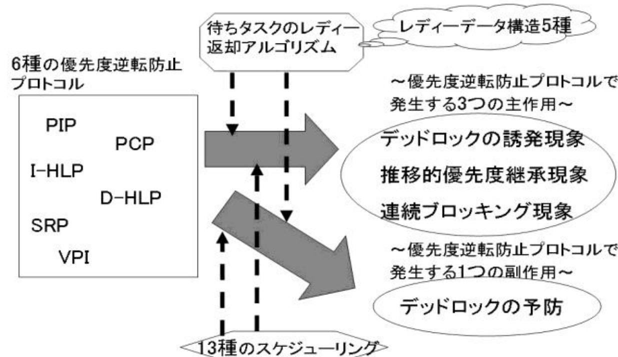


図3 複数のポリシがタスクに与える影響

5　スケジューリング動的切替え機能

本研究OSは,メモリへOSをリロードせずに，スケジューリング自体を動的に切替えいくことが可能である．切替えは,実装独自であるスケジューリング切替えシステムコールで行う方式とした．これにより,同一のタスクセットに対して異なるスケジューリングを適用したレスポンスを体験する事ができる．スケジューリング切替えシステムコールはユーザ任意のタイミングで発行する事ができる．具体的には，initタスクとユーザタスクから発行が可能である．

本研究OSで，initタスクから異なるスケジューリングの切替え方法の一例(FCFSスケジューリングから優先度スケジューリングへ切替え)を図4に示す．

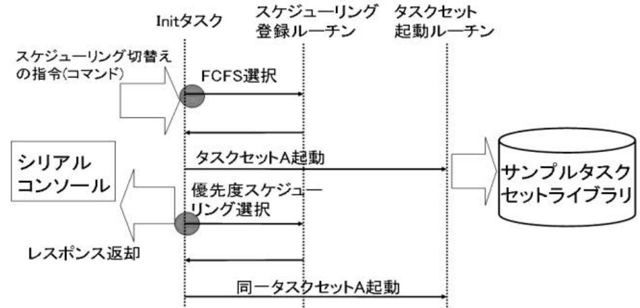


図4 異なるスケジューリングの切替え方法

上図の例は，ユーザからシリアルコンソールを通して,スケジューリング切替えのコマンドが入力されると，initタスクからスケジューリング切替えシステムコールが発行される．そして，同一タスクセットに対して異なるスケジューリングを適用している．

6　他OSとの可読性比較調査結果

本研究OSと他OSとの可読性比較調査の結果を下表1に記す．

表1 他OSとの可読性比較調査結果

	ソースコード行(コメント比率)	アセンブラ行(アセンブラ比率)	①goto文	②条件付きマクロ	③関数マクロ
本研究OS(H8)	28537(28%)	140(1%)	0	0	2
本研究OS(ARM)	30008(30%)	212(1%)	0	0	6
H8/OS[3]	10726(2.8%)	197(1.8%)	0	169	2
toppers/ssp[4]	20880(12%)	625(2.9%)	15	100	42

表1の①～③は[2]におけるルールの一部である．

表1より，本研究OSはコメントが多く，アセンブラが少ないので，可読性が向上したと考えられる．

7　まとめ

複数のポリシ/メカニズムを搭載した組込みOSとサンプルタスクセットライブラリがH8とARMで実装できた．また，OSの可読性を向上させる事ができた．

今後の課題として,直観な学習と詳細な学習の完全対応を行なう．さらに，複数のポリシ/メカニズムに対する動作の差異の学習を向上させる機能を本研究OSに組込む事である．

参考文献

[1]μITRON ver4.02.00仕様書
http://www.ertl.jp/ITRON/SPEC/mitron4-j.html
[2]MISRA C
http://www.openrtp.jp/wiki/_hara/ja/RtORB/MISRA-C-RULE.html
[3]H8/OSソースコード
http://mes.sourceforge.jp/h8/index-j.html
[4]TOPPERS/SSPカーネルソースコード
http://www.toppers.jp/ssp-kernel.html

複数のポリシ/メカニズムを搭載した学習向け組込みOSの実装

茂木高宏†，岩澤京子‡，早川栄一‡

†拓殖大学大学院博士前期課程　電子情報工学専攻
‡拓殖大学　工学部　情報工学科

目次

1. 背景と問題点
2. 目的
3. 方針
4. 全体構成
5. 複数のポリシとメカニズム
6. 実装機器と可読性評価
7. おわりに

背景

- 組込みOSの設計として，複数のポリシ/メカニズムが考えられている
- 組込みOSアプリケーション技術者では，

  実現したいアプリケーション特性に適応したOSのポリシ/メカニズムの選択が行なわれている

  - 複数のポリシ/メカニズムに対する知識が求められている

    ➡ 複数のポリシ/メカニズムの内部構造及び動作の理解

問題点

- 組込みOSでは，メカニズムからポリシの完全分離が困難
  - ポリシに対する柔軟性の低下

    ➡ 複数のポリシの動作学習が難しい

  複数のポリシにおける動作の差異の学習も困難
- メカニズムは，組込みOSの特徴に応じて変化
  - 単一環境下から，OSの特徴を無視した複数のメカニズムの動作学習が難しい
- OS記述方式が複雑化し，内部構造の理解が困難
  - コンフィギュレーションの存在

    ➡ 仕様が明確化されていない

目的

- 複数のポリシ/メカニズムを搭載した組込みOSの自作
  - 複数のポリシ/メカニズムの動作学習が可能
  - 複数のポリシ/メカニズムに対する動作差異の学習が容易に可能
  - 可読性を向上させたOSソースコード

対象：組込みアプリケーション技術者

~~1. 背景と問題点~~
~~2. 目的~~
3. 方針
4. 全体構成
5. 複数のポリシとメカニズム
6. 実装機器と可読性評価
7. おわりに

方針：OS設計方針

- 複数のメカニズム
  - システムコールの方式
- 複数のポリシ
  - タスクスケジューリング
  - 優先度逆転防止プロトコル
  - ➡ 動作の学習をするために
    - サンプルタスクセットライブラリを提供
    - OSの基本的な機能を提供
      - 広く使用されているμITRON4.0仕様を参考

方針：OS記述方針

- C言語，アセンブラ，リンカスクリプトを使用して記述
- OSソースコード可読性考慮として，
  - コンフィギュレーションを行なわない
    - ➡ すべてのシステムコールはμITRON4.0の動的APIを使用
  - MISRA C2004を参考にコーディング
  - コメント比率の増加とアセンブラ比率の減少

方針：学習方針

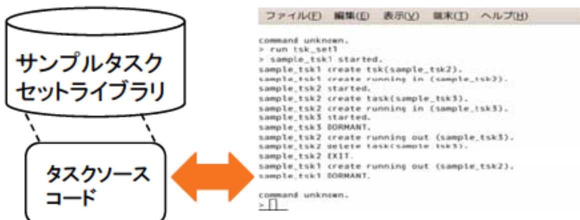
- OSの動作は実機上
- 複数のポリシ/メカニズムの動作の学習方法
  - 直感な動作学習
    - ➡ 同研究室で開発された可視化ツールを使用
  - 詳細な動作学習
    - ➡ シリアルコンソールに表示されるシリアルメッセージ
    - ➡ サンプルタスクセットライブラリ

方針：直観な動作学習

- 可視化ツールの使用
  - 複数のポリシ(タスクスケジューリング)がタスクに与える動作の学習が可能

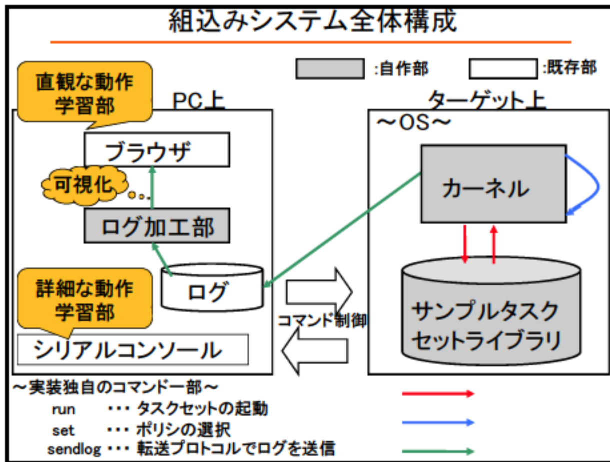
方針：詳細な動作学習

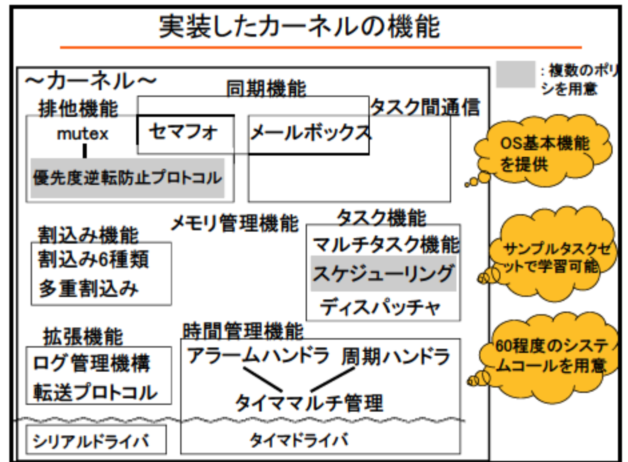
- シリアルコンソールを使用
  - 複数のポリシにおけるタスク切替えの事象を学習
  - カーネルオブジェクトの学習



- 実機上のOSは，シリアルコンソールからコマンドで制御

1. ~~背景と問題点~~
2. ~~目的~~
3. ~~方針~~
4. **全体構成**
5. 複数のポリシとメカニズム
6. 実装機器と可読性評価
7. おわりに





~~1. 背景と問題点~~

~~2. 目的~~

~~3. 方針~~

~~4. 全体構成~~

5. 複数のポリシとメカニズム

6. 実装機器と可読性評価

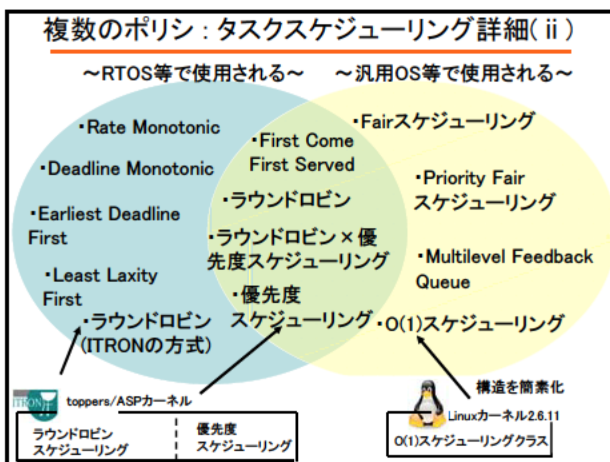
7. おわりに

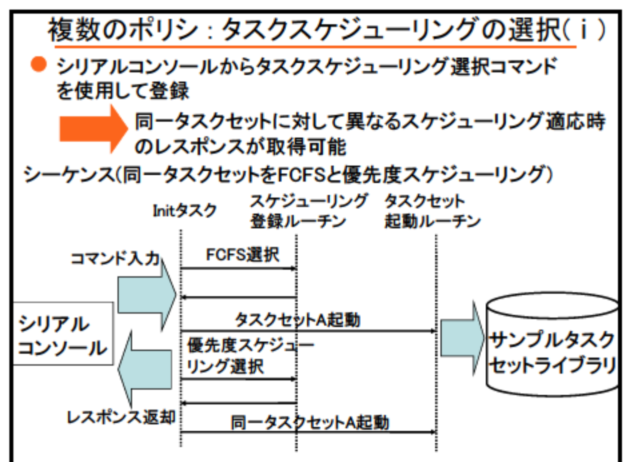
複数のポリシ：タスクスケジューリング詳細(i)

- 13種のタスクスケジューリングを実装
  - 広く採用されているタスクスケジューリングを対象
  - 比較対象用として，汎用OS等で使用されるスケジューリングも対象

サンプルタスクセットにより，動作学習が可能

- スケジューリングによって，レディー管理データ構造が変化
  - レディー管理データ構造を5種実装





同一タスクセットに異なるスケジューリング適応のレスポンス

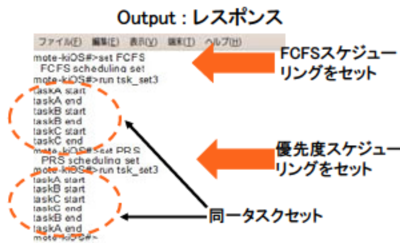
Input : サンプルタスクセット

```
TaskA(優先度低) {
    puts("taskA start");
    タスク生成と起動(TaskB);
    puts("taskA end");
}
```

```
TaskB(優先度中) {
    puts("taskB start");
    タスク生成と起動(TaskC);
    puts("taskB end");
}
```

```
TaskC(優先度高) {
    puts("taskC start");
    puts("taskC end");
}
```

Output : レスポンス



FCFSスケジューリングをセット

優先度スケジューリングをセット

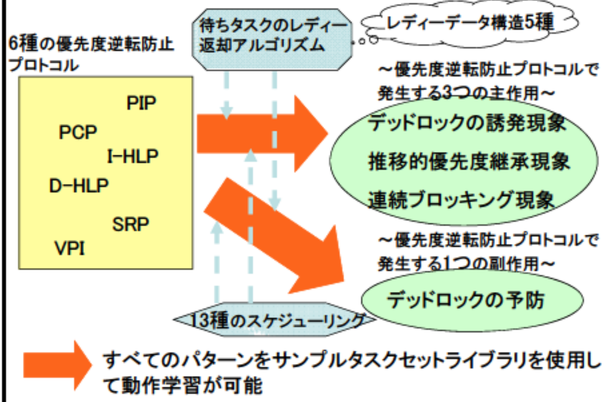
同一タスクセット

複数のポリシ : 優先度逆転防止プロトコル(i)

- 6種の優先度逆転防止プロトコルを実装
  - Priority Inheritance Protocol(以下PIP)
  - Priority Ceiling Protocol(以下PCP)
  - Immediate Highest Locker Protocol(以下I-HLP)
  - Delay Highest Locker Protocol(以下D-HLP)
  - Stack Resource Policy(以下SRP)
  - Virtual Priority Inheritance(以下VPI)

➡ mutexの属性として実装

複数のポリシ :優先度逆転防止プロトコル(ii)



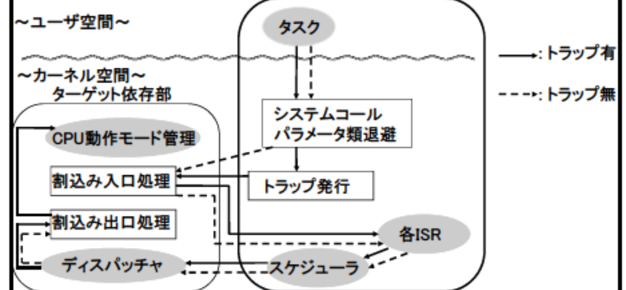
➡ すべてのパターンをサンプルタスクセットライブラリを使用して動作学習が可能

複数のメカニズム : システムコールの方式

- トラップ有りのシステムコール
- トラップ無しのシステムコール

2通り用意

➡ トラップ無しではCPU動作モードを切替ない



~~1. 背景と問題点~~

~~2. 目的~~

~~3. 方針~~

~~4. 構成~~

~~5. 複数のポリシとメカニズム~~

6. 実装機器と可読性評価

7. おわりに

実装 : ターゲットH8

～ボード情報～

- CPU : H8/300H
- クロック : 25MHz
- メモリ : DRAM(2MB)
  RAM(16KB)
  フラッシュ(512KB)
- シリアルポート : RS232C ×1
- LANポート : RTL8019ASコントローラ ×1

実装：ターゲットARM

～ボード情報～

- ・CPU：DM3730(ARM cortex-A8)
- ・CPUクロック：1GHz
- ・メモリ：
  - LPDDR RAM(512MB)
  - SRAM(64KB)
  - BootROM(1MB)
  - NANDなし
- ・シリアルポート：RS232C ×1(UART)
- ・JTAGポート ×1

他OSとの可読性比較調査の結果

	ソースコード行（コメント比率）	アセンブラ行（アセンブラ比率）	Ⅰ(件)	Ⅱ(件)	Ⅲ(件)
本研究OS(H8)	28537 (28%)	140 (1%)	0	0	2
本研究OS(ARM)	30008 (30%)	212 (1%)	0	0	6
H8/OS	10726 (2.8%)	197 (1.8%)	0	169	2
toppers/ssp	20880 (12%)	625 (2.9%)	10	100	42

MISRA C:2004のルール

(例) Ⅰ：goto文 → 制御フローの必須事項

Ⅱ：条件付きマクロ制限 → 前処理指令の推奨事項

Ⅲ：関数マクロ制限 → 前処理指令の推奨事項

コメント記述ルール

- ファイル及び関数ヘッダコメント
- プリプロセスとグローバルデータ
- アセンブリ記述行

おわりに

成果

- 複数のポリシ/メカニズムを搭載した組込みOSとサンプルタスクセットライブラリがH8とARMで実装できた
- OSの可読性を向上させる事ができた

今後の課題

- 直観な学習と詳細な学習の完全対応を行う
- 複数のポリシ/メカニズムに対する動作差異の学習を可能にする

Alkanet: 仮想計算機モニタを用いたマルウェアトレーサ

大月 勇人 † 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 背景

近年, マルウェアの脅威が問題となっている. マルウェア対策には, マルウェアを解析し, どのような挙動をするかを調査する必要がある. しかし, 1 日に数千もの新種や亜種が出現しているため, 1 体のマルウェアの解析に時間を費やすことができない.

マルウェアの解析では, まず, 実際にマルウェアが実行した動作を観測する動的解析を行い, マルウェアの概略を把握する. 最近のマルウェアの多くは, アンチデバッグと呼ばれる機能を持つ [1]. これは, マルウェア自身が動的解析されていることを検知し, 実行の停止や解析の妨害などを行うものである. 従来の動的解析ツールは, Windows が提供するデバッグ支援機能を用いて動作する. しかし, この機能は本来ソフトウェアをデバッグするためのものであり, マルウェアは解析されていることを容易に検出できる. 対策として, 動的解析ツールの隠蔽やアンチデバッグ機能自体の無効化, あるいはデバッグ支援機能に依存しない解析機能などが必要である. しかし, すべてのアンチデバッグの手法を回避することは困難である. また, 動作環境への影響が大きくなり, オーバヘッドの増大や不正確な解析の原因となる.

動的解析によるマルウェアの動作環境へ影響を抑制するために, 仮想計算機モニタ (VMM) やエミュレータを用いて, 解析機構をマルウェア動作環境の外部に実現する手法がある. VMM やエミュレータは, それらの実現する仮想環境内で動作する OS よりも高い権限で動作するため, OS やその上で動作している全てのプロセスを透過的に監視することが可能である. しかし, エミュレータは, システム全体をソフトウェアでエミュレートするため, オーバヘッドが問題となってしまう. また, 一般的な VMM は, 特定のハードウェアをエミュレートすることや, VMM とゲスト OS との通信用のインターフェースを有するなどといった特徴を有してことが多いため, マルウェアに検出されやすい.

また, 解析の対象についても十分検討する必要である. 近年, 活動の隠蔽や解析の困難化のために, 別のプロセスにスレッドを挿入し, 活動するマルウェアが増加している. マルウェアを動的解析を行う既存のシステムの多くは, プロセスレベルでマルウェアを区別しているため, 別のプロセスに挿入されたスレッドを追跡できない. 上記の背景から, アンチデバッグ機能を持つマルウェアや他のプロセスへ感染するマルウェアについても, 観測が可能な動的解析システムが必要である. そこで, VMM を用いた動的解析システム Alkanet を開発している.

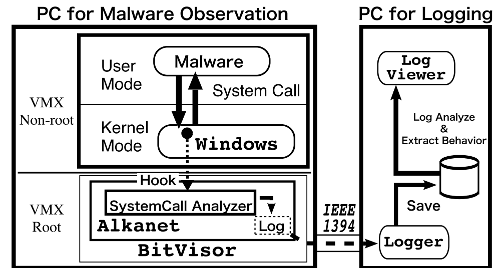


図 1 Alkanet の全体構成

以下, 本論文では, 2 章で Alkanet の概要と構成について述べる. 3 章で実際に Alkanet 上で動作させたマルウェアの解析結果と考察について述べる. 最後に, 4 章で本稿をまとめる.

2 Alkanet

2.1 概要

Alkanet は VMM を用いたマルウェア動的解析システムである. アンチデバッグを回避し動的解析を行うためには, マルウェアより高い権限で解析機構を実現する必要がある. VMM にマルウェア解析機構を実現することで, 多くのアンチデバッグ手法を回避できる.

マルウェアの挙動を取得するためには, その意図の理解容易性および解析速度の観点から, 機械語レベルよりも API レベルのトレースが有効である. 特に, ユーザモードのマルウェアがシステムに影響を及ぼす動作を行うためには, システムコールを発行する必要があることから, システムコールのトレースによってマルウェアの挙動を取得する.

取得したシステムコールトレースのログにより, マルウェアの挙動を分析できるが, 複雑なマルウェアであった場合, 記録されるログは人手での分析するには膨大な量となる. そこで, システムコールトレースのログをさらに分析し, マルウェアの特徴的な挙動を抽出したレポートを出力するツール群も構築している.

2.2 構成

Alkanet の全体構成を図 1 に示す. Alkanet は, VMM である BitVisor[2] の拡張機能として実装している. BitVisor は, ホスト OS を必要としないハイパーバイザ型の VMM である. Intel 製 CPU における仮想化支援機能の Intel VT(Virtualization Technology) を利

```
No. [5212, 5213]: Polipos.exe (Cid: 54c.18c) -> svchost.exe (Cid: 480.2c4) (Code Injection)
   No. [5288, 5289]: svchost.exe (Cid: 480.2c4) -> svchost.exe (Cid: 480.22c)
     ...
     No. [11340, 11341]: svchost.exe (Cid: 480.22c) -> svchost.exe (Cid: 480.720)
       No. [14368, 14369]: svchost.exe (Cid: 480.720) -> rundll32.exe (Cid: 220.7f8) (Code Injection)
```

図 3　コードインジェクションされた svchost.exe から派生するスレッド

用しているため, ソフトウェアのみで実現されたエミュレータや VMM に比べ, 高速に動作することが可能である. また, Windows を修正なしで実行することができる. さらに, BitVisor はハードウェアのエミュレートを行わない準パススルー型の VMM である. そのため, ゲスト OS は実マシンに搭載されているハードウェアが見られる. よって, QEMU などの特定のハードウェアをエミュレートするエミュレータや VMM に比べ, ハードウェア的特徴からマルウェアに検出されることはない. マルウェアの実行環境であるゲスト OS には, 32bit 版 Windows XP SP3 を用い, この環境で発行されるシステムコールをフックし, その種類や引数を取得する.

Alkanet で取得したログは, ロギング用 PC から IEEE 1394 を用いて取得する. IEEE 1394 は, 接続先デバイスの物理メモリを Direct Memory Access で読み書きできる. そのため, マルウェアに検知・妨害されずにログの取得が可能である. また, ログ取得後に, ログを分析し, マルウェアの挙動を示すレポートを出力するツール群もロギング用 PC 上で動作する.

2.3　取得情報

Windows における実行単位は, スレッドである. また, マルウェアによるコードインジェクションによって, 通常のプロセスの中にも「悪意あるスレッド」が存在する場合がある. したがって, システムコール発行元をスレッドレベルで区別するために, プロセス ID とスレッド ID の組である Cid とイメージ名を取得する. さらに, マルウェアの挙動を調査するために, システムコールの引数と戻り値を取得する. しかし, 引数や戻り値には, ポインタや OS 固有のデータ構造が用いられることが多く, その値だけでは不十分な場合がある. したがって, これらのデータ構造を解釈し, 必要な情報を補う必要がある. 以上から, 次の情報を取得する.

- システムコール発行元の Cid とイメージ名
- システムコール番号
- システムコールの引数と戻り値
- 固有のデータ構造に対する補足情報

3　評価

Alkanet がマルウェア解析に有効であることを確認するために, CCC DATAset 2011[3] で活動が記録されている実際のマルウェアを用いて解析を行った. ここでは, Polipos.exe と呼称する.

図 2 に Polipos.exe を実行して得たシステムコールのログの一部を示す. ここでは, Polipos.exe が, NtCreateThread を発行し, 別のプロセスの explorer.exe に対して 1e8 の ID を持つスレッドを作成している. これはコードインジェクションの挙動である. 図 3 は, 図 2 のようなログを分析することで, プロセスを越えて派生するスレッドを追跡し, 階層構造状に示している. ここでは, Polipos.exe によって, svchost.exe に作成されたスレッドから派生したスレッドが, さらに rundll32.exe に拡散している. このように, マルウェアが別のプロセスにコードインジェクションを行った場合でも, スレッドレベルで区別し, 追跡可能であることが確認できた.

```
No. :    6339        Time:    689820849
Type:    sysenter
SNo.:    35 (NtCreateThread)
Cid :    bc.304      Name:    Polipos.exe
Note:    PID: b0, ProcessName: explorer.exe

No. :    6340        Time:    689820959
Type:    sysexit
Ret :    0 (STATUS_SUCCESS)
SNo.:    35 (NtCreateThread)
Cid :    bc.304      Name:    Polipos.exe
Note:    Cid: b0.1e8, ProcessName: explorer.exe
```

図 2　explorer.exe へのコードインジェクション

4　まとめ

本論文では, 仮想計算機モニタを用いて, マルウェアの動的解析を実現する “Alkanet” の実装, およびマルウェアが観測できることについて述べた.

今後の課題として, ハニーポットなどと連携し, ネットワークを用いた挙動の解析機能の充実が挙げられる. また, Alkanet のログを, 既存のシステムコールのログを用いた異常検知やマルウェアのクラスタリングを行う既存手法などに利用できるか評価を行う.

参考文献

[1] N. Falliere: “Windows Anti-Debug Reference,” http://www.symantec.com/connect/articles/windows-anti-debug-reference (2007, Last accessed, July 2012).

[2] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo and K. Kato: “BitVisor: a thin hypervisor for enforcing i/o device security,” In Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, pp. 121–130, Washington, DC, USA (2009), ACM.

[3] 畑田充弘, 中津留勇, 秋山満昭: “マルウェア対策のための研究用データセット 〜 MWS 2011 Datasets 〜,” コンピュータセキュリティシンポジウム 2011 論文集, 第 2011 巻, pp. 1–5 (2011).

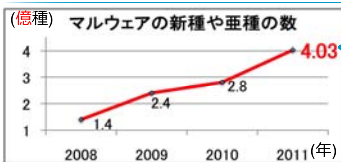
Alkanet: 仮想計算機モニタを用いたマルウェアトレーサ

大月 勇人 (立命館大学)

もくじ

1. 研究背景
2. 既存技術の問題点
3. アプローチ
4. Alkanet
5. システムコールフックの流れ
6. マルウェアの挙動と Alkanet の監視するシステムコール
7. 実際に観測されたマルウェアの挙動
8. まとめ

背景



マルウェアが急速に増加！
2011年には**4億300万種**の新種や亜種が出現！
(Symantec のデータより†)

- 短時間で解析し, マルウェアの意図や概略を把握したい
  - マルウェアを実行し, 挙動を観測することで解析する動的解析が有効
- しかし, マルウェアの巧妙化により, 観測自体が困難となっている
  - アンチデバッグ:
    観測ツールを検知し, 観測・解析を妨害する
  - コードインジェクション:
    一般のプロセスに感染し, 「悪意あるスレッド」を潜ませる

† http://www.symantec.com/ja/jp/about/news/release/article.jsp?prid=20100428_02,
http://www.symantec.com/ja/jp/about/news/release/article.jsp?prid=20110412_01,
http://www.symantec.com/ja/jp/about/news/release/article.jsp?prid=20120501_01

既存技術の問題点

- アンチデバッグの回避
  - 1つ1つのアンチデバッグについて対策
    - 例) PhantOm: OllyDbg のプラグイン
    - **すべてのアンチデバッグを回避することは困難である**
  - OS のデバッグ API を用いずにデバッガ相当の機能を実現
    - 例)VAMPiRE: ページフォルトハンドラやトラップフラグによる制御など
    - **動作環境への影響が大きく, 実行速度の低下が著しい**
  - 仮想環境を用いて OS やプロセスを透過的に監視
    - 例) TTAnalyze: QEMUベースのマルウェア自動解析ツール
    - **エミュレートするハードウェアの特徴や, ゲスト・ホスト間の通信路から検出される**
- 悪意あるスレッドの追跡
  - デバッガ: アタッチできるプロセスが1つのみ
  - TTAnalyze など解析システム: プロセスレベルでしかマルウェアを区別しない
  - **別のプロセスに潜んだスレッドを追跡できない**

アプローチ (1/2)

- マルウェアに検知されない観測システムの実現
  - マルウェアよりも高い権限で動作
  - マルウェア動作環境への影響を抑制

仮想計算機モニタ(VMM)として実現

- VMM を検出されないために
  - ハードウェア構成を固定しない

  準パススルー型 VMM BitVisor をベースにする

    - ゲスト OS は実マシンのハードウェアをそのまま認識する
  - ゲスト OS と通信せずに内部の情報を得る

  ゲストOSのメモリの内容を解釈し, 情報を取得

アプローチ (2/2)

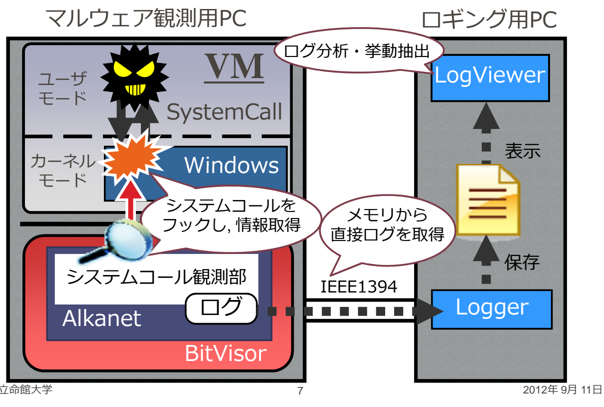
- マルウェアの意図を理解しやすい情報を提供
  - 粒度の観点から命令単位よりもAPI単位の観測が有効
  - 悪意あるスレッドがシステムに影響を与えるにはシステムコールが必要

システムコールトレースを基にした挙動解析

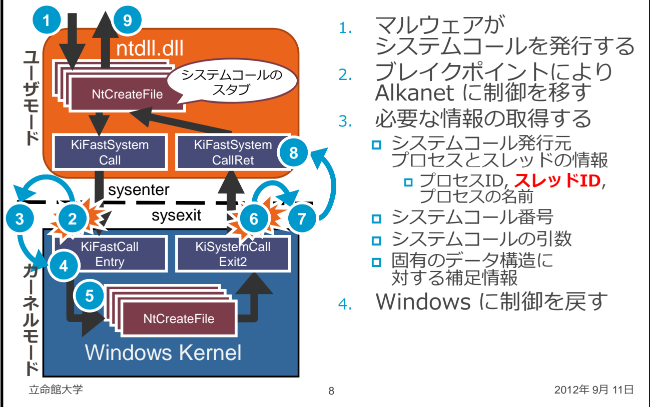
- 悪意あるスレッドを追跡し, 挙動を観測
  - コードインジェクションを構成する挙動を観測
    - 別プロセスへのメモリ書換え, DLL 挿入, **スレッド作成**など
  - システムコールの発行元をスレッドレベルで区別

作成されたスレッドの情報と発行元の情報から悪意あるスレッドを追跡

Alkanet

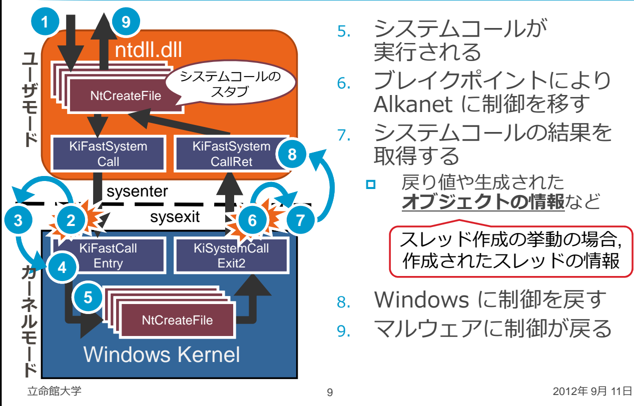


システムコールフックの流れ (1/2)



1. マルウェアが システムコールを発行する
2. ブレイクポイントにより Alkanet に制御を移す
3. 必要な情報の取得する
   - システムコール発行元 プロセスとスレッドの情報
     - プロセスID, **スレッドID**, プロセスの名前
   - システムコール番号
   - システムコールの引数
   - 固有のデータ構造に 対する補足情報
4. Windows に制御を戻す

システムコールフックの流れ (2/2)



5. システムコールが 実行される
6. ブレイクポイントにより Alkanet に制御を移す
7. システムコールの結果を 取得する
   - 戻り値や生成された **オブジェクトの情報**など

スレッド作成の挙動の場合, 作成されたスレッドの情報

8. Windows に制御を戻す
9. マルウェアに制御が戻る

マルウェアの機能を構成する挙動

- 一般的な挙動
  - ファイルの作成/削除, 読み書き
  - レジストリの参照, 設定
  - ネットワークへの送信, 受信
  - プロセスの作成, 終了
  - ドライバのロード, アンロード
- コードインジェクションの挙動
  - **CreateRemoteThread**: 指定したプロセスにスレッドを作成する API
  - LoadLibrary や WriteProcessMemory などと 組合せると, 別プロセスに任意のコードを実行させ られる

Alkanet が監視するシステムコール

挙動	システムコールの例
ファイル	NtCreateFile, NtReadFile, NtWriteFile, …
レジストリ	NtQueryValueKey,NtSetValueKey,…
仮想メモリ	NtWriteVirtualMemory, NtProtectVirtualMemory, …
ファイルマッピング	NtCreateSection, NtOpenSection, NtMapViewOfSection, …
プロセス	NtCreateProcessEx, NtTerminateProcess, …
スレッド	NtCreateThread,NtTerminateThread, NtSetContextThread, …
ネットワーク	NtDeviceIoControlFile, …
ドライバ	NtLoadDriver, NtUnloadDriver

Alkanet が監視するシステムコール

挙動	システムコールの例
ファイル	NtCreateFile, NtReadFile, NtWriteFile, …
レジストリ	NtQueryValueKey,NtSetValueKey,…
仮想メモリ	**NtWriteVirtualMemory, NtProtectVirtualMemory, …**
ファイルマッピング	**NtCreateSection, NtOpenSection, NtMapViewOfSection, …**
プロセス	NtCreateProcessEx, NtTermin…
スレッド	**NtCreateThread,NtTerminateThread, NtSetContextThread, …**
ネットワーク	NtDeviceIoControlFile, …
ドライバ	NtLoadDriver, NtUnloadDriver

コードインジェクション に用いられる システムコールへ対応

メモリ書き込み/スレッド挿入の挙動

No.	6337
Time	689820579
Type	sysenter
Ret	- (-)
SNo.	115 (NtWriteVirtualMemory)
Cid	bc.304
Name	Polipos.exe
Note	PID: b0, ProcessName: explorer.exe

explorer.exe の メモリ空間に書き込み

No.	6338
Time	689820647
Type	sysexit
Ret	0 (STATUS_SUCCESS)
SNo.	115 (NtWriteVirtualMemory)
Cid	bc.304
Name	Polipos.exe
Note	PID: b0, ProcessName: explorer.exe

成功

No.	6339
Time	689820849
Type	sysenter
Ret	- (-)
SNo.	35 (NtCreateThread)
Cid	bc.304
Name	Polipos.exe
Note	PID: b0, ProcessName: explorer.exe

explorer.exe へ スレッド作成

No.	6340
Time	689820959
Type	sysexit
Ret	0 (STATUS_SUCCESS)
SNo.	35 (NtCreateThread)
Cid	bc.304
Name	Polipos.exe
Note	Cid: b0.1e8, ProcessName: explorer.exe

成功

作成された スレッドのID

DLL ロードの挙動

No.	1334
Time	110380881
Type	sysexit
Ret	0 (STATUS_SUCCESS)
SNo.	6c (NtMapViewOfSection)
Cid	7a4.7bc
Name	explorer.exe
Note	PID: 7a4, ProcessName: explorer.exe, Flags: 10000a0 (File | HadUserReference | Image), BaseAddress: 76930000, SectionOffset: 0, ViewSize: 8000, FileName: ¥WINDOWS¥system32¥linkinfo.dll, Protect: 4 (PAGE_READWRITE)

DLL(セクションオブジェクト)を 仮想アドレス空間にマップ

No.	1338
Time	110381872
Type	sysexit
Ret	0 (STATUS_SUCCESS)
SNo.	89 (NtProtectVirtualMemory)
Cid	7a4.7bc
Name	explorer.exe
Note	PID: 7a4, ProcessName: explorer.exe, BaseAddress: 76931000, ProtectSize: 1000, NewProtect: 20 (PAGE_EXECUTE_READ), OldProtect: 4 (PAGE_READWRITE)

マップした領域に 実行権限を付与

スレッドの追跡

```
No. [5212, 5213]: Polipos.exe (Cid: 54c.18c) -> svchost.exe (Cid: 480.2c4) (Code Injection)
  No. [5288, 5289]: svchost.exe (Cid: 480.2c4) -> svchost.exe (Cid: 480.22c)
   No. [5959, 5960]: svchost.exe (Cid: 480.22c) -> svchost.exe (Cid: 480.38c)
   No. [6392, 6393]: svchost.exe (Cid: 480.22c) -> svchost.exe (Cid: 480.360)
   No. [11340, 11341]: svchost.exe (Cid: 480.22c) -> svchost.exe (Cid: 480.720)
    No. [14368, 14369]: svchost.exe (Cid: 480.720) -> rundll32.exe (Cid: 220.7f8) (Code Injection)
     No. [14546, 14547]: rundll32.exe (Cid: 220.7f8) -> rundll32.exe (Cid: 220.488)
     ...
   No. [11844, 11845]: svchost.exe (Cid: 480.22c) -> svchost.exe (Cid: 480.24c)
    No. [15080, 15081]: svchost.exe (Cid: 480.24c) -> alg.exe (Cid: 34c.1c8) (Code Injection)
     No. [15240, 15241]: alg.exe (Cid: 34c.1c8) -> alg.exe (Cid: 34c.5ac)
     ...
   No. [13214, 13215]: svchost.exe (Cid: 480.22c) -> svchost.exe (Cid: 480.7e0)
    No. [16586, 16587]: svchost.exe (Cid: 480.7e0) -> explorer.exe (Cid: 538.510) (Code Injection)
     No. [16744, 16745]: explorer.exe (Cid: 538.510) -> explorer.exe (Cid: 538.6ac)
     ...
   No. [13802, 13803]: svchost.exe (Cid: 480.22c) -> svchost.exe (Cid: 480.308)
   No. [14422, 14423]: svchost.exe (Cid: 480.22c) -> svchost.exe (Cid: 480.2d0)
   ...
```

- システムコールのログから，別プロセスへ挿入された スレッドを追跡可能であることを確認
- ログ分析ツールを利用することで， マルウェアの挙動をさらに理解しやすくできる

スレッドの追跡

Polipos.exe が svchost.exe にスレッド作成

スレッドが派生

さらに別のプロセスへ感染

```
No. [5212, 5213]: Polipos.exe (Cid: 54c.18c) -> svchost.exe (Cid: 480.2c4) (Code Injection)
  No. [5288, 5289]: svchost.exe (Cid: 480.2c4) -> svchost.exe (Cid: 480.22c)
   No. [5959, 5960]: svchost.exe (Cid: 480.22c) -> svchost.exe (Cid: 480.38c)
   No. [6392, 6393]: svchost.exe (Cid: 480.22c) -> svchost.exe (Cid: 480.360)
   No. [11340, 11341]: svchost.exe (Cid: 480.22c) -> svchost.exe (Cid: 480.720)
    No. [14368, 14369]: svchost.exe (Cid: 480.720) -> rundll32.exe (Cid: 220.7f8) (Code Injection)
     No. [14546, 14547]: rundll32.exe (Cid: 220.7f8) -> rundll32.exe (Cid: 22
     ...
   No. [11844, 11845]: svchost.exe (Cid: 480.
    No. [15080, 15081]: svchost.exe (Cid: 480.24c) -> alg.exe (Cid: 34c.1c8) (Code Injection)
     No. [15240, 15241]: alg.exe (Cid: 34c.1c8) -> alg.exe (Cid: 34c.5ac)
     ...
   No. [13214, 13215]: svchost.exe (Cid: 480.22c) -> svchost.exe (Cid: 480.7e0)
    No. [16586, 16587]: svchost.exe (Cid: 480.7e0) -> explorer.exe (Cid: 538.510) (Code Injection)
     No. [16744, 16745]: explorer.exe (Cid: 538.510) -> explorer.exe (Cid: 538.6ac)
     ...
   No. [13802, 13803]: svchost.exe (Cid: 480.22c) -> svchost.exe (Cid: 480.308)
   No. [14422, 14423]: svchost.exe (Cid: 480.22c) -> svchost.exe (Cid: 480.2d0)
   ...
```

- システムコールのログから，別プロセスへ挿入された スレッドを追跡可能であることを確認
- ログ分析ツールを利用することで， マルウェアの挙動をさらに理解しやすくできる

コードインジェクションの挙動の流れ

1. スレッド作成
   108.118 Polipos.exe **NtCreateThread** Cid: 370.11c, ProcessName: winlogon.exe => STATUS_SUCCESS
2. スレッドのコンテキスト取得
   108.118 Polipos.exe **NtGetContextThread** Cid: 370.11c, ProcessName: winlogon.exe => STATUS_SUCCESS
3. メモリ確保 & 権限設定
   108.118 Polipos.exe **NtAllocateVirtualMemory** Pid: 370, ProcessName: winlogon.exe, BaseAddress: 0xd90000, Protect: 0x40 (PAGE_EXECUTE_READWRITE),… => STATUS_SUCCESS
   108.118 Polipos.exe **NtProtectVirtualMemory** Pid: 370, ProcessName: winlogon.exe, BaseAddress: 0xd90000, NewProtect: 0x40 (PAGE_EXECUTE_READWRITE),… => STATUS_SUCCESS
4. メモリ書き込み
   108.118 Polipos.exe **NtWriteVirtualMemory** Pid: 370, ProcessName: winlogon.exe => STATUS_SUCCESS
5. スレッドのコンテキスト設定
   108.118 Polipos.exe **NtSetContextThread** Cid: 370.11c, ProcessName: winlogon.exe => STATUS_SUCCESS
6. スレッドの実行開始
   108.118 Polipos.exe **NtResumeThread** Cid: 370.11c, ProcessName: winlogon.exe => STATUS_SUCCESS

サービスとして起動する挙動

1. マルウェアがファイルを作成

78.80 Allaple.exe **NtCreateFile ¥??¥C:¥WINDOWS¥system32¥urdvxc.exe** => STATUS_SUCCESS
78.80 Allaple.exe NtWriteFile ¥WINDOWS¥system32¥urdvxc.exe => STATUS_SUCCESS

2. services.exe がレジストリを設定

39c.230 **services.exe NtCreateKey** ¥REGISTRY¥MACHINE¥SYSTEM¥CONTROLSET001¥**SERVICES¥MSWINDOWS** => STATUS_SUCCESS
39c.230 **services.exe** NtSetValueKey ¥REGISTRY¥MACHINE¥SYSTEM¥CONTROLSET001¥SERVICES¥MSWINDOWS, **Type = 0x110** => STATUS_SUCCESS
39c.230 **services.exe** NtSetValueKey ¥REGISTRY¥MACHINE¥SYSTEM¥CONTROLSET001¥SERVICES¥MSWINDOWS, **ImagePath = "C:¥WINDOWS¥system32¥urdvxc.exe" /service**, => STATUS_SUCCESS
...

3. services.exe が独立したプロセスとして サービス起動

39c.4cc **services.exe NtCreateProcessEx** Pid: a0, ProcessName: urdvxc.exe, **¥WINDOWS¥system32¥urdvxc.exe** => STATUS_SUCCESS

ドライバのロードの挙動

1. マルウェアがファイルを作成

```
158.170 smsg.exe    NtCreateFile ¥??¥C:¥WINDOWS¥system32¥drivers¥sysdrv32.sys
                    => STATUS_SUCCESS
158.170 smsg.exe    NtWriteFile  ¥WINDOWS¥system32¥drivers¥sysdrv32.sys => STATUS_SUCCESS
```

2. services.exe がレジストリを設定

```
39c.2f0 services.exe NtCreateKey ¥REGISTRY¥MACHINE¥SYSTEM¥CONTROLSET001¥SERVICES¥SYSDRV32
                    => STATUS_SUCCESS
39c.2f0 services.exe NtSetValueKey ¥REGISTRY¥MACHINE¥SYSTEM¥CONTROLSET001¥SERVICES¥SYSDRV32,
                    Type = 0x1 => STATUS_SUCCESS
39c.2f0 services.exe NtSetValueKey ¥REGISTRY¥MACHINE¥SYSTEM¥CONTROLSET001¥SERVICES¥SYSDRV32,
                    ImagePath="¥??¥C:¥WINDOWS¥system32¥drivers¥sysdrv32.sys"
                    => STATUS_SUCCESS
...
```

3. services.exe がドライバをロード

```
39c.2f0 services.exe NtLoadDriver ¥Registry¥Machine¥System¥CurrentControlSet¥Services¥sysdrv32
                    => STATUS_SUCCESS
```

立命館大学 19 2012年 9月 11日

まとめ

- Alkanet
  - VMM を用いてアンチデバッグを回避する
  - スレッド単位でマルウェアを追跡し，システムコールをトレースする
  - ログを元にマルウェアの特徴的な挙動を抽出するツール群も作成
  - 別プロセス内に作成されたスレッドも追跡可能であることを確認
- 今後の課題
  - ネットワークを用いた挙動を観測する機能の強化
    - 別のコンピュータに攻撃させないように配慮する必要がある
    - 既存のハニーポットとの連携
  - システムコールトレースログを元に
    異常検知やクラスタリングを行う既存手法に
    Alkanet のログが利用できるか評価を行う

立命館大学 20 2012年 9月 11日

ソフトウェアによるメモリエラーの検出と訂正

若林 大晃 †
† 立命館大学大学院理工学研究科

1 はじめに

1 年間に何らかのメモリエラーが発生する確率は，常時稼働しているサーバ（1–4GB の DIMM を複数搭載）の場合，計算機ごとに約 32.2% であるという報告がされている [1]．ECC メモリでは，メモリエラーの検出・訂正が可能であるが，一般向けに広く普及している Non-ECC メモリでは，メモリエラーの検出は困難である．そのため，メモリエラーの発生に気付かずに処理を続けると，重要なデータを失ったことに気付かない可能性がある．

以上の背景から，Non-ECC メモリのエラーに起因する重要なデータの破損を防ぐ研究を行っている．具体的には，アプリケーションのコンパイル時に，メモリエラーを検出するコード（命令）を加えることで，データの破損を検出し，訂正することで実現する．

2 メモリエラーと対策

メモリエラーが発生すると，メモリに記憶していた値が破損してしまう．メモリエラーには，ソフトエラーとハードエラーの 2 種類がある（表 1）．ソフトエラーは，一時的に発生するエラーであり，正しい値に書き直すことで修復することができる．一方，ハードエラーは，ハードウェアの故障が原因のため，正しい値を書き直しても修復はできず，同じアドレスで何度もエラーが発生する．

メモリエラーから復帰するためには，メモリエラーを検出し，正しい値に復元する必要がある．これは，メモリエラーを検出・訂正するためのコードとデータが必要であること意味する．そのため，検出・訂正用のコードとデータを作成することが大きな課題である．

3 メモリエラーの修復における課題点

メモリエラーを修復するためには，以下の二つの課題を解決する必要がある．

1. エラー検出・訂正アルゴリズム（コードとデータ）
2. エラー検出・訂正コードの付加手法

3.1 エラー検出・訂正アルゴリズム

課題点 1 については，実現した場合にかかる計算量を考慮して，適切なアルゴリズムを選択する必要がある．例えば，以下のようなアルゴリズムが考えられる（表 2）．

- データの複製（検出）: 保護データの複製を作ることで，エラーの検出が可能
- データの複製（検出・訂正）: 保護データの複製を複数作ることで，エラーの訂正が可能
- ハミング符号: ECC メモリでも用いられているエラー訂正符号の一つであり，2 ビットまでのエラー検出もしくは 1 ビットのエラー訂正が可能

表 1　メモリエラーの種類

種類	持続時間	原因
ソフトエラー	一時的	α 線や中性子線
ハードエラー	永続的	ハードウェアの故障

表 2　エラー検出・訂正アルゴリズムの例

種類	検出	訂正	データ 8bit に必要なデータ量
データの複製（検出）	○	×	8bit
データの複製（検出・訂正）	○	○	16bit 以上
ハミング符号	2bit	1bit	4bit

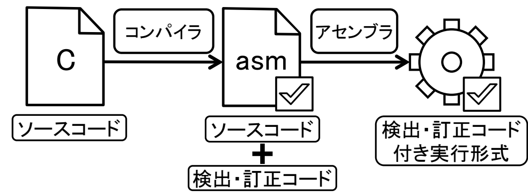


図 1　メモリエラー検出・訂正コードの生成機構

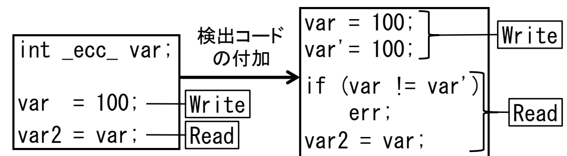


図 2　メモリエラーの検出コードを付加した例

計算量の多いアルゴリズムは，オーバーヘッドによる実行効率の低下が懸念される．そのため，実行効率と機能のトレードオフを考慮して選択しなければならない．また，アルゴリズムごとに要するデータ量が異なるため，おのおのに適したメモリレイアウトが必要である．エラー検出・訂正データは，アプリケーションが利用する領域と分離して確保し，保護するデータと対応付ける必要がある．

3.2 エラー検出・訂正コードの付加

メモリエラーの検出・訂正を行うため，一度メモリからレジスタに読み出したデータは，メモリを経由せずに用いなければならない．そのためには，適切にレジスタを割り当てる必要がある．

4 コード挿入によるメモリエラーの検出・訂正手法

本研究では，アプリケーションにメモリエラーを検出・訂正する機能を加えることで，データの保護を実現する．検出・訂正コードは，アプリケーションのコンパイル時に自動的に付加

表 3　オーバヘッド計測環境

項目	内容
CPU	Intel Core-i7 920 2.67GHz
メモリ	6GB
OS	Debian GNU/Linux 6.0 Squeeze
カーネル	Linux 2.6.32-5

表 4　オーバヘッドの見積り

READ	データの複製	約 2 倍
WRITE	データの複製とハミング符号	約 26 倍

する（図 1）．具体的には，ソースコードに記述されている変数に対して，重要であるという印（図 2 左，_ecc_）を付け，その変数を扱う命令にエラー検出・訂正用のコードを加える（図 2 右）．これにより，アプリケーションの開発者は，保護したい変数に印を付けることで，変数をメモリエラーから保護することができる．

エラー検出・訂正コードは，変数を READ/WRITE する命令に付加する．実際に付加するコードは，アルゴリズムによって異なるが，図 2 の例では，WRITE 時にデータの複製を作成し，READ 時に比較することでメモリエラーの検出を行っている．

5　オーバヘッドの見積り

5.1　計測内容

本手法を適用した場合にかかる実行時のオーバヘッドを見積もるために，エラー検出・訂正アルゴリズムを実装したプログラムを作成し，メモリエラーが発生しない場合の READ/WRITE アクセスにかかるオーバヘッドを計測した．作成した計測プログラムは，検出に「データの複製」，訂正に「ハミング符号」を用いるもので，1.5GB のメモリを下位アドレスから上位アドレスに向けて，順に READ/WRITE アクセスを行う．READ アクセスは，文字定数'c' で初期化したメモリを 64 バイト毎に読み出す処理を 64 回繰り返した（キャッシュミスを発生させるため）．WRITE アクセスは，あらかじめ確保しておいたメモリに，1 から順番に値を書き込み，同時にエラー検出・訂正用データ（「データの複製」と「ハミング符号」）の作成を行う．検出と訂正におのおの別のアルゴリズムを用いる意図として，READ アクセス時のオーバヘッドを軽減する目的がある．

5.2　計測結果

表 3 の環境で計測を行った結果，検出・訂正アルゴリズムを実装してないプログラムと比較して，WRITE アクセスのオーバヘッドは約 26 倍，READ アクセスのオーバヘッドは約 2 倍であった（表 4）．このことから，WRITE アクセスが少なく，READ アクセスが多いアプリケーションでは，本手法は有用であると考えられる．

6　今後の研究計画

研究は以下の三つの大きな流れに沿って進めている．

(1) 該当箇所に命令を挿入するための機構の作成
(2) アルゴリズムの実装と検証
(3) 全体評価

現在の進捗状況は，計画 (1) が完了し，計画 (2) を進めている段階である．アルゴリズムの実装として，「データの複製（検出）」と「ハミング符号」が限られた文法において動作している．

今後は，メモリエラー検出・訂正アルゴリズムを実装し，その評価を行う．これにより，有用なアルゴリズムの選別を行い，C 言語の完全な文法で利用できるように実装を行う．その後，本機構の全体的な評価を行い，修士論文としてまとめる予定である．

7　関連研究

ソフトエラーを検出することを目的とした研究は，既存の研究においても行われており，EDDI[2] や Fault-tolerant typed assembly language[3] があげられる．EDDI は，アプリケーションのコンパイル時に命令とデータ領域を複製することで，ソフトエラーの検出を実現している．また，Fault-tolerant typed assembly language は，ソフトウェアとハードウェアのハイブリッド手法であり，ソフトエラー検出を効率的に行うためのハードウェアを定義し，そのハードウェアを用いて多重化した命令の実行を行う．

これらの研究は，メモリエラーだけでなく，あらゆる回路で発生するソフトエラーを対象としているため，全ての命令に対して，多重化のような検出手法が必要である．本研究では，特にメモリエラーに着目することで，特定のメモリを扱う命令のみを対象としている．これにより，全ての命令を対象とした場合に実現が難しいような，より強力なメモリエラー検出・訂正アルゴリズムの実現や，オーバヘッドの削減が期待できる．

8　おわりに

本稿では，メモリエラーによる重要なデータの破損の検出と訂正手法について述べた．データの保護は，アプリケーションにメモリエラーの検出・訂正を行う機能を付加することで実現する．エラー検出・訂正機能は，アプリケーションのコンパイル時に自動的に付加する．これにより，アプリケーションの開発者は，重要な変数の型に印をつけることで，変数をメモリエラーから保護することができる．

現在は，メモリエラー検出・訂正アルゴリズムの実装を行っている段階である．今後は，この実装と評価を行い，有用なアルゴリズムの選別と実装を行い，全体的な評価を行う．

参考文献

[1] Bianca Schroeder, Eduardo Pinheiro and Wolf-Dietrich Weber: “DRAM errors in the wild: a large-scale field study,” In Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems, pp. 193–204 (2009).

[2] N. Oh, P.P. Shirvani and E.J. McCluskey: “Error detection by duplicated instructions in super-scalar processors,” Reliability, IEEE Transactions on, Vol. 51, No. 1, pp. 63–75 (2002).

[3] Frances Perry, Lester Mackey, George A. Reis, Jay Ligatti, David I. August and David Walker: “Fault-tolerant typed assembly language,” SIGPLAN Not., Vol. 42, No. 6, pp. 42–53 (2007).

ソフトウェアによるメモリエラーの検出と訂正

立命館大学大学院
毛利研究室
M2 若林 大晃

目次

- はじめに
- メモリエラー
  - 概要
  - 検出・訂正の対処方法と課題
- メモリエラーの検出・訂正機構
- 実装
- 関連研究
- おわりに

はじめに (1/2)

- 1年間にメモリエラーが発生する確率は約32.2%(*)
  - 24時間稼働するサーバで，1～4GBのメモリを複数搭載
- メモリエラーの問題点
  - メモリ上の値が破損する
  - メモリエラーに気付かずに処理を続ければ，
    重要なデータを失ったことに気付かない可能性がある
- メモリエラーへの対処方法
  - サーバ，組込み機器: ECCメモリ

(*) Bianca Schroeder, Eduardo Pinheiro and Wolf-Dietrich Weber: "DRAM errors in the wild: a large-scale field study"

はじめに (2/2)

- 一般向けのPCの場合
  - メモリエラーは考慮されていない
    - 必要な人は個人でECCメモリを搭載する等
    - ECCメモリが利用できないこともある
  - ECCメモリは高価なため，全てのPCに搭載できない

⇒ ソフトウェアで対処することで，特殊なハードウェアなしでメモリエラーの検出・訂正が可能

ソフトウェアを用いたメモリエラー検出・訂正により，誤ったデータを用いて動作し続けることを防止する

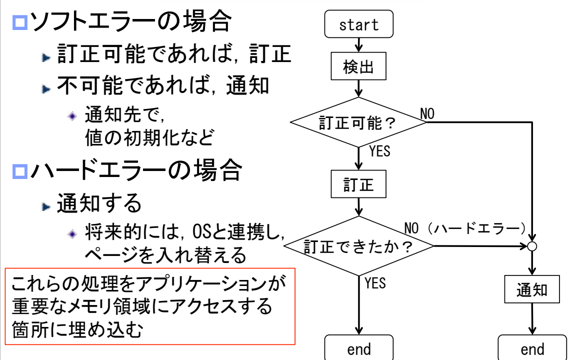
メモリエラー

- メモリエラー
  - メモリに記憶していた値の破損の原因
- 発生要因
  - ソフトエラー: α線や中性子線が当たる（一時的なエラー）
  - ハードエラー: メモリを構成する素子の故障（永続的なエラー）
- 対策
  - ソフトエラー
    - 正しい値に書き直すことで，修復することができる
  - ハードエラー
    - 常にエラーが発生するため，修復は不可能
    - エラーのたびに訂正，
      または別の領域を用いることで動作を続けられる

メモリエラーの対処

- ソフトエラーの場合
  - 訂正可能であれば，訂正
  - 不可能であれば，通知
    - 通知先で，
      値の初期化など
- ハードエラーの場合
  - 通知する
    - 将来的には，OSと連携し，
      ページを入れ替える

これらの処理をアプリケーションが重要なメモリ領域にアクセスする箇所に埋め込む



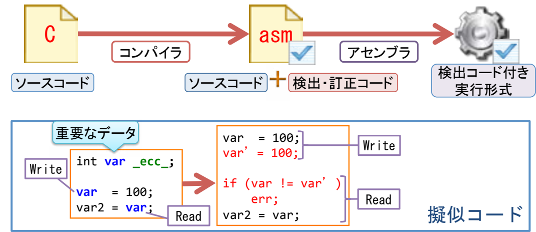
検出・訂正の課題

- 検出・訂正用のコード(命令)の作成
  - メモリエラーの検出・訂正を行うため, 一度読み出した値を書き戻さないように, 検出・訂正を行う必要がある.
  - 適切にレジスタを割り付ける方法が必要
- データ領域の割当て
  - 領域の確保: 検出・訂正用のメモリを確保する方法
  - 対応付け: データと検出・訂正用データを対応付ける方法
- アルゴリズムの選択と適用方法
  - どのようにしてアルゴリズムを適用するか

種類	検出	訂正	8bitのデータに必要な領域
値の複製(検出)	○	×	8bit
値の複製(検出・訂正)	○	○	16bit以上
ハミング符号	2bit	1bit	4bit

メモリエラー検出・訂正機構

- メモリエラー検出・訂正機能をアプリケーションに付加する
  - コンパイラを用いて自動的に付加する
    - 型または変数に「重要マーク」をつける
    - その変数を扱う命令に, 検出・訂正用のコードを付加する
  - プログラマは, 重要なデータを扱う変数に「重要マーク」をつける



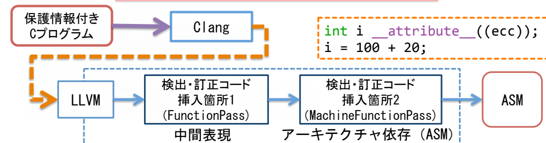
全体的な研究計画

1. 検出・訂正コードを挿入する機構の作成 …… 完了
   - 各アルゴリズムの検証に利用
2. アルゴリズムの実装と検証 …… 途中
   - 実際にアプリケーションに適応し, 検証
     - メモリレイアウト, オーバヘッド, …
   - 有用なアルゴリズムの選別
     - Cの完全な文法で扱えるように実装を行う
3. 訂正不可能なエラーの処理方法の検討・実装
   - 現在は, エラーハンドラの呼び出し
4. 全体評価

実装環境と実装内容

- 実装環境

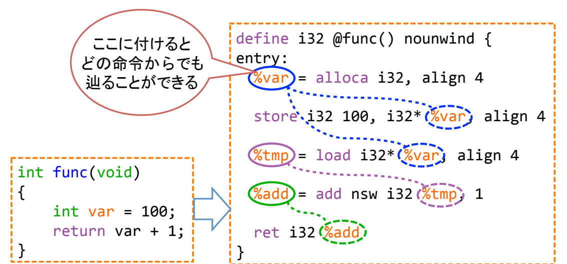
項目	内容
コンパイラ	Clang 2.9, LLVM 2.9
アーキテクチャ	Intel 64



- 命令挿入機構の実装
  - 実装1: Cプログラムの_ecc_の解析→中間表現に印付け
  - 実装2: 中間表現レイヤで_ecc_付き命令に命令を追加
  - 実装3: ASMレイヤで_ecc_付き命令に命令を追加

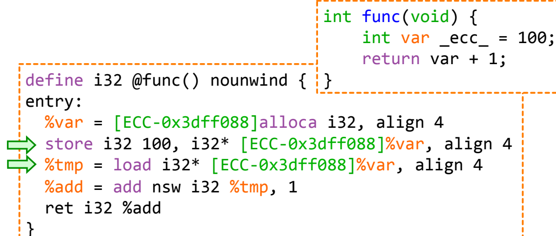
実装: Cプログラムの_ecc_の解析

- Clangを修正
  - __attribute__((ecc))の解析を可能にする
  - 中間コード生成器で, データ構造に_ecc_情報を付ける



実装: 命令追加(中間表現部)

- LLVMを修正
  - Function Passで, _ecc_が付いているLOAD/STORE命令を探す
    - 自力で探索する
  - その命令の直前・直後に命令を挿入
    - 命令挿入の仕組みはLLVMの機能を利用



実装： 命令追加（ASM部）

- LLVMを修正
  - ASMの各命令と，基になった中間表現命令との関係があるため，それを利用する

```
define i32 @func() nounwind {
entry:
  %var = [ECC-0x3dff088]alloca i32, align 4
  store i32 100, i32* [ECC-0x3dff088]%var, align 4
  %tmp = load i32* [ECC-0x3dff088]%var, align 4
  %add = add nsw i32 %tmp, 1
  ret i32 %add
}
```

```
MOV32mi %RSP, 1, %noreg, -4, %noreg, 100
        ; mem:ST4[[ECC-0x3dff088]%var]
%EAX<def> = MOV32ri 101
RET %EAX<imp-use,kill>
```

実行例： 生成されたASMのdiff

- 中間表現レイヤで命令挿入 → ASM

```
int var _ecc_;

int main(void) {
    var = 100;
    return 0;
}
```

```
--- before.s
+++ after.s
@@ -8,6 +8,11 @@
 # BB#0:                              # %entry
        movl    $0, -4(%rsp)
        xorl    %eax, %eax
+       movl    $var, %ecx
+       andq    $2097151, %rcx        # imm = 0x1FFFFF
+       movabsq $68719476736, %rdx    # imm = 0x1000000000
+       orq     %rcx, %rdx
+       movl    $100, (%rdx)
        movl    $100, var(%rip)
        ret
 .Ltmp0:
```

複製を作る
アドレスの計算
複製を作る

今後の展望

- メモリエラー検出・訂正アルゴリズムの実装
  - 有効なアルゴリズムを選別・実装
  - 訂正不可能エラー時の処理の検討・実装
  - 全体的な評価
- 想定されるメモリアクセスのオーバヘッド
  - メモリアクセスが多いアルゴリズムの場合
    - READ/WRITE： 約26倍
      （このうち，98%がECC用領域とアプリ領域の関連付け）
  - READ時に最適化した場合
    - READ（検出時最適化）： 2～3倍

アプリケーション全体として見た場合は，
オーバヘッドはさらに小さいものになる

関連研究

- EDDI
  - ソフトウェア多重化
  - 命令と記憶領域を全て多重化する

```
 ld r12=[GLOBAL]
+ld r22=[GLOBAL+offset]
 add r11=r12,r13
+add r21=r22,r23
+cmp.neq.unc p1,p0=r11,r21
+cmp.neq.or p1,p0=r12,r22
+(p1) br faultDetected
```

記憶領域も
多重化

- Fault-tolerant Typed Assembly Language
  - ソフトウェアとハードウェア（CPU）による多重化
  - CPUの動作を厳密に定義し，かつ命令多重化のための
    レジスタを搭載することで，高速なエラー検出を実現
- 提案手法との違い
  - 提案手法では，メモリエラーに着目
  - より強力な検出・訂正の実現やオーバヘッド削減が期待

おわりに

- メモリエラー検出・訂正機構
  - アプリケーションにメモリエラー検出・訂正のための命令を付加
    - コンパイラを用いて自動的に行う
    - C言語の型や変数に印を付け，その変数を扱う命令に
      検出・訂正コードを付加する
- 現在の進捗
  - メモリエラー検出・訂正コードの挿入機構を実装
    - 保護する変数を扱う命令に対して検出・訂正コードを挿入する
  - アルゴリズムの実装中
- 今後の予定
  - アルゴリズムの実装と検証
  - 全体的な評価

管理 VM 監視のためのメモリアクセス通知機構の開発

猪飼淳 †　　　齋藤彰一 †
† 名古屋工業大学

1　はじめに

IaaS 型クラウドにおいてユーザが利用する計算資源は仮想マシン (VM) であり，VM はサービス提供者の管理 VM によって管理される．クラウド管理者は管理 VM の権限を行使することでユーザ VM 内のメモリ内の情報を覗き見ることが可能であり，ユーザ VM 内のメモリ上の機密情報が漏洩する可能性がある．

この問題を解決するために管理 VM がユーザ VM へメモリをアクセスする際にメモリを暗号化する手法 [1] が提案されているが，ユーザはメモリが管理 VM に覗かれたことを検知する術を持たないため，ユーザは情報漏洩が発生する環境にいることを認識することはできない．

本研究では仮想マシンモニタ (VMM) が管理 VM のユーザ VM へのメモリの覗き見を検知し，ユーザ VM へ通知するシステムを提案する．ユーザ VM は管理 VM による覗き見を禁止するメモリ領域を指定することができる．

2　管理 VM による情報漏洩

IaaS 型クラウドにおいてユーザはネットワークを経由して VM にアクセスを行う．この際ユーザがアクセスする計算資源はクラウドサービスを提供するクラウド管理者によって管理されるため，ユーザは計算資源のメンテナンスから解放されるメリットを享受することができる．

ユーザ VM はクラウド管理者が保有する管理 VM によって管理される．管理 VM はハイパーコールと呼ばれる管理 VM に与えられた特権命令を用いて，VM の起動，終了，サスペンド，マイグレーションといった操作を行い，ユーザ VM の管理を行う．

上述した操作を行う際，管理 VM はユーザ VM のメモリへアクセスする必要がある．その際，管理 VM はメモリマップ用ハイパーコールを実行し，ユーザ VM のメモリへアクセスを行う．管理 VM はメモリマップ用ハイパーコールを用いることによりユーザ VM 内の全てのメモリ領域にアクセスすることが可能であり，権限を過剰に利用することでユーザ VM 内のメモリ上の機密情報が漏洩する可能性がある．

3　既存手法 VMCrypt

3.1　概要

VMCrypt はユーザ VM のメモリを暗号化することにより，管理 VM へのメモリ内の機密情報の漏洩を防止する手法である．VMCrypt は管理 VM がユーザ VM のメモリへアクセスしようとした際，つまりメモリマップ用ハイパーコールを実行した際に VMM がユーザ VM のメモリを暗号化し，管理 VM には暗号化したメモリを見せる．一方，ユーザ VM が自身のメモリへアクセスを試みた際は暗号化を行わず，メモリをそのまま見せる．

3.2　問題点

VMCrypt の問題点としてユーザ VM 自身がメモリが覗かれたことを知ることができないという点が挙げられる．VMCrypt ではメモリ覗き見があったことを検知するのは VMM でありユーザ VM は一切関与しない．そのため，ユーザ VM は情報漏洩の可能性のある環境にいることを認識することができない．

また，管理 VM によるメモリアクセスは必ずしも不正行為にのみ用いられるわけではなく，管理 VM からユーザ VM 内の rootkit を検出するシステムといったセキュリティ向上のためにアクセスするケースが存在する．これらのシステムはメモリを暗号化することによって動作不可能になる．

4　提案手法

4.1　概要

管理 VM によるメモリアクセスをユーザが認識するためにメモリアクセス通知機構を提案する．これにより VMM が管理 VM によるメモリマップ用ハイパーコールを検知した際，ハイパーコール実行時のパラメータをユーザ VM に通知することでユーザ VM は VMM か通知された情報を見ることで管理 VM の動向を知ることができる．

また，ユーザ VM 自身が自身のメモリ上でアクセス不可能な領域を指定することができるメモリアクセス

Jun IKAI †　Shoichi SAITO †
†Nagoya Institute of Technology

制御機構を提案する．これによりユーザはメモリアクセス制御ポリシを工夫することで管理 VM と協調するセキュリティシステムを動作させることが可能になる．

4.2 メモリマップの検知

管理 VM がユーザ VM 内のメモリへアクセスを行う場合，必ず管理 VM はメモリマップ用ハイパーコールを用いる．ハイパーコールは必ず一度 VMM を経由して実行されるため，VMM 内でメモリマップ用ハイパーコールの実行を監視することで，管理 VM がユーザ VM に対してメモリマップを行おうとしているかを検知することができる．

4.3 メモリアクセス通知機構

VMM が管理 VM によるユーザ VM へのメモリマップを検知した後，管理 VM の行ったメモリマップのログをユーザ VM に通知する．これは VMM と一般 VM への共有メモリを作成することによって実現される．VMM はメモリマップ検知を行った際に共有メモリに書き込み，ユーザ VM は共有メモリからログを読み取ることによって管理 VM によるメモリマップがあったことを知ることができる．また，管理 VM によるログの改ざんを防止するため，作成する共有メモリは管理 VM によってアクセスされないようにする必要があり，この仕組みは以下で述べるメモリマップ拒否リストによって実現される．

4.4 メモリアクセス制御機構

VMM は管理 VM によるメモリマップ用ハイパーコールの実行を検知した際にハイパーコールを失敗させることでユーザ VM へのメモリアクセスを禁止することができる．しかし，ユーザ VM へのメモリアクセス全てを禁止した場合，サスペンドやマイグレーションといった IaaS 運営上必須な管理操作や，セキュリティシステムの管理 VM へのオフロード手法を正常に動作させることが不可能になる．そのため，ユーザ VM は VMM に対して管理 VM によるメモリマップを拒否したい領域を指定し，ユーザ VM が管理 VM に対してアクセスを禁止できるようにする．これはメモリマップログの通知と同様に VMM とユーザ VM 間の共有メモリによって実現される．VMM は管理 VM によるメモリマップ用ハイパーコールを検知した際，共有メモリを探索し，メモリマップが禁止されているページである場合とログ通知用の共有メモリ領域である場合はメモリマップを失敗させる．

表 1: 実験環境

VMM	Xen 4.1.2
管理 VM	Linux 3.4.4/ Core-i3 2.4GHz 4core/ 7GB
ユーザ VM	Linux 3.4.4/ VCPU 1core/ 1GB/ 準仮想化

表 2: メモリマップのオーバヘッド

無修正版	$22\mu s$
提案手法	$29\mu s$

5 実験

実験環境を表 1 に示す．通知機構，制御機構の動作確認と管理 VM によるメモリマップ用ハイパーコールのオーバヘッドの測定の 2 つの実験を行った．

5.1 動作確認

管理 VM 上でユーザ VM 内にロードされているカーネルモジュールをスキャンするプログラムを実行した際の管理 VM 上の出力画面とユーザ VM 上のログを観察した．このプログラムを実行した際，管理 VM の画面にはユーザ VM のカーネルモジュール名の一覧が出力され，ユーザ VM のログには管理 VM がアクセスを行ったアドレスが出力されており，通知機構の動作が確認できた．

次に制御機構の動作を確認するため，ユーザ VM がカーネルモジュール管理構造体をアクセス不能にした．その結果，管理 VM のメモリマップ用ハイパーコールは失敗しカーネルモジュール名を画面に出力することはできなかった．以上により，制御機構の動作が確認できた．

5.2 オーバヘッドの測定

無修正版の Xen と提案手法でメモリマップ用ハイパーコールにかかるオーバヘッドを計測した．結果は表 2 のようになった．結果より，提案手法は無修正版と比べて $7\mu s$ のオーバヘッドがあり，これはメモリマップの検知やアクセス制御の判定にかかる時間であると考えられる．

6 まとめ

管理 VM によるユーザ VM 内のメモリアクセスをユーザ VM に通知するメモリアクセス通知機構と，ユーザ VM 自身が管理 VM にメモリアクセスを拒否する領域を指定するメモリアクセス制御機構を提案した．これらの手法によりユーザは管理 VM の動向を知ることができ，また，管理 VM と協調するセキュリティシステ

ムを動作させせることが可能になる．今後の課題としてサスペンド，マイグレーションに対応することが考えられる．

参考文献

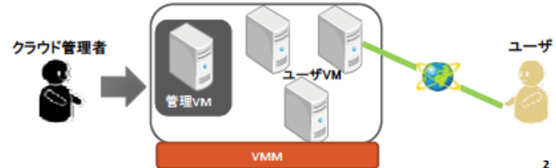
[1] Hidekazu Tadokoro, S. C., Kenichi Kourai: Preventing Information Leakage from Virtual Machines' Memory in IaaS Clouds (2012).

管理VM監視のための
メモリアクセス通知機構の開発

名古屋工業大学大学院
工学研究科 情報工学専攻 齋藤研究室
猪飼 淳

IaaS型クラウド

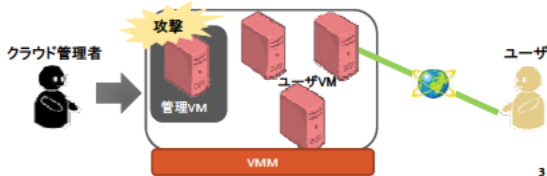
- 仮想マシン(VM)をインターネット経由でユーザに提供
- ユーザは計算資源のメンテナンスが不要
- クラウド管理者は管理VMからユーザVMを管理
  - VMの起動、終了
  - サスペンド
  - マイグレーション



2

セキュリティリスク

- クラウド管理者を信頼しなければならない
  - ユーザから見て管理者が信頼できるか判断するのは困難
- 管理VMの権限が強い
  - 管理VMが陥落するとユーザVMも陥落する
  - 管理者が過剰に権限を用いる可能性がある



3

管理VMの権限

- ユーザVMの操作
  - 起動、終了、サスペンド、マイグレーション
- ユーザVMの操作にはメモリアクセスを伴う
  - 管理VMはユーザVMのメモリへのアクセス権を持つ

- 問題点
  - 管理者はメモリアクセス権限の不正利用が可能
    - 管理者はユーザVM内のメモリを覗き見ることが可能
    - メモリ上には機密情報がある
      - パスワード、ファイルキャッシュ
  - メモリが覗かれたことを検知するのは困難

4

既存手法

- ユーザVMのメモリの暗号化†, ††
  - 管理VMに対して暗号化したメモリを見せる
    - 暗号化はVMMが行う
  - ユーザVMに対しては暗号化せずそのまま見せる

† A Trusted Virtual Machine in an Untrusted Management Environment [IEEE 2011]
†† Preventing Information Leakage from Virtual Machines' Memory in IaaS Clouds [ACS 2012]

5

既存手法の問題点

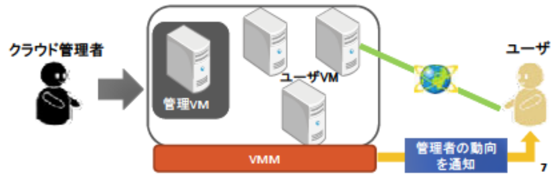
- 問題点1
  - ユーザVMはメモリが覗かれたことを知ることができない
    - メモリの覗き見があったことを検知するのはVMM
    - ユーザVMは漏えいの可能性のある環境にいることがわからない
- 問題点2
  - 管理VMによるメモリアクセスは必ずしも悪ではない
    - セキュリティ向上のためにアクセスするケース
      - 管理VMからユーザVM内のルートキットを検出するシステム†
      - 管理VMへセキュリティシステムをオフロードする研究††
    - メモリを暗号化するとオフロードしたシステム等は動作不能

† Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Reconstruction [ACM 2007]
†† VM Shadow: 既存IDSをオフロードするための実行環境 [情報処理学会 論文誌 2011]

6

提案1：メモリアクセス通知機構

- 管理VMによるメモリアクセスをユーザへ通知
  - ユーザVMは情報漏えいが発生したことを検知できる
  - 管理VMの信頼度を測る指標として用いる
- 通知機構の利用例
  - 情報漏えいの証拠として利用
  - アクセスログを用いた侵入検知システム



7

既存手法の問題点

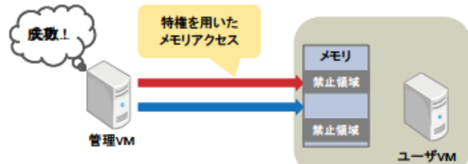
- 問題点1
  - ユーザVMはメモリが覗かれたことを知ることができない
    - メモリの覗き見があったことを検知するのはVMM
    - ユーザVMは漏えいの可能性のある環境にいることがわからない
- 問題点2
  - 管理VMによるメモリアクセスは必ずしも悪ではない
    - セキュリティ向上のためにアクセスするケース
      - 管理VMからユーザVM内のルートキットを検出するシステム†
      - 管理VMへセキュリティシステムをオフロードする研究††
    - メモリを暗号化するとオフロードしたシステム等は動作不能

† Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Reconstruction [ACM 2007]
†† VM Shadow: 既存IDSをオフロードするための実行環境 [情報処理学会 論文誌 2011]

8

提案2：メモリアクセス制御機構

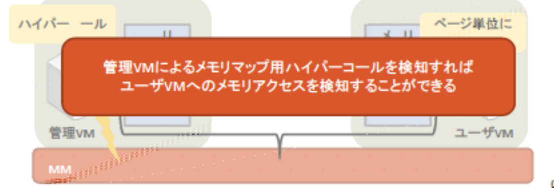
- ユーザVMからアクセス不可能の領域を指定可能
  - きめ細かいメモリアクセス制御ができる
  - 制御ポリシ次第で管理VMと協調するセキュリティシステムが動作可能
- 暗号化のコストが不要
  - メモリアクセス自体を失敗させるため



9

管理VMによるメモリアクセス

- メモリマップを行った後にユーザVMのメモリへアクセスする
  1. 管理VMはメモリマップ用ハイパーコールを発行しメモリマップを要求
  2. ハイパーコールによりVMMへ処理が移る
  3. VMMはページテーブルの更新操作を行いメモリマップをする
  4. 管理VMは自身のメモリ上にマップされたページに対してアクセスする



0

メモリマップの検知

- VMMがハイパーコールの発行を監視する
  - Xen Security Modules (XSM) を利用
    - Xenのセキュリティツール開発用フレームワーク
    - VMM内で発生するイベントに対してフックをかけることが可能
- ページテーブル更新操作をフック
  - 管理VMによるメモリアクセス関するものだけを取り出す
  - フック時にフック関数の引数を確認することで特定可能

11

実装1：メモリアクセス通知機構

- VMM – ユーザVM間の共有メモリによる通知
  1. 共有メモリとして使うメモリ空間はユーザVMが作成
  2. ハイパーコールを用いてVMMに共有メモリのアドレスを渡す
     - アドレスはVMMが理解できる形に変換する
  3. VMMは共有メモリにアクセスログを書き込む
  4. ユーザVMは共有メモリからアクセスログを読み取る

12

実装2 : メモリアクセス制御機構

- 通知機構と同じく共有メモリを用いる
  - ユーザVMはアクセス拒否する領域を共有メモリに書き込む
  - VMMはメモリマップ検知時に共有メモリを調べアクセス制御を行う
- …の追加・削除インタフェース
  - デバイスドライバ形式
  - 拒否アドレスの指定方法
    - アドレス指定によるページ単位の拒否
      - パスワードを書き込むバッファを確保した際にその領域をアクセス不可にする
    - プロセスIDによるプロセス管理構造体ページの拒否
      - SSHサーバのプロセス管理構造体をアクセス不可に

13

実験

- 実験環境

VMM	Xen 4.1.2
物理マシン	Core-i3 2.4GHz 4core / 8GB
管理VM	Linux 3.4.4 / VCPU 4core / 7GB
ユーザVM	Linux 3.4.4 / VCPU 1core / 1GB / 準仮想化

- 実験1: メモリアクセス通知とメモリアクセス制御の確認
  - 管理VM上でユーザVMのメモリを覗くプログラムを実行
    - ユーザVMにロードされているカーネルモジュールをスキャンするプログラム
  - 管理VMの上での出力結果とユーザVM上のログを観察
  - アクセス制御なし版とあり版で実験
    - アクセス制御はカーネルモジュール管理構造体をアクセス不能した場合

14

実験1: アクセス通知と制御の確認

- アクセス制御なし
  - 管理VMの出力結果

```
# ./scan_lkm

deny_list_logger
ip6t REJECT
nf_conntrack_ipv6
nf_defrag_ipv6
ip6table_filter
ip6_tables
nf_conntrack_ipv4
nf_defrag_ipv4
xt_state
nf_conntrack
coretemp
crc32c_intel
xen_blkfront
  .
  .
  .
```

  - ユーザVMのログ

メモリマップ試行回数 / 物理アドレス (MFN)

```
count:780 mfn:1cf8f5
count:781 mfn:1ca6df
count:782 mfn:1fac14
count:783 mfn:1fac10
count:784 mfn:139e4c
count:785 mfn:1cf98a
count:786 mfn:1ca6df
count:787 mfn:1fac14
count:788 mfn:1fac10
count:789 mfn:139e4c
count:790 mfn:1ca553
count:791 mfn:1ca6df
count:792 mfn:1fac14
      .
      .
      .
```

ページディレクトリの取得 (アドレス変換)

管理構造体の取得

15

実験1: アクセス通知と制御の確認

- アクセス制御あり
  - 管理VMの出力結果

```
# ./scan_lkm

Failed to map guest memory
```

  - ユーザVMのログ

メモリマップ試行回数 / 物理アドレス (MFN)

```
count:803 mfn:1ca6df
count:804 mfn:1fac14
count:805 mfn:1fac10
count:806 mfn:139e4c
count:807 mfn:1cf8f5 Denied
```

メモリマップを拒否

16

実験2 : オーバーヘッドの測定

- 管理VMによる1ページ当たりのメモリマップのコスト
  - 無修正版と提案手法実装版と比較

無修正版	22 μ s
提案手法(アクセス制御なし)	29 μ s
提案手法(アクセス制御あり)	29 μ s

- 考察
  - メモリマップの検知やアクセス制御の判定の分だけ負荷がかかっている
    - アクセス制御判定対象が増えるとオーバーヘッドが増大する可能性がある
  - 暗号化をする場合と比べると負荷は小さい
    - 暗号化した場合にかかる時間は無修正版の7倍程度

17

今後の展望(1/2)

- サスペンド、マイグレーションの対応
  - 暗号化が必須
    - メモリイメージがファイルとして管理VM上に置かれるため覗き見防止が必要
    - 復帰時には元のメモリの内容を復元する必要がある
  - サスペンド、マイグレーションの検知の実現
    - サスペンド、マイグレーションの検知を行いその際は暗号化
    - 通常時はメモリマップを失敗させるよう振る舞う

既存手法と比べて通常時の暗号化のコストの削減

18

今後の展望(2/2)

- ログの利用
  - ログを用いた侵入検知システム
    - ログからどのような攻撃を受けているか調べる
      - 例）管理VMからプロセススキャンを受けている等
  - サスペンド、マイグレーションの検知
- メモリアクセス以外にも拡張
  - VMMはメモリアクセス以外も管理VMの振る舞いを検知できる
    - スケジュールポリシーの変更
    - メモリ割り当て量の変更
  - Service Level Agreement を満足しているかユーザが確認するシステム

19

まとめ

- 管理VMによるメモリアクセスを通知するシステムを提案
  - ユーザVMが管理者の動向を知ることが可能になる
  - メモリアクセスの検知を行うのはVMMでVMMは信頼する
  - 通知機構はVMM – ユーザVM間の共有メモリを用いて実現
  - 管理VMがユーザVM内の情報を盗むプログラムで通知の実現を確認
- アクセス制御機構を実装
  - ユーザVMが管理VMからアクセスされたくない領域を指定
  - ユーザプログラムからアクセス不可領域を追加・削除
  - メモリ暗号化と比べて少ないコストでメモリの保護が可能

20

ユーザ関数のカーネル内実行を可能とした非同期カーネルサービスインタフェースの実現

安井 裕亮 †　　齋藤 彰一 †
† 名古屋工業大学

1 はじめに

従来の割り込みによるシステムコールは性能低下を引き起こすことが指摘されている．このような問題点に対して，割り込みを必要としないシステムコールである FlexSC[1] が提案されている．

本研究では，この FlexSC をベースに，ユーザ関数のカーネル内実行を可能とした非同期カーネルサービスインタフェースを実現する．

2 既存手法：FlexSC

2.1 概要

FlexSC はシステムコール実行専用のカーネルスレッドとシステムコール要求発行用のリクエストキューから構成される．リクエストキューはアプリケーションとカーネルスレッドの間で共有される共有メモリである．アプリケーションはシステムコールを発行する際に，割り込みを発生させる代わりに，リクエストキューにシステムコール要求を書き込む．書き込まれた要求はカーネルスレッドによって取得され，システムコールが実行される．

FlexSC ではシステムコールの発行に割り込みを必要としないため，従来のシステムコールのようにアプリケーションやカーネルの実行を妨げることはない．また，アプリケーションの実行とシステムコールの実行が切り離されているため，それぞれを別のコア上で実行できるというメリットも生まれる．

2.2 問題点

FlexSC を適用した Apache や MySQL，memcached などの並列タスク処理アプリケーションは高い速度向上を実現している．しかし本研究では，FlexSC を適用したこれらのアプリケーションではリクエストキューへの操作が頻発してしまうことに着目した．そしてこのリクエストキュー操作の回数を削減することによって，さらなるアプリケーションの高速化を実現できると考えた．

Yusuke YASUI †　Shoichi SAITO †
†Nagoya Institute of Technology

3 提案手法：Sakura Call

3.1 概要

Sakura Call はユーザ関数のカーネル内実行を可能とした非同期カーネルサービスインタフェースである．FlexSC をベースとしており，FlexSC の基本的な構成要素を継承しているが，Sakura Call ではシステムコールの代わりに任意のユーザ関数の実行をカーネルスレッドに要求することができる．

Sakura Call ではシステムコール単位ではなくユーザ関数単位でカーネルスレッドに処理を要求できるため，FlexSC を用いた場合に比べて，リクエストキュー操作の回数を削減することができる．

3.2 ユーザ関数実行部の実装

Sakura Call では，アプリケーションはユーザ関数実行要求としてユーザ関数の関数ポインタと関数に与える引数をリクエストキューに書き込む．カーネルスレッドはリクエストキューに書き込まれた関数ポインタを引数と共に呼び出すことでユーザ関数を実行する．

カーネルスレッドによって実行されるユーザ関数中のシステムコール呼び出しは，Sakura Call が提供するマクロによって記述される必要がある．このマクロはコンパイル時にシステムコールサービスルーチンを直接呼び出すコードに展開される．これによりユーザ関数はカーネルスレッドが実行可能な形態となる．

4 評価

4.1 評価方法

評価環境を表 1 に示す．評価実験では従来手法，FlexSC，Sakura Call の 3 つの手法において並列タスク処理を実行し，その実行時間の比較を行った．なお，従来手法とは，Sakura Call のカーネルスレッドをユーザスレッドに置き換えたものであり，FlexSC は論文の情報を元に独自に作成したものを用いている．

実験は 4 コア環境で行い，1 コアにタスク生成スレッドを，残りのコアにワーカースレッドを配置した．ワーカースレッドとは従来手法におけるタスク実行用のユー

表 1: 評価環境

CPU	Intel(R) Core(TM) i5 CPU 760 @ 2.80GHz
Memory	8GB
Kernel	Linux i686 2.6.38.4
Storage	SSD SATA2 3.0Gb/s 64GB

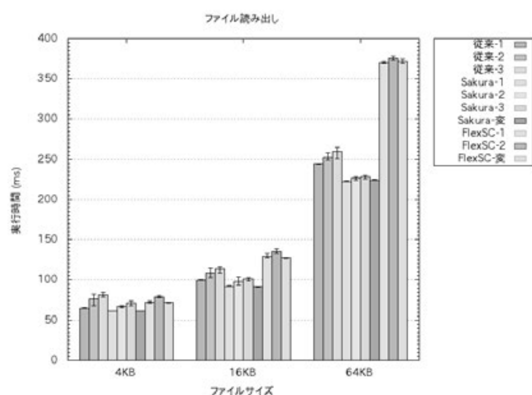


図 1: ファイル読み出し

ザスレッド，及び Sakura Call，FlexSC におけるカーネルスレッドを指す．

実験に用いたタスクは以下の 4 つである．

loop 指定回数ループを回す

getpid 指定回数 getpid を呼び出す

ファイル読み出し open → read（複数回）→ close

ファイル書き込み open → write（複数回）→ close

なお，タスク生成スレッドによって生成されたタスク数は 65536 個，システムコールのバッファサイズは 4KB とした．

4.2 評価結果

loop では予想通り従来手法と Sakura Call での実行時間に差がないことが確認された．これにより今回の評価実験における従来手法と Sakura Call の公平性が示された．また getpid では従来手法に比べて Sakura Call が大きく実行時間を削減していることを確認した．

ファイル書き込み（図 2）では Sakura Call による効果をあまり得られなかったが，ファイル読み出し（図 1）では Sakura Call により最大約 10%の高速化を達成した．しかし一方で，FlexSC ではファイル読み出し，ファイル書き込みのどちらにおいても，予想に反して他の両手法の実行時間を大きく上回る結果となった．この原因を特定するため，実行時間の内訳の調査を行ったところ，FlexSC におけるシステムコール処理時間が Sakura Call におけるタスク処理時間に対して大きすぎることがわかった．この原因の調査は今後の課題である．

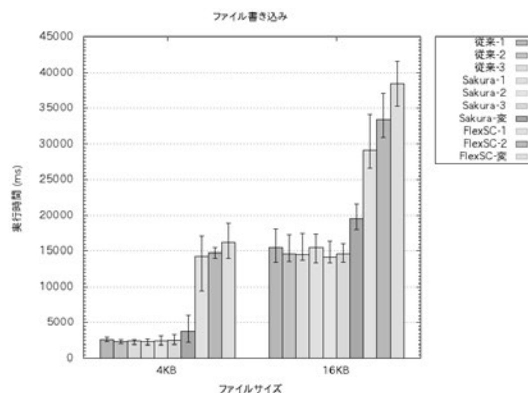


図 2: ファイル書き込み

5 Sakura Call の課題

Sakura Call では任意のユーザ関数をカーネルモードで実行するため，ユーザ関数を安全に実行する仕組みが求められる．この要求に対しては，セグメント機構を用いた対策が有効であると考えているが，未実装であるため，今後実装していく必要がある．また FlexSC に比べてアプリケーションの移植が困難であるという問題点あり，この問題の解決も今後の課題である．

6 まとめ

ユーザ関数のカーネル内実行を可能とした非同期カーネルサービスインタフェースである Sakura Call を提案した．この手法ではシステムコール単位ではなくユーザ関数単位での処理をカーネルに要求することができる．今後の課題としては，ユーザ関数の安全な実行を保障する仕組みの実現や，移植の困難性の問題の解決などが挙げられる．

参考文献

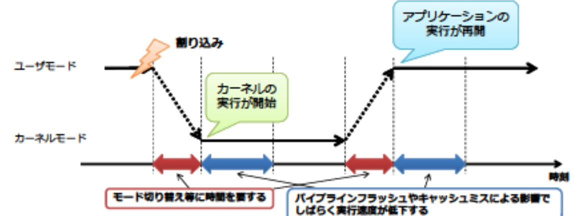
[1] Livio Soares, M. S.: FlexSC: Flexible System Call Scheduling with Exception-Less System Calls (2010), 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10).

ユーザ関数のカーネル内実行を可能とした非同期カーネルサービスインタフェースの実現

2012/09/11
名古屋工業大学大学院
齋藤研究室
安井 裕亮

研究背景

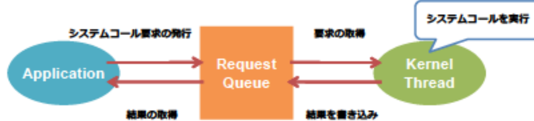
- システムコール
  - 一般的に割り込みを用いて実装されてきた
- 従来のシステムコールの問題点
  - アプリケーションやカーネルの実行を妨げる



2

FlexSC†

- 割り込みを用いないシステムコール
  - システムコール実行用のカーネルスレッド
  - システムコール要求発行用のリクエストキュー



- 特長
  - 割り込みを必要としないため効率的な実行を妨げない
  - アプリケーションとシステムコールを異なるコア上で実行できる

†FlexSC: Flexible System Call Scheduling with Exception-Less System Calls, OSDI '10

3

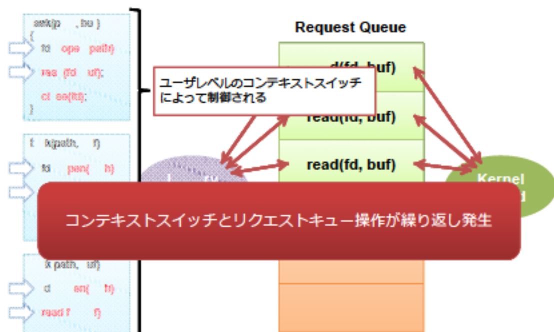
FlexSCが提供するライブラリ

- 並列タスク処理アプリケーション向けライブラリ
  - FlexSC-Threads（マルチスレッドアプリケーション）
    - POSIX Thread互換ライブラリ
    - アプリケーションのコード変更が不要
      - Apacheで最大116%、MySQLで最大105%の性能向上
  - libflexsc（イベントドリブンアプリケーション）
    - libeventライクなライブラリ
    - 少ないコード変更で移植可能
      - memcachedで最大35%、nginxで最大120%の性能向上

FlexSCでの並列タスク処理には削減可能なコストが存在することに着目
これらを削減することで更なる高速化を目指す

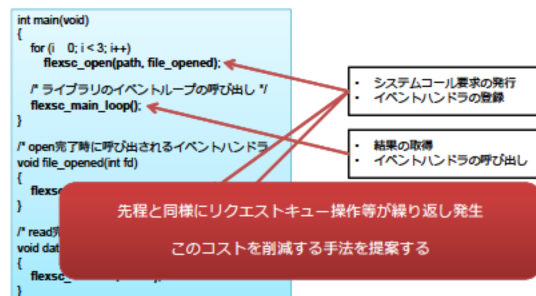
4

FlexSC-Threadsを用いたアプリケーションの挙動



5

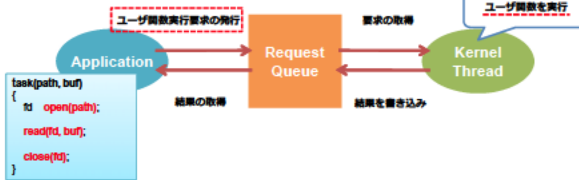
libflexscを用いたアプリケーションの挙動



6

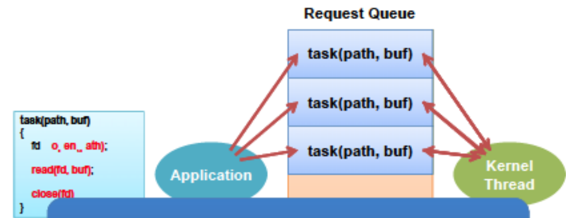
提案手法：Sakura Call

- ユーザ関数のカーネル内実行を可能とした非同期カーネルサービスインタフェース
  - システムコールの代わりに任意のユーザ関数の実行をカーネルスレッドに要求できる



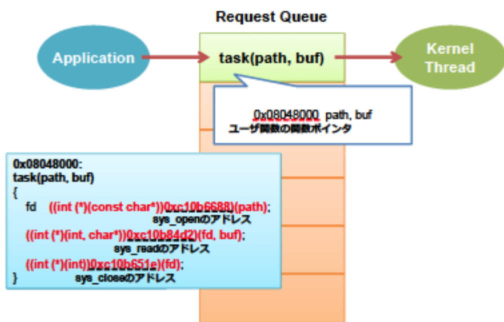
7

Sakura Callを用いたアプリケーションの挙動



8

Sakura Callにおけるユーザ関数実行



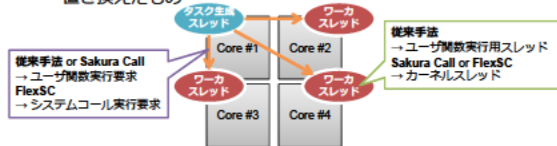
9

評価実験

- 実験環境

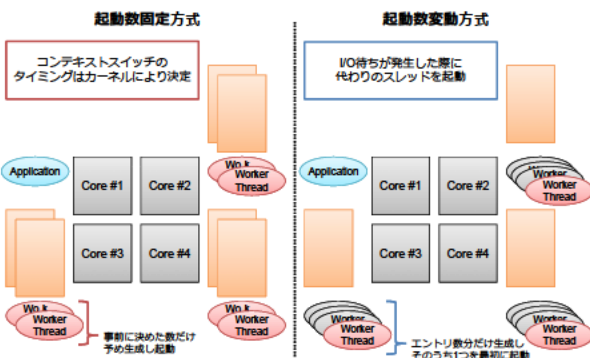
CPU	Intel® Core™ i5 CPU 760 @ 2.80GHz (4 Cores)
Memory	8GB
Kernel	Linux i686 2.6.38.4
Storage	SSD SATA2 3.0Gb/s 64GB

- 実験内容
  - 従来手法、FlexSC、Sakura Callの３つの手法において並列タスク処理時間を比較
    - 従来手法：Sakura Callのカーネルスレッドをユーザスレッドに置き換えたもの



10

ワーカスレッドのスケジューリング方式



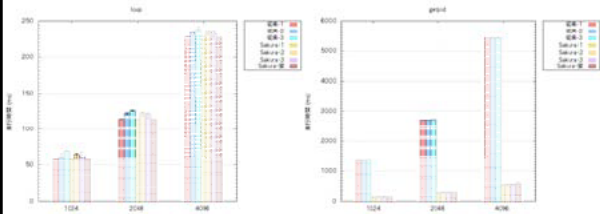
11

実験項目

- 実験に用いたタスク
  - loop（従来手法、Sakura Callのみ）
    - 指定回数空のループを回す
  - getpid（従来手法、Sakura Callのみ）
    - 指定回数getpidを呼び出す
  - ファイル読み出し
    - open → read（複数回）→ close
  - ファイル書き込み
    - open → write（複数回）→ close
- 実験パラメータ
  - タスク数：65536
  - システムコールのバッファサイズ：4KB

12

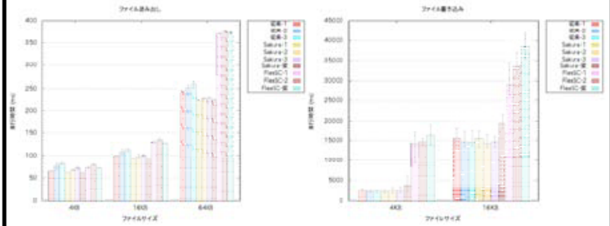
実験結果（1／2）



- 結果
  - loopでは実行時間に差は見られない
  - getpidではSakura Callによるシステムコール発行コストの削減により大幅に実行時間が減少している

13

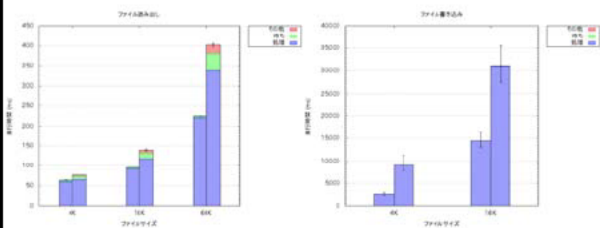
評価結果（2／2）



- 結果
  - ファイル読み出しにおいてSakura Callは最大10%の実行時間削減を達成
  - FlexSCでは予想に反して実行時間が増加

14

実行時間の内訳



- 考察
  - リクエストキューの操作時間はリクエストキューへの操作回数に比例（予想通り）
  - システムコールの処理時間が予想に反して増大

15

Sakura Callの課題

- アプリケーションの移植がFlexSCに比べて困難
  - FlexSCでは提供するライブラリにより簡単に移植可能
  - Sakura Callではプログラマにより書き換える必要がある
    - この問題の解決は今後の課題
- セキュリティ対策が必要
  - セグメント機構を用いた対策が可能であると考える
    - 参考
      - Cosy: Develop in User-Land, Run in Kernel-Mode, HotOS 2003
      - Efficient and Safe Execution of User-Level Code in the Kernel, PDPS 2005

16

まとめと今後の予定

- まとめ
  - 提案手法：Sakura Call
    - ユーザ関数のカーネル内実行を可能とした非同期カーネルサービスインタフェース
    - ユーザ関数単位での処理要求が可能
    - 従来のシステムコールとの比較において有効性を証明
- 今後の予定
  - FlexSCにおいて実行時間が増加する原因の調査
  - アプリケーション移植の問題の解決
  - セキュリティの対策

17

Cgroups による CPU 資源管理の性能評価

富樫 荘太 †
† 立命館大学大学院情報理工学研究科

1 はじめに

近年, カーマルチメディア機器や携帯電話などにおいて, プリインストールアプリとダウンロードアプリの両立が求められる. プリインストールアプリは, 機器に元々インストール済みのアプリであり, 中心的な機能を提供する. 一方, ダウンロードアプリは, ユーザが後から追加するアプリであり, 補助的な機能を提供する. このような場合, プリインストールアプリがダウンロードアプリの影響を受けないようにする必要がある. 本研究では, 決められた周期で動作するプリインストールアプリに対して適切な資源管理が可能な機構について調査および検討を行う. 本稿では, Linux が提供する資源管理機構 Cgroups[1] を使用し, その性能評価を行った. なお, プリインストールアプリに 60fps で画面に出力する描画系アプリを想定した. この場合, プリインストールアプリは, 16msec 周期での性能保証を必要とする.

2 Cgroups

2.1 概要

Linux 2.6.24 以降, カーネルの機能として Cgroups が提供されている. Cgroups は, プロセスをグループ化し, グループ毎の CPU, メモリ, I/O, ネットワークといった資源毎にリソースコントローラを割り当てて資源管理を行う. グループは, 仮想ファイルシステムを通じて階層構造を構成し, 管理内容および管理対象を分割可能である. サブグループは, 特に指定しない限りは親の管理内容を継承する.

2.2 cpu リソースコントローラ

cpu リソースコントローラは, グループ単位の CPU 利用率の制御が可能である. cpu リソースコントローラは, 以下に示すパラメータが存在する.

cpu.shares
: ノーマルプロセス (以下, CFS プロセス) のタイムスライスの重み付けをグループ単位で行う.

cpu.cfs (cpu.cfs_quota_us, cpu.cfs_period_us)
: ノーマルプロセスの CPU 利用率を時間で指定し制御する. cpu.cfs_period_us に指定された周期において, cpu.cfs_quota_us だけ CPU の使用を許可する (= throttling 処理).

cpu.rt (cpu.rt_runtime_us, cpu.rt_period_us)
: リアルタイムプロセス (以下, RT プロセス) の CPU 利用率を時間で指定し制御する. cpu.rt_period_us に指定された周期において, cpu.rt_runtime_us と利用可能 CPU 数の積だけ CPU の使用を許可する (= throttling 処理)[2].

2.3 throttling 処理の共存と想定システムへの検討

図 1 に, cpu.cfs と cpu.rt の throttling 処理の関係について示す. 図 1 のように, Cgroups による CPU 資源管理は, 各ランキューを階層状にすることで実現している. RT の場合, サブグループを, サブグループのランキュー内で最高優先度のスケジュールエンティティの優先度をそのグループの優先度としている. CFS の場合, 赤黒木でスケジュールエンティティが仮想実行時間をもとに順序付けされる. これらは, shares や nice の値によって再度エンキューする位置が異なる.

各ランキューは実行時間を使い切った場合にデキューされ,

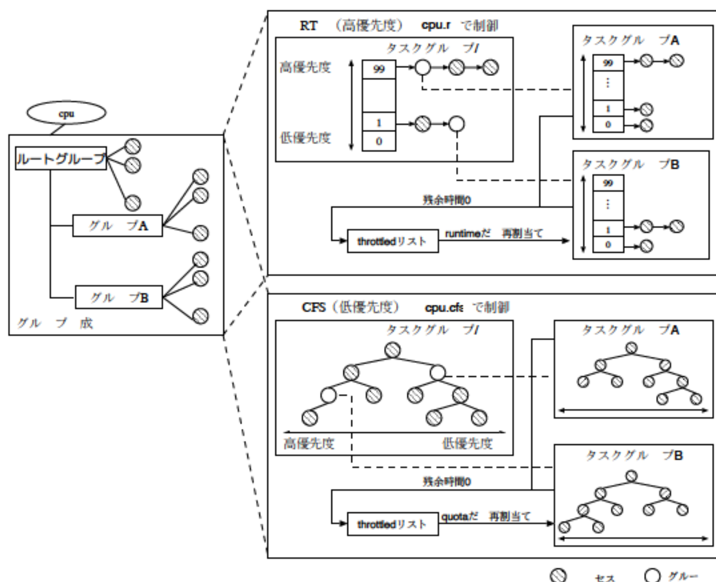


図 1 cpu リソースコントローラを用いた CPU 制御

表 1 評価環境

項目	内容
CPU	Intel Core2 Quad Q6600 2.4GHz
Memory	2.0GB
OS	Fedora 14
Kernel	Linux Kernel 3.2.1

period にて指定した周期がくると再度時間を割り当てられる. そのため, cpu.cfs や cpu.rt は, 上限値のみ保証し, 下限値を保証しない. 特に, cpu.cfs は, RT や CFS で動く他プロセスの影響を大きく受ける. したがって, 想定システムを考慮すると cpu.cfs でのプリインストールアプリの厳密な性能保証には適さない. 一方, cpu.rt は, RT が CFS より高優先度で動作するため, プリインストールアプリ側に自分より高優先度の RT プロセスが存在しない場合, 下限値を上限値に近づけることが可能である.

3 cpu リソースコントローラの評価

3.1 評価目的

プリインストールアプリの 16msec 周期毎の性能保証のために, cpu.rt を用いて制御しその評価を行う. プリインストールアプリとみなした RT ビジーループを動作させ, cpu.rt によって上限値を設定し, RT の優先度を最高優先度にすることで下限値を上限値に近づける. 本稿では, 条件を変えて評価を行い, 各々最小値と最大値を求め下限値と上限値の制御ができているかを確認し, また, 平均値と標準偏差値を求めることで, 平均値に対するばらつきを検討する.

3.2 評価内容

評価は, 表 1 のような環境で行った. 評価では, 図 2 に示すように cpu リソースコントローラと, CPU 数の割当てを設定する cpuset リソースコントローラの 2 つを使用する. Linux の制約により, ルートグループの CPU 数は変更不可能であり, システムプロセスはルートグループに所属しなければならない. よって, ルートグループのサブグループとして, 計測対象および負荷

表 2　各条件における RT タスク群の平均 CPU 利用率

コア数	内容	最小値 (%)	最大値 (%)	平均値 (%)	標準偏差値
1 個	RT ビジーループ 1 個	49.7678	56.2210	50.3877	1.5964
	RT ビジーループ 2 個	49.8177	56.2123	50.3894	7.9851
	RT ビジーループ 3 個	49.7896	56.2128	50.3891	8.3651
	RT ビジーループ 1 個 (負荷あり)	49.8594	56.2469	50.3011	1.3627
	RT ビジーループ 2 個 (負荷あり)	49.7833	56.2432	50.3009	8.0821
	RT ビジーループ 3 個 (負荷あり)	49.8271	56.2553	50.2997	7.7367
2 個	RT ビジーループ 1 個	49.9559	56.2380	50.4107	1.5957
	RT ビジーループ 2 個	49.9600	56.2250	50.4123	7.9390
	RT ビジーループ 3 個	24.9074	31.1885	53.6585	5.6100
	RT ビジーループ 1 個 (負荷あり)	49.8169	56.2365	50.2940	1.3613
	RT ビジーループ 2 個 (負荷あり)	49.9542	56.2630	50.2963	7.5317
	RT ビジーループ 3 個 (負荷あり)	24.9521	31.2247	53.5280	4.2040

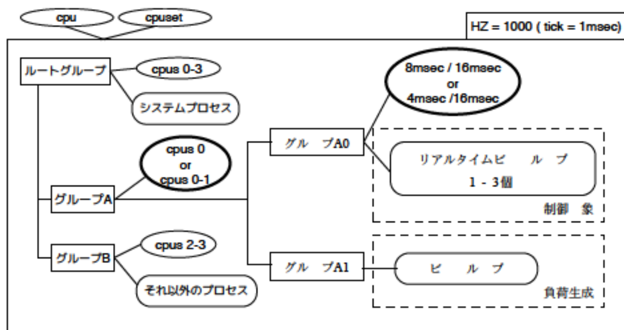


図 2　Cgroups の評価用設定

生成のプロセス (グループ A), それ以外の全プロセス (グループ B) の 2 つのグループに分割した. グループ A およびグループ B は cpuset によって CPU 資源を分割し, グループ B のプロセスの影響がグループ A のプロセスに影響しないようにする. また, 同様の理由でルートグループの負荷分散を無効化する. グループ A は, CPU0 だけを割り当てる場合と, CPU0〜1 を割り当てる場合の 2 通りで評価し, グループ B は常に CPU2〜3 のみを割り当てる. グループ A は, 計測対象となるプロセスグループ A0 と負荷生成のプロセスグループ A1 の 2 つのサブグループを持つ. グループ A0 は, RT ビジーループを 1〜3 個実行し, グループの CPU 利用率を 16msec 中で 50% になるように設定する. つまり, CPU0 の場合は cpu.rt を 8msec/16msec に, CPU0〜1 の場合は cpu.rt を 4msec/16msec に設定する. グループ A1 は, CFS ビジーループを 0〜1 個実行し, 同一 CPU で動作する制御対象のプロセスへの影響を検討する. throttling 処理の起点となるタイマ割込みを HZ を 1000 に設定し 1msec 間隔で生じるように設定した. 計測には, ftrace を使用し, 16msec 周期で 100 回連続してグループ A0 の CPU 利用率を計測する. また, 正確に計測するために, CPU 負荷に応じて動的にタイマ割込み数が変動する Dynamic Tick を無効にする.

3.3　結果と考察

表 2 のような評価結果が得られた. 各条件における最小値, 最大値, 平均値, 標準偏差値を示す. 特に CPU2 個の場合, 最小値と最大値はいずれかの CPU においての CPU 利用率が最小のものと最大のものを示す. 平均値および標準偏差値は, それぞれ 2 つの CPU における平均値および標準偏差値の和を示す.

3.3.1　CPU 2個で並行動作する場合

結果より, CPU 2 個で RT ビジーループ 3 個の場合が他と大きく異なることが分かる. CPU 数 2 個の場合で RT ビジーループが 1,2 個の場合, 負荷の有無に関わらず, 片方の CPU で RT ビジーループは動作し, ビジーループが 1 個の場合は 50%, ビジーループが 2 個の場合はおのおの 25% をとり順番に割り当てられる. 一方で, CPU2 個の場合で RT ビジーループが 3 個の場合, 2 つの CPU で 2 個の RT ビジーループと 1 個の RT ビジーループに分かれて動作する. このとき, 2 個の RT ビジーループがおのおの 25% で順番に割り当てられるのに対して, 1 個の RT ビジーループは, 31% と 25% を交互にとる. このように, CPU 数 2個で RT ビジーループ 3 個の場合だけ 2 個の CPU で並行して動作することにより, 最小値および最大値が 50% より大きく下回り, 平均値も約 3.6% 上回る.

3.3.2　CPU1 個のみで動作する場合

CPU2 個で RT ビジーループ 3 個以外の場合, RT ビジーループは片方の CPU のみで実行される. このとき, 最大値, 最小値ともに期待される値は 50% であるが, 最小値はどれも 0.1% 程度小さく, 最大値はどれも 6.2% 程度大きい. また, 最小値と最大値について条件と結果の相関関係は見られなかった. cpu.rt によって制御可能な最大値が大きくなるのは, タイマ割込みが 1msec 周期で入るのに対して cpu.rt の制御間隔が 16msec 周期であるため, $1msec/16msec = 6.2\%$ 程度の計測誤差が生じるためである.

平均値は, どれも 50% 程度となったが, 標準偏差を見ると条件によってばらつきが変動していることが分かる. グループ A0 内のプロセス個数を増加させた場合ばらつきが増大し, グループ A1 から負荷をかけた場合ばらつきが多くで減少した.

4　おわりに

本稿では, Linux における資源管理機能である Cgroups について, CPU 資源管理の性能評価を行った. 複数 CPU で並行動作する場合, 平均値が増大した. また, 同一グループ内でのプロセス個数を増加させた場合や同一 CPU を共有する別グループから負荷をかけた場合にばらつきが変動した. 今後, これらの性能低下の妥当性を検討し必要ならその解決手法を模索する.

また, 今回は cpu.rt の制御で下限値を上限値と等しくするため, RT ビジーループの優先度を最高優先度に設定した. しかし, これはカーネルサービスに影響を及ぼすため, 想定システムにおいて望ましい優先度設定とは言えない. そのため, 今後はカーネルサービスとプリインストールアプリの両立をするために適切な優先度設定も検討する. さらに, 他資源の管理についても性能の調査および検討を行う.

参考文献

[1] Menage, P.: Linux Kernel Documentation/cgroups/cgroups.txt, http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt (2012).

[2] Linux Kernel Documentation/scheduler/sched-rt-group.txt, http://kernel.org/doc/Documentation/scheduler/sched-rt-group.txt (2012).

CgroupsによるCPU資源管理の性能評価

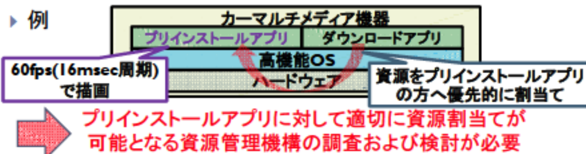
立命館大学大学院　情報理工学研究科
富樫 荘太

CgroupsによるCPU資源管理の性能評価

- はじめに
- cpu.rtとcpu.cfsの実装調査
  - throttling処理の実装調査
    - tickとローカルタイマ割り込み
    - 動的タイマを用いたthrottling処理
  - 想定システムへの適用可能性
- cpu.rtのthrottling処理の性能評価
  - 同一CPUを使用するプロセスの影響
  - グループ内プロセス数の変化による影響
  - コア数の変化による影響

はじめに

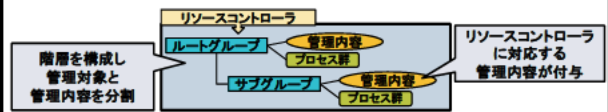
- 近年,カーマルチメディア機器,携帯電話等において
  プリインストールアプリとダウンロードアプリの両立の要求
  - プリインストールアプリ:
    音楽/ビデオプレイヤー,ハンズフリー通話機能など
  - ダウンロードアプリ: ユーザが後から追加するアプリ
- プリインストールアプリがダウンロードアプリの影響を
  受けないようにする必要がある
- 例



プリインストールアプリに対して適切に資源割当てが
可能となる資源管理機構の調査および検討が必要

Cgroups(Control Groups)

- Linuxに既存の資源管理機構
- プロセスグループ毎の資源管理が可能
- 各資源毎にリソースコントローラを持つ
  - CPU, I/O, メモリ, Network等の資源管理が可能
- グループは階層を構成し管理内容/対象の変更が可能



プリインストールアプリに描画系を想定した場合
16msec程度の精度での資源管理が必要となるため
Cgroupsの管理周期の精度について性能調査を行う

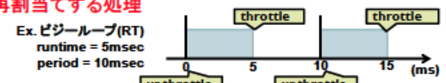
cpuリソースコントローラ

- グループ間のCPU利用率の比率を制御するパラメタ
  - cpu.shares
    - グループ毎に設定したshares値の重みに基づいて
      グループに属するノーマルプロセスにCPUを割り当てる
- CPU利用率を時間によって制御する (throttling処理)を
  可能にするパラメタ
  - cpu.cfs (cpu.cfs_quota_us / cpu.cfs_period_us)
    - ノーマルプロセスが対象
    - period時間においてquota時間だけCPUを割り当てる
  - cpu.rt (cpu.rt_runtime_us / cpu.rt_period_us)
    - リアルタイムプロセスが対象
    - period時間においてruntime時間だけCPUを割り当てる

throttling処理

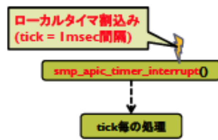
- cpu.cfsやcpu.rtによる指定周期でプロセスグループに
  指定したCPU時間以上を使わせないように制御する処理
- throttleとunthrottleの2つの処理に分けられる
  - throttle
    グループが時間を使い切った場合にそのグループが
    周期更新までCPUを使用させないようにする処理
  - unthrottle
    時間を使い切ったグループに対して周期更新の際に
    CPUを再割当てする処理

Ex. ビジーループ(RT)
runtime = 5msec
period = 10msec



ローカルタイマ割込み

- 各CPUはHZで設定した値に基づきタイマ割込みを受けてタスクの切換えを行う
- カーネルオプションHZで設定可能
  - HZ=1000のときはtickは1msec間隔でおきる
- x86環境だとsmp_apic_timer_interrupt()がタイマハンドラ



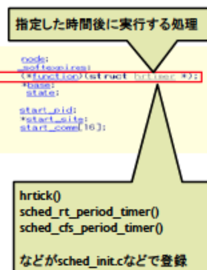
スケジューリングクラス

- タイマ割込み毎にカレントプロセスを以下の優先順序で選択
  - ① rt-schedクラス
    - SCHED_FIFOポリシ：リアルタイム優先度, FIFO
    - SCHED_RRポリシ：リアルタイム優先度, RR
    - ⇒ cpu.rtによる制御は このスケジューリングに従う
  - ② fair-schedクラス
    - SCHED_OTHERポリシ：公平なCPU分配
    - ⇒ cpu.sharesやcpu.cfsによる制御は このスケジューリングに従う
  - ③ idle-schedクラス
    - SCHED_DLEポリシ：アイドル状態

動的タイマ:hrtimer

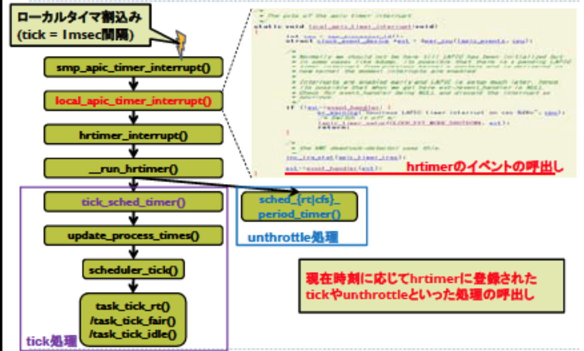
- hrtimer(High-Resolution Timer)
  - nanosleepなどの時限処理に用いられる
- 構造体

```
struct hrtimer {
        struct timerqueue_node          node;
        ktime_t                         _softexpires;
        enum hrtimer_restart            (*function)(struct hrtimer *);
        struct hrtimer_clock_base       *base;
        unsigned long                   state;
#ifdef CONFIG_TIMER_STATS
        int                             start_pid;
        void                            *start_site;
        char                            start_comm[16];
#endif
};
```



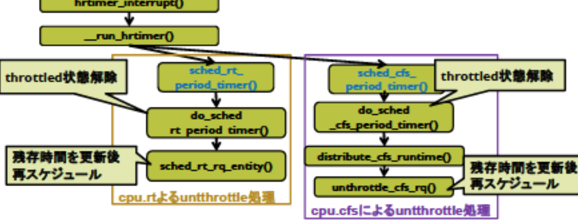
- API
  - hrtimer_init()
  - hrtimer_start()
  - hrtimer_cancel()
  - hrtimer_forward()
  - ・・・

動的タイマによるtickやunthrottle処理の実現



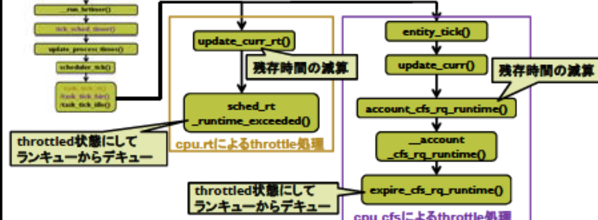
unthrottle処理の実装

- throttled状態の解除
  (throttledリストからデキュー)
- スケジュールエンティティの
  残存時間を更新後再スケジュール

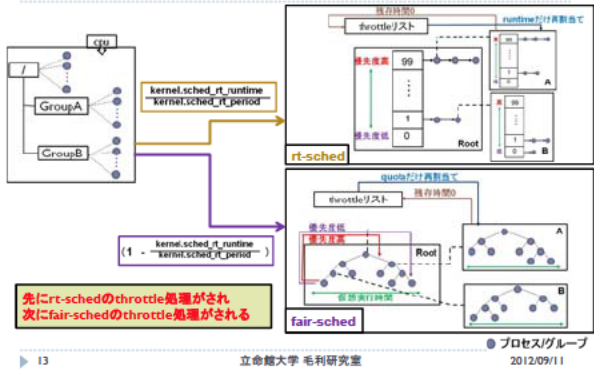


throttle処理の実装

- スケジュールエンティティの
  残存時間を実行時間で減算
- throttled状態にして(throttledリストへキュー)
  その後ランキューからデキュー



cpu.rtとcpu.cfsのthrottling処理の共存



実装から見たthrottling処理のまとめと想定システムへの適用可能性の検討

- throttling処理はHZ(～1000)によって決定される
  ローカルタイマ割込み(1msec～)を契機とする
  ⇒16msecの制御の場合
  それ以下の粒度での割込みが必要となる
- throttling処理は上限値の保証しかできないため
  横取りなどの影響で下限値は最悪0%になりうる
  ⇒下限値を上限値に近づけるためにはcpu.rtを使用し
  最高優先度でプリインストールアプリを保護し
  ダウンロードアプリはそれ以下の優先度で動作させる

cpu.rtの評価を行い適用可能性のさらなる検討

cpu.rtのthrottling処理の性能評価

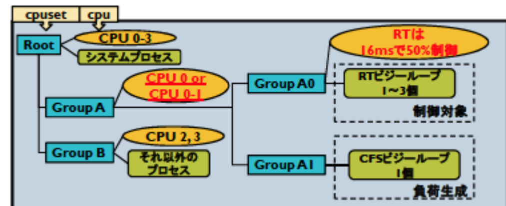
- 評価目的
  - cpu.rtを用いた16msec周期毎のCPU資源管理の評価
    - 60fps(16msec)レベルにおいてcpu.rtの制御が可能か
      条件を変えて確認
      - 同一CPUを使用するプロセスの影響
      - グループ内プロセス数の変化による影響
      - コア数の変化による影響
- 評価概要
  - cgroupsを用いて最高優先度99のリアルタイムプロセスとして
    動作するビジーループ(RTビジーループ)に
    CPU資源を優先的に割り当てそれを計測し評価する

評価内容(1/2)

- コア数を1個, 2個のもとで
  グループAに属する
  RTプロセス全体のCPU利用率を
  16msで50%になるように制御する

評価環境

CPU	Intel Core2 Quad Q6600 2.4GHz
Memory	2.0GB
OS	Fedora 14
Kernel	Linux Kernel 3.2.1

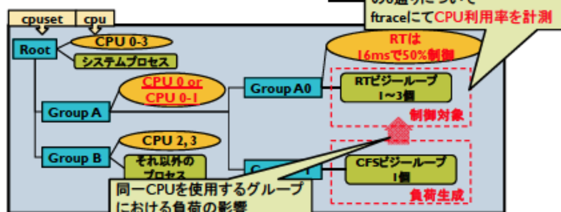


評価内容(2/2)

- コア数を1個, 2個のもとで
  グループAに属する
  RTプロセス全体のCPU利用率を
  16msで50%になるように制御する

CPU0のみの場合と
CPU0とCPU1の場合について
RTビジーループ1個
RTビジーループ2個
RTビジーループ3個
RTビジーループ1個(負荷あり)
RTビジーループ2個(負荷あり)
RTビジーループ3個(負荷あり)
の6通りについて
ftraceにてCPU利用率を計測

CPU
Mem
OS
Kern



注意すべき評価設定(1/2)

- HZ=1000に設定
  - 16msecの資源管理を行うために,
    Linuxのローカルタイマ割込みを推奨される範囲で最大にし,
    1msec粒度でのthrottling処理にする
- Dynamic Tickの無効化
  - Dynamic Tickを有効にするとCPU利用率に応じて
    HZの回数が変動し, throttling処理やftraceによる測定に
    影響するため無効化する

注意すべき評価設定(2/2)

- cpu.rtで複数コアを制御する場合

$$CPU利用率上限値 = \frac{cpu.rt_runtime_us}{cpu.rt_period_us} \times 利用可能CPUコア数$$

であるため，16ms中に50%のみ動作を許可する場合
1コアの場合は8ms/16msだけ，
2コアの場合は4ms/16msだけ，グループAに許可する

- Rootグループの負荷分散を無効にする
  - 有効である場合，グループA下で動作するRTプロセスのthrottlingが上手く働かないため

各条件におけるRTタスク群の平均CPU利用率(1/7)

コア	内容	最小値(%)	最大値(%)	平均値(%)	標準偏差値
1個	RTビジーループ1個	49.7678	56.2210	50.3877	1.5964
	RTビジーループ2個	49.8177	56.2123	50.3894	7.9851
	RTビジーループ3個	49.7896	56.2128	50.3891	8.3651
	RTビジーループ1個(負荷あり)	49.8594	56.2469	50.3011	1.3627
	RTビジーループ2個(負荷あり)	49.7833	56.2432	50.3009	8.0821
	RTビジーループ3個(負荷あり)	49.8271	56.2553	50.2997	7.7367
2個	RTビジーループ1個	49.9559	56.2380	50.4107	1.5957
	RTビジーループ2個	49.9600	56.2250	50.4123	7.9390
	RTビジーループ3個	24.9074	31.1885	53.6585	5.6100
	RTビジーループ1個(負荷あり)	49.8169	56.2365	50.2940	1.3613
	RTビジーループ2個(負荷あり)	49.9542	56.2630	50.2963	7.5317
	RTビジーループ3個(負荷あり)	24.9521	31.2247	53.5280	4.2040

各条件におけるRTタスク群の平均CPU利用率(2/7)

コア	内容	最小値(%)	最大値(%)	平均値(%)	標準偏差値
1個	RTビジーループ1個	49.7678	56.2210	50.3877	1.5964
	RTビジーループ2個	49.8177	56.2123	50.3894	7.9851
	RTビジーループ3個	49.7896	56.2128	50.3891	8.3651
	RTビジーループ1個(負荷あり)	49.8594	56.2469	50.3011	1.3627
	RTビジーループ2個(負荷あり)	49.7833	56.2432	50.3009	8.0821
	RTビジーループ3個(負荷あり)	49.8271	56.2553	50.2997	7.7367
2個	RTビジーループ1個	49.9559	56.2380	50.4107	1.5957
	RTビジーループ2個	49.9600	56.2250	50.4123	7.9390
	RTビジーループ3個	24.9074	31.1885	53.6585	5.6100
	RTビジーループ1個(負荷あり)	49.8169	56.2365	50.2940	1.3613
	RTビジーループ2個(負荷あり)	49.9542	56.2630	50.2963	7.5317
	RTビジーループ3個(負荷あり)	24.9521	31.2247	53.5280	4.2040

2コアのときについては，最小値(コア毎)，最大値(コア毎)，
平均値(コア毎の平均の和)，標準偏差(コア毎の標準偏差の和)

各条件におけるRTタスク群の平均CPU利用率(3/7)

コア	内容	最小値(%)	最大値(%)	平均値(%)	標準偏差値
1個	RTビジーループ1個	49.7678	56.2210	50.3877	1.5964
	RTビジーループ2個	49.8177	56.2123	50.3894	7.9851
	RTビジーループ3個	49.7896	56.2128	50.3891	8.3651
				50.3011	1.3627
				50.3009	8.0821
				50.2997	7.7367
2個	RTビジーループ1個	49.9559	56.2380	50.4107	1.5957
	RTビジーループ2個	49.9600	56.2250	50.4123	7.9390
	RTビジーループ3個	24.9074	31.1885	53.6585	5.6100
	RTビジーループ1個(負荷あり)	49.8169	56.2365	50.2940	1.3613
	RTビジーループ2個(負荷あり)	49.9542	56.2630	50.2963	7.5317
	RTビジーループ3個(負荷あり)	24.9521	31.2247	53.5280	4.2040

2コアRTビジーループ3個の場合のみ2コアで並行動作する
(2コアでRTビジーループ1～2個の場合は片方のコアで動作)
これにより平均値が期待される50%より3.6%程度増大

各条件におけるRTタスク群の平均CPU利用率(4/7)

コア	内容	最小値(%)	最大値(%)	平均値(%)	標準偏差値
1個	RTビジーループ1個	49.7678	56.2210	50.3877	1.5964
	RTビジーループ2個	49.8177	56.2123	50.3894	7.9851
	RTビジーループ3個	49.7896	56.2128	50.3891	8.3651
	RTビジーループ1個(負荷あり)	49.8594	56.2469	50.3011	1.3627
	RTビジーループ2個(負荷あり)	49.7833	56.2432	50.3009	8.0821
	RTビジーループ3個(負荷あり)	49.8271	56.2553	50.2997	7.7367
2個	RTビジーループ1個	49.9559	56.2380	50.4107	1.5957
	RTビジーループ2個	49.9600	56.2250	50.4123	7.9390
	RTビジーループ3個	24.9074	31.1885	53.6585	5.6100
	RTビジーループ1個(負荷あり)	49.8169	56.2365	50.2940	1.3613
	RTビジーループ2個(負荷あり)	49.9542	56.2630	50.2963	7.5317

2コアで並行して動く場合を除き，最小値と最大値は6%程度の開きがあり
最小値は期待される値から0.2%程度ほど小さく，最大値は6.2%程度大きい
これはタイマ割込み1ms周期に対してunthrottleの幅が16msであり
計測タイミングのズレによって+6.25% (1ms/16ms)になるため

各条件におけるRTタスク群の平均CPU利用率(5/7)

コア	内容	最小値(%)	最大値(%)	平均値(%)	標準偏差値
1個	RTビジーループ1個	49.7678	56.2210	50.3877	1.5964
	RTビジーループ2個	49.8177	56.2123	50.3894	7.9851
	RTビジーループ3個	49.7896	56.2128	50.3891	8.3651
	RTビジーループ1個(負荷あり)	49.8594	56.2469	50.3011	1.3627
	RTビジーループ2個(負荷あり)	49.7833	56.2432	50.3009	8.0821
	RTビジーループ3個(負荷あり)	49.8271	56.2553	50.2997	7.7367
2個	RTビジーループ1個	49.9559	56.2380	50.4107	1.5957
	RTビジーループ2個	49.9600	56.2250	50.4123	7.9390
	RTビジーループ3個	24.9074	31.1885	53.6585	5.6100
	RTビジーループ1個(負荷あり)	49.8169	56.2365	50.2940	1.3613
	RTビジーループ2個(負荷あり)	49.9542	56.2630	50.2963	7.5317
	RTビジーループ3個(負荷あり)	24.9521	31.2247	53.5280	4.2040

2コアで並行して動く場合を除くと
その他はコア数によらず平均値は1%未満のずれ

各条件におけるRTタスク群の平均CPU利用率(6/7)

コア	内容	最小値(%)	最大値(%)	平均値(%)	標準偏差値
1個	RTビジーループ1個	49.7678	56.2210	50.3877	1.5964
	RTビジーループ2個	49.8177	56.2123	50.3894	7.9851
	RTビジーループ3個	49.7896	56.2128	50.3891	8.3651
	RTビジーループ1個(負荷あり)	49.8594	56.2469	50.3011	1.3627
	RTビジーループ2個(負荷あり)	49.7833	56.2432	50.3009	8.0821
	RTビジーループ3個(負荷あり)	49.8271	56.2553	50.2997	7.7367
2個	RTビジーループ1個	49.9559	56.2380	50.4107	1.5957
	RTビジーループ2個	49.9600	56.2250	50.4123	7.9390
	RTビジーループ3個	24.9074	31.1885	53.6585	5.6100
	RTビジーループ1個(負荷あり)	49.8169	56.2365	50.2940	1.3613
	RTビジーループ2個(負荷あり)	49.9542	56.2630	50.2963	7.5317
	RTビジーループ3個(負荷あり)	24.9521	31.2247	53.5280	4.2040

2コアで並行して動く場合を除くとプロセスの個数に応じて標準偏差が増大

25

各条件におけるRTタスク群の平均CPU利用率(7/7)

コア	内容	最小値(%)	最大値(%)	平均値(%)	標準偏差値
1個	RTビジーループ1個	49.7678	56.2210	50.3877	1.5964
	RTビジーループ2個	49.8177	56.2123	50.3894	7.9851
	RTビジーループ3個	49.7896	56.2128	50.3891	8.3651
	RTビジーループ1個(負荷あり)	49.8594	56.2469	50.3011	1.3627
	RTビジーループ2個(負荷あり)	49.7833	56.2432	50.3009	8.0821
	RTビジーループ3個(負荷あり)	49.8271	56.2553	50.2997	7.7367
2個	RTビジーループ1個	49.9559	56.2380	50.4107	1.5957
	RTビジーループ2個	49.9600	56.2250	50.4123	7.9390
	RTビジーループ3個	24.9074	31.1885	53.6585	5.6100
	RTビジーループ1個(負荷あり)	49.8169	56.2365	50.2940	1.3613
	RTビジーループ2個(負荷あり)	49.9542	56.2630	50.2963	7.5317
	RTビジーループ3個(負荷あり)	24.9521	31.2247	53.5280	4.2040

2コアで並行して動く場合を除くとグループA0で動作する
CFSビジーループの負荷による標準偏差値は多くで減少

26 2012/09/11

まとめ

- 評価結果
  - 下限値と上限値に平均して6.4%程度の誤差
  - 複数CPUで並行動作する場合平均値が3.6%上昇
  - 同一グループ内でのプロセス個数を増加させた場合に
    その個数に応じてばらつきが増大
  - 同一CPUを共有する別グループから負荷をかけた場合に
    ばらつきが多くで減少
- 今後の予定
  - これらの性能低下の妥当性と必要ならその解決手法の検討
  - cpu.rtの制御対象のプロセスの優先度と
    カーネルサービスとの兼ね合いを検討
  - Cgroupsの他資源管理についても性能調査と検討

27 立命館大学 毛利研究室 2012/09/11

マルチコア・メニーコア混在型計算機における資源管理代行方式の試作と評価

深沢 豪†
東京農工大学大学院工学府情報工学専攻†

1 はじめに

大規模シミュレーション等，高い演算性能が必要な用途に用いられるスーパーコンピュータは現在ペタフロップス級の演算性能を有しているが，次世代ではエクサフロップス級の演算性能が求められる．我々は，汎用的な CPU コアを多数集積したメニーコア CPU を用いてエクサスケールを目指した次世代スーパーコンピュータの開発を行っている．メニーコアは高い並列演算性能を有するが，コア単体性能が低いという欠点がある．そこで，従来のマルチコア CPU をメニーコアとともに用いることで，メニーコアにはない高い単スレッド性能を得る．本計算機では，特性が異なる両 CPU に合わせて 2 種類の OS を用いる．メニーコアでは並列演算アプリケーションを高速に実行するために軽量 OS を用い，マルチコアではメニーコアのジョブ管理や I/O 処理を行うために汎用 OS を用いる．両 OS が連携してアプリケーションの資源管理を実施することで，両 CPU の特性を生かした高い演算性能を目指す．

本研究では，資源管理を両 OS が連携して実施することによる性能低下に着目し，低遅延かつ広帯域な資源管理方式を提案する．マルチコア・メニーコア混在型計算機の模擬環境上で本資源管理方式を試作し，プロセス管理と I/O 管理性能について評価を行う．

2 課題と目的

本計算機ではOSをまたいで資源管理を実施することによる性能低下が課題となる．OSをまたいだデータ転送速度が低い場合，ジョブの展開やI/Oアクセスに多大な時間を要する．また，OS間通信路の並列性が乏しい場合，複数の資源管理が同時に発生した際の待ち時間が増大し，メニーコアのスケーラビリティが低下する．これらの課題に対処するため，本研究ではOSをまたいだジョブ管理とI/Oを低遅延かつ広帯域に実施するための資源管理方式を提案する．OS間でのデータコピー回数を削減するほか，低遅延かつ並列なOS間通信方式を導入することで，単一OSで構成されるLinuxと同等の高い資源管理性能を実現する．

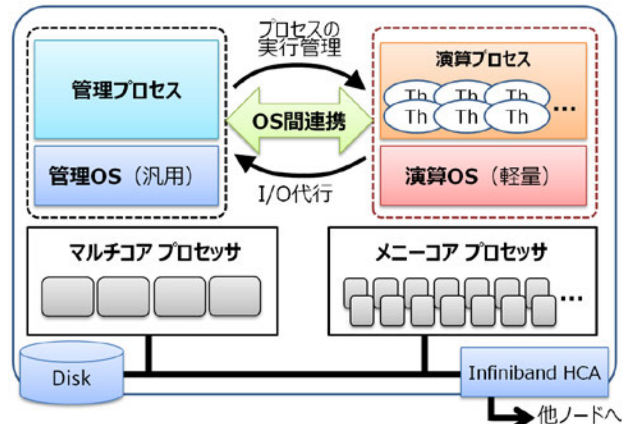


図 1 マルチコア・メニーコア混在型計算機のシステム構成

3 システム構成

3.1 ハードウェア構成

本研究で対象とする計算機は Intel 社が開発中の MIC[1]のように，汎用 CPU コアを多数集積したメニーコア CPU に，既存のマルチコア CPU を組み合わせたものである．メニーコア CPU とマルチコア CPU はそれぞれ独自のメモリとメモリ空間を有するが，互いのメモリをマッピングすることで直接アクセスが可能であるほか，CPU 間割り込み(IPI)を利用できる．ディスク等の I/O 機器はマルチコア側へ接続されており，メニーコアからの制御は限定される．

3.2 システムソフトウェアの構成

本計算機では，メニーコア CPU の豊富なコア資源を有効に活用することで並列演算性能を追求する．しかし，メニーコアのコア単体性能が低いため，メニーコア上では並列演算処理に特化した軽量 OS を用い，マルチコア上の汎用 OS で軽量 OS の動作を支援する．そこで，本研究では図1に示すようにメニーコア上で軽量なスレッド実行基盤とカーネルを特徴とした軽量 OS ”Future/MULiTh”を「演算 OS」として用いる．一方，マルチコア上では汎用 OS の Linux を「管理 OS」として用いる．演算 OS 上には資源割り当て単位である「演算プロセス」とプログラム実行実体である「演算スレッド」が存在し，演算プロセスは管理 OS 上の「管理プロセス」によって生成・管理される．

A Study of Hybrid Operating System for a Parallel Computer with Multi-Core and Many-Core Processors
Go Fukazawa†
† Department of Computer and Information Sciences, Graduate School of Engineering, Tokyo University of Agriculture and Technology

4 資源管理代行方式の設計

4.1 演算プロセスの管理代行方式

本計算機では演算プロセスを管理プロセスが管理するため，管理 OS が演算プロセスの生成・管理を行う．管理 OS で実施できないメニーコア CPU の初期化を除いたすべての処理を管理 OS が実施することで，OS 間通信頻度を削減した低遅延なプロセス管理を実現する．本手法ではメニーコア側メモリへのバイナリやページテーブルの展開方式が課題となる．OS 間での冗長なデータコピーはプロセス生成速度を低下させる要因となるため，本研究では演算 OS からメニーコア側メモリへ直接データを展開する．また，直接データ転送に必要な OS 間でのアドレス変換を O(1)で実現するアドレス変換表を構築することで，異種 OS が連携することによる遅延を極力削減したプロセス生成を実現する．なお，演算プロセスへのコア・メモリ資源の割り当ては静的に実施することで，アプリケーション実行中に資源管理処理が発生し，実行性能が低下する事態を避ける．

4.2 ファイル I/O の管理代行方式

本計算機における I/O 機器はマルチコア側へ接続されており，演算 OS からの制御が困難である．このため，演算 OS で発生したファイル I/O は管理 OS で代行処理する．ファイルシステムとデバイスドライバに管理 OS のものを用いることで，バックグラウンド処理や割り込み等の OS ノイズで演算 OS の軽量性が損なわれる事態を避ける．本方式では演算 OS から管理 OS へ I/O を依頼する際の遅延や，管理 OS からのデータ転送方式が課題となる．本研究では，管理 OS 内での処理をカーネル空間内で完結させ，特権レベル切り替えによる遅延を削減するほか，複数同時に依頼されたファイル I/O を並列に処理することで I/O 依頼にともなう遅延を削減する．また，管理 OS のファイルシステムからメニーコア側メモリ上の I/O バッファへ直接 I/O を発行することで，低遅延・広帯域に I/O データを転送する．これらの方策により，異種 OS が連携することによる性能低下が発生しにくいファイル I/O 方式を実現する．

4.3 OS 間通信方式

本計算機における OS 間通信路は，メニーコア CPU とマルチコア CPU から直接アクセス可能な共有メモリ上に構築したパケットキューと，CPU 間割り込み（IPI）で構成する．通信相手の受信キューへ直接パケットを書き込み，IPI で通信発生を通知することで，OS 間でのネゴシエーションを要さない低遅延な通信を実現する．また，パケットキューを演算プロセスごとに用意し，プロセス間での排他制御を除去することで，複数の資源管理を同時に実施する際のスケーラビリティを向上させる．

表 1　演算プロセスの生成時間

種別	所要時間	割合
演算プロセスの生成時間	114us	152%
〈参考〉Linux の プロセス生成時間	75us	100%

表 2　演算プロセス生成時間の内訳

項目名	所要時間	割合
演算プロセス生成処理	99us	86.8%
OS 間通信 / CPU 初期化処理	11us	9.6%
その他	4us	3.5%
合計	114us	100%

5 試作と評価

5.1 模擬環境を用いた試作

本研究では，Intel Xeon X5690（6 コア，3.47GHz）プロセッサを 2 個搭載した NUMA 型計算機をマルチコア・メニーコア混在型計算機に見立てて評価環境を構築した．一方の CPU をマルチコアとし，管理 OS である Linux を搭載した．もう一方の CPU をメニーコアとし，独自に開発中の演算 OS カーネル Future を実装した．

5.2 演算プロセス生成性能の評価

管理プロセスによる演算プロセス生成の所要時間およびその内訳を測定した結果を表1，表2に示す．直ちに終了する空のプログラムを用いたところ，比較対象の Linux の 1.52 倍で生成が完了した．全体の約 87%がプロセス生成処理であるが，冗長な実装を今後改良することで，Linux と同等のプロセス生成性能を実現できると考えている．なお，演算 OS ではデマンドページングを実施しないため，プログラムのバイナリサイズ増加にともない，プロセス生成処理の所要時間が増加する．しかし，MPI アプリケーションのように同一バイナリで複数の演算プロセスを生成する場合には，バイナリ転送を一本化可能な API を検討することでプロセス生成時間を削減できる．

5.3 I/O アクセス性能の評価

演算 OS から管理 OS へファイル I/O を依頼する際に生じる遅延の内訳を表3に示す．全体の約半分が OS 間通信にともなう遅延，残りが管理 OS 内での遅延時間となった．Shadow ページテーブルの探索は全体の約 7%で完了しているが，既存のページテーブルを用いた方式では少なくとも 2 倍程度の時間を要するため，Shadow ページテーブルの導入で全体の遅延時間を 6%程度削減したと言える．

次に，ファイル I/O の帯域幅を評価するため，管理 OS の RAM ディスク上のファイルへ演算 OS 側から read/write 関数を呼び出した際の所要時間を測定した．結果を図2に示す．Write 性能は比較対象とし

表３ I/0 代行処理の遅延時間内訳

項目名	所要時間	割合
OS 間通信	2.24us	45.8%
ワーカースレッド起床処理	1.85us	37.9%
アドレス変換	0.34us	6.9%
その他	0.45us	9.3%
合計	4.88us	100%

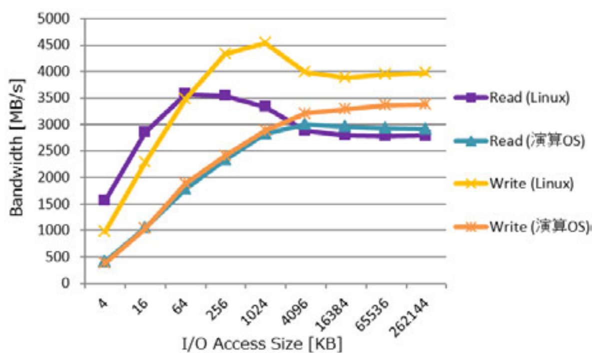


図２ I/0 代行方式の帯域幅

た Linux の帯域幅をすべての I/O サイズで若干下回り，256MB 単位では Linux の 0.85 倍の帯域幅を示した．Read 性能は Write とは異なり I/O サイズが 4MB 以上で演算 OS の帯域幅が Linux を若干上回った．256MB 単位では Linux の 1.05 倍の帯域幅を示した．Read で Linux を上回った要因としては演算プロセスへの静的メモリ割り当てが挙げられる．Linux ではデマンドページングによる例外処理で遅延が発生するため，性能が低下したと考えらえる．Write で十分な性能を発揮できなかった原因は現在調査中であるが，Linux は速度が低下する CPU をまたいだメモリアクセスを実施しないため，演算 OS での I/O よりも高い帯域幅を示したと考えている．

6 おわりに

本研究では，マルチコア・メニーコア混在型計算機において，管理 OS と演算 OS が連携して実施するジョブ管理方式と I/O 管理方式を提案した．OS 間でのデータコピー回数と通信遅延を削減することにより，低遅延・広帯域な資源管理を実現する．模擬環境上で提案した資源管理方式を試作・評価したところ，プロセス生成遅延は Linux の 1.52 倍，ファイルへの 4MB 以上の Read アクセスは Linux と同等の帯域幅であることを確認した．今後は，実環境での評価やベンチマークの実施を通して性能向上に向けた改良を行うとともに，I/O を集約化することで小サイズ I/O の帯域幅を改善する．

参考文献

[1] Intel: Many Integrated Core (MIC) Architecture – Advanced (online), Available via http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html (accessed June.26.2012).

[2] Giampapa, M., Gooding, T., Inglett, T., Wisniewski, R.W.: Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK, In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, pp.1-10 (2010).

[3] Shimizu, M., Yonezawa, A. Remote process execution and remote file i/o for heterogeneous processors in cluster systems, In Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID '10, pp. 145-154 (2010).

マルチコア・メニーコア混在型計算機における資源管理代行方式の試作と評価

東京農工大学工学府情報工学専攻・並木研究室
深沢 豪

背景（1/3） 2

次世代型スパコンの研究

- 現在：ペタフロップス → 次世代：エクサフロップス
  - Sequoia: 16PFlops, 京: 10PFlops, Mira: 8PFlops [1]
- 演算性能の向上手段
  - 並列演算性能を高める
    - CPU単体性能の向上は限界 → コア数を増やす
    - エクサ達成には 100コアのCPU x 10万個 が必要 [2]
      - 京：8コアのCPU x 88128個
  - アクセラレータ（GPGPU/メニーコア）による並列演算性能向上
    - スカラー演算に優れたマルチコア＋並列演算に優れたGPGPU/メニーコア
    - Intel MIC (Many Integrated Core)
      - 従来のマルチコアCPUへアクセラレータとして接続するメニーコア
      - Knights Ferry: 32コア, Knights Corner (Xeon Ph): 61コア

[1] TOP500 List – June 2012, "http://www.top500.org/ ist/2012/06/100"
[2] HPCI 技術ロードマップ白書, "http://www.open-supercomputer.org/workshop/sdhpc/", 2012

背景（2/3） 3

マルチコア・メニーコア混在型計算機の開発

- マルチコアとメニーコア（数百個）で構成した計算機でクラスタを構成し エクサスケールの演算性能を目指す
- メニーコアの欠点をマルチコアで補う
  - メニーコアはコア単体性能が低いため軽量OS上でアプリケーションを実行
    - ➡ マルチコア上の汎用OSが軽量OSの動作を支援
    - マルチコア（管理コア）: I/O処理・ジョブ管理を汎用OS（管理OS）で実行
    - メニーコア（演算コア）: 並列演算アプリケーションを軽量OS（演算OS）で実行
  - CPU間の高速・低遅延な内部バスで連携を行うことで ノード間で同様の構成をとるBlue Gene等よりも低遅延な連携を実現

背景（3/3） 4

プロセスモデルとI/O機器構成

- 「管理プロセス」と「演算プロセス/スレッド」
  - 資源割当単位 → 演算プロセス
    プログラム実行実体 → 演算スレッド
  - 管理プロセスは演算プロセスを管理
- I/O機器は管理コア側へ接続

管理OSによる演算コア側資源の代行管理

- 演算プロセスの生成・管理を管理OSが代行
  - 管理プロセスが演算プロセスを管理するため遅延削減に有効
- 演算OSで発生したI/Oアクセスを管理OSが代行
  - 管理コアへ接続されたI/O機器を演算OSから操作するのは困難
  - ファイルシステム・デバイスドライバによるOSノイズを削減可能

関連研究 5

- Bproc [1]
  - Linuxクラスタでシングルシステムイメージ環境を提供
  - I/Oと計算は同一ノードで処理される
- Blue Gene [2], 清水ら [3]
  - I/Oノード と 計算ノードにシステムを分離
    - I/Oノードが計算ノードに対して2次記憶へのアクセス機能を提供
  - ノード間ネットワークによりノード間でI/Oデータを転送

→ ノード単位で処理を分離

- FOS (Factored Operating Systems) [4]
  - マイクロカーネルによりCPUコアごとにOS機能を分散配置
    - アプリケーション・ファイルシステム・デバイスドライバが独立動作
  - デバイスドライバによるI/Oデータをメッセージに載せ ファイルシステムを介してアプリケーションへ転送

→ ノード内のヘテロ構成に未対応

[1] H ndrika, E.: Bproc he beowulf istributed process pace,
In Proces ings of he 16 h int rna l nal c nfer nce on Sup rc mpu ing, ICS '02, pp. 129- 38 (2002).
[2] Giampapa, M., et, l: E periences wi h a Lightweight Supercomput r K rn l
Less ns Le rned from Blue G ne's CNK, SC '10, pp. 1-10 (2010).
[3] Shimizu, M., et, l: Remote process execu i n and r mote le i/o for het rog neous process rs in clust r systems,
CCGRID '10, pp. 145-154 (2010).
[4] W ntzlaff, D., et, al.: An opera ing system for mul ic re nd l ud : mechani ms and implem nta i n, SoCC '10, pp. 3-14 2010).

課題 6

資源管理を代行することによる性能低下

- OSをまたいだプロセス生成とI/Oデータ転送
  - プロセスへの資源割り当てでOS間通信を実施することによる遅延
  - I/O機器から演算OSに至る過程で、 バイナリ・I/Oデータを複数回コピーすることによる時間増
- I/O要求集中への対処
  - I/O開始までの遅延時間により演算プロセスで発生する待ち時間
- OS間通信方式
  - OS内の処理やネゴシエーションによる遅延
  - OS間通信路の逼迫による待ち時間

プロセス管理の所要時間増大
I/Oアクセスの帯域幅とスケーラビリティ低下
⇨ 計算機全体の演算性能が低下

研究目標/方針 7

研究目標

- 異種OSの連携によるプロセス管理とI/Oアクセスを低遅延かつ広帯域に実施し，汎用OSと同等のプロセス管理・I/Oアクセス性能を実現する

研究方針

- 演算プロセスを低遅延に生成可能なプロセス管理方式を提案
  - プロセスへの資源割り当てを管理OS内で完結
  - 演算コアのメモリへのバイナリ展開をゼロコピーで実施
- 二次記憶へのアクセスを低遅延・広帯域に行うI/O代行方式を提案
  - I/Oデータの転送をゼロコピーで実施
  - I/O要求集中時に各々のI/Oアクセスを並列に処理
  - ノード間通信等のI/O方式は別途検討
- 低遅延かつ並列なOS間通信方式を提案

2012/09/11 JSASS13

システムの全体構成 8

- 各資源管理機能を処理内容に応じた適切なレイヤーへ配置
  - レイヤー内で処理を閉じることで特権レベル切り替え時間を削減
- CPUをまたいだメモリアクセスとCPU間割り込み(IPI)による通信

2012/09/11 JSASS13

演算OSのシステム構成 9

I/Oアクセス委託機構

- 管理OSによるI/Oアクセス代行機能を演算プロセスへ提供
  - 本機構は低水準ファイルI/O機能を提供
  - 「ストリームI/O機構」により高水準ファイルI/O機能を提供

CPUコンテキスト管理機構

- 演算プロセスの生成・管理に必要なCPUコンテキスト設定機能を管理OSへ提供

OS間通信機構

- OS間共有メモリとIPIによるパケット通信機能を両OSへ提供
  - ユーザ・カーネルの各空間内で通信処理を完結

2012/09/11 JSASS13

管理OSのシステム構成 10

演算プロセス管理機構

- 演算プロセスの生成・実行管理機能を管理プロセスへ提供
  - 未使用資源を管理OS側で保持
    - 資源割り当てを管理OS内で完結
  - メニーコアのメモリへ直接アクセス
    - バイナリ等の展開を低遅延に実施

I/Oアクセス代行機構

- Linuxのファイルシステムへのアクセス機能を演算OSへ提供
  - メニーコアのメモリへ直接アクセス
    - ゼロコピーによる低遅延・広帯域なI/O
  - 複数のワーカースレッドで演算OSからのI/O要求を並列に処理
    - 演算OSのI/O待ち時間を削減

2012/09/11 JSASS13

管理OSと演算OSのAPI 11

管理OSのAPI

- 演算プロセスの実行管理APIを提供
  - 複数の演算プロセスを1つの管理プロセスから生成・管理可能

API種別	処理内容	OS間連携
lwp_create, destroy	演算プロセスの生成・破棄	○
lwp_suspend, resume	演算プロセスの停止・再開	○
lwp_wait	演算プロセス終了まで待機	×

演算OSのAPI

- I/O代行を隠蔽する低・高水準APIを提供
  - 低水準APIは演算OSの擾乱低減に有効
- 軽量スレッドライブラリを制御するPOSIXスレッドAPIを提供

API種別	処理内容	OS間連携
exit	軽量プロセスの実行終了	○
open, close, read, write, lseek, fsync, fdatasync	低水準I/Oアクセス	○
fopen, fclose, …	高水準I/Oアクセス	○
pthread_*	POSIXスレッドの制御	×

2012/09/11 JSASS13

演算プロセス管理（1/2） 12

演算プロセス生成の流れ

- 演算プロセスへ割り当てる資源(物理メモリ, コア)の取得
- ディスク中のバイナリと作成したページテーブルをメニーコア側メモリへ直接展開
- 演算OSへハードウェアコンテキストの初期化を依頼
- メニーコアCPUを初期化(ページテーブル・エントリーポイントの設定)

2012/09/11 JSASS13

演算プロセス管理（2/2） 13

低遅延・低負荷なプロセス管理方式

- 管理OSによる演算プロセスの生成・実行
  - CPUコンテキスト設定以外の処理を管理OS内で実施
    - 演算OS単体でプロセスへの資源割り当て，コンテキスト操作を実現
      - OS間通信の頻度削減 → プロセス生成オーバーヘッドの減少
  - バイナリとページテーブルをメニーコアのメモリ上へ直接展開
    - 単一構成のOSと同等の遅延時間で演算プロセスを生成
- 演算プロセスへの静的なコア・メモリ割り当て
  - アプリケーション実行に十分な量のコア・物理メモリを静的に割り当て
    - プロセスマイグレーション，ページフォルト等のOSノイズによるアプリケーション実行性能低下を避ける
  - プログラマからのヒントに基づいた割り当て量
    - アプリケーションの挙動を熟知したプログラマが，使用コア数・Heap/Stack領域の静的割り当て量を決定
      - Heap/Stack領域が枯渇すると，拡張処理によるOSノイズが発生

2012/09/11 JSASS13

I/O管理（1/2） 14

I/Oアクセスの流れ

- I/Oライブラリから管理OSへI/Oアクセスを依頼
- I/Oバッファの演算OS用アドレスを管理OS用アドレスに変換し，ファイルシステムへI/Oアクセスを要求
- ファイルシステムは演算プロセスのI/Oバッファへ直接I/Oアクセス

2012/09/11 JSASS13

I/O管理（2/2） 15

低遅延・広帯域なI/Oアクセス代行方式

- 演算OSで発生した二次記憶へのI/Oを管理OSが代行処理
  - I/O機器からI/Oバッファへ直接データを転送
    - 無駄なコピーを排除 → 低遅延・広帯域なI/Oデータ転送
  - 複数発生したI/Oアクセスを同時に処理
    - 演算OSでの待ち時間を削減 → スケーラビリティ改善

ゼロコピーI/Oのためのアドレス変換

- ゼロコピーI/Oアクセスの課題
  - I/Oバッファのアドレスが両OSで異なる
    - 演算プロセスの仮想アドレス vs. Delegateeプロセスの仮想アドレス
- 仮想→仮想 変換を行うアドレス変換表を構築
  - 両OSのページテーブルを参照する方式とは異なり，O(1)でアドレス変換が完了

2012/09/11 JSASS13

OS間通信 16

OS間共有メモリとIPIによるOS間通信

- OS間共有メモリ上でパケットを交換
  - 通信相手のキューへパケットを直接書き込むことでネゴシエーションを省略
    - OS間での同期処理を排除 → 低遅延な通信を実現
  - 演算プロセスごとにパケットキューを確保
    - プロセス間同期処理を排除 → スケーラビリティ改善
- 通信発生の検出は，ポーリングとIPIに対応
  - IPIのベクタ番号で通信が発生したキューを判別

宛先の動的決定による管理OSの負荷分散

- 管理OSへの通信の宛先をラウンドロビン方式で決定
  - 管理OSのコア数 ＜ 演算OSのコア数
    - 管理OSへの通信は可能な限り分散させる必要がある
  - ラウンドロビンは軽量だが，厳密な負荷分散は難しい
    - 必要に応じて管理OSの負荷を反映可能なアルゴリズムを検討

2012/09/11 JSASS13

試作・評価環境 17

- 2個のマルチコアCPUを管理コア・演算コアに見立てたマルチコア・メニーコア混在型計算機の模擬環境
  - 開発中の演算OSは1コアかつカーネル空間のみで動作
  - OS間通信相手への通知にはIPIを用いた

- Linux Kernel 2.6.18
- Intel Xeon X5690 3.46GHz 6core x2
- DDR3-1333 12GB x2

CPUをまたいだメモリアクセス
Read：1.01倍
Write：0.69倍

2012/09/11 JSASS13

演算プロセス生成性能を評価 18

直ちに終了する最小プログラムを演算プロセスとして生成

- 演算プロセス生成処理のオーバーヘッド
  - OS間通信の割合を低く抑えることに成功
    - 資源管理代行方式の遅延の主要因
    - ゼロコピー通信が有効
  - 演算プロセス生成処理には高速化の余地がある
- プロセス生成 → 終了までの時間をLinuxと比較
  - OS間通信を要さないLinuxに対して，0.47倍の時間増で収まっている
  - ELFパーサー・ローダーに性能改善の余地あり

実行環境	所要時間	比率
演算OS上での実行	110us	147%
Linux上での実行	75us	100%

プロセス生成処理（99us）
- ELFのパース，ロード
- ページテーブルの作成

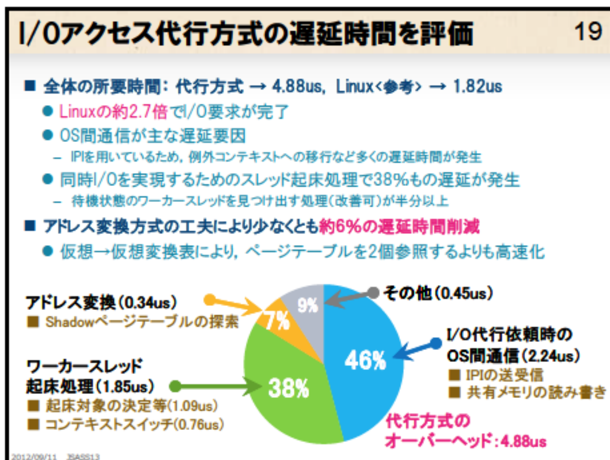
演算OS初期化処理 / OS間通信（11us）
- パケット送受信
- プロセスコンテキスト作成

10%
87%

2012/09/11 JSASS13

I/Oアクセス代行方式の遅延時間を評価　19

- 全体の所要時間：代行方式 → 4.88us，Linux<参考> → 1.82us
  - Linuxの約2.7倍でI/O要求が完了
  - OS間通信が主な遅延要因
    - IPIを用いているため，例外コンテキストへの移行など多くの遅延時間が発生
  - 同時I/Oを実現するためのスレッド起床処理で38%もの遅延が発生
    - 待機状態のワーカースレッドを見つけ出す処理（改善可）が半分以上
- アドレス変換方式の工夫により少なくとも約6%の遅延時間削減
  - 仮想→仮想変換表により，ページテーブルを2個参照するよりも高速化

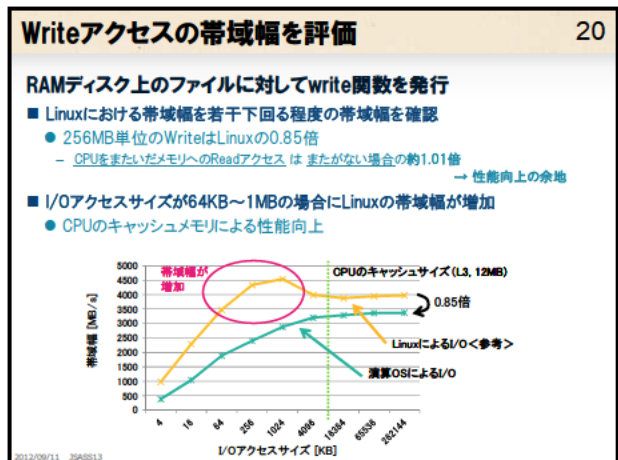


2012/09/11　JSASS13

Writeアクセスの帯域幅を評価　20

RAMディスク上のファイルに対してwrite関数を発行

- Linuxにおける帯域幅を若干下回る程度の帯域幅を確認
  - 256MB単位のWriteはLinuxの0.85倍
    - CPUをまたいだメモリへのReadアクセス は またがない場合の約1.01倍
      → 性能向上の余地
- I/Oアクセスサイズが64KB～1MBの場合にLinuxの帯域幅が増加
  - CPUのキャッシュメモリによる性能向上

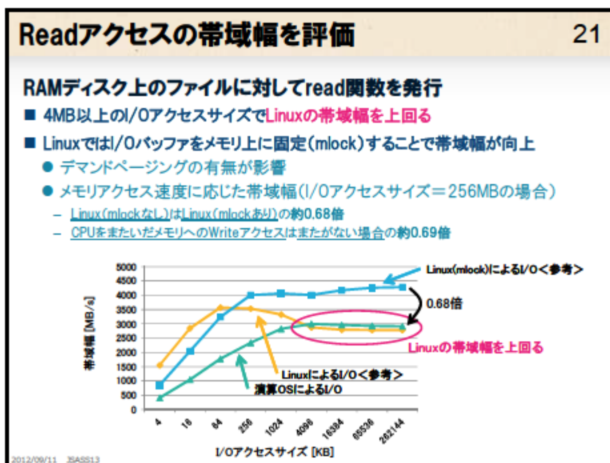


2012/09/11　JSASS13

Readアクセスの帯域幅を評価　21

RAMディスク上のファイルに対してread関数を発行

- 4MB以上のI/OアクセスサイズでLinuxの帯域幅を上回る
- LinuxではI/Oバッファをメモリ上に固定(mlock)することで帯域幅が向上
  - デマンドページングの有無が影響
  - メモリアクセス速度に応じた帯域幅（I/Oアクセスサイズ＝256MBの場合）
    - Linux(mlockなし)はLinux(mlockあり)の約0.68倍
    - CPUをまたいだメモリへのWriteアクセスはまたがない場合の約0.69倍



2012/09/11　JSASS13

考察　22

演算プロセス生成方式

- バイナリサイズが小さい場合は低遅延なプロセス生成を実現
  - バイナリサイズに比例して遅延時間が大きくなる方式
    - 同一バイナリで複数プロセスを生成する際の処理を一本化するAPIが必要
      - MPIアプリケーションの生成に応用可能

I/Oアクセス代行方式

- OS間通信の遅延時間が大きい
  - I/Oサイズを大きくしなければ帯域幅を稼げない
    - Read, Writeの帯域幅は4MB単位以上でLinuxとの差が収束した
  - OS間でのネゴシエーション排除のみでは，遅延時間の削減に限界がある
    - IPIとポーリングの使い分けが課題として残る
- ゼロコピーI/OによりLinuxと同等のファイルI/O帯域幅を実現
  - メニーコア側メモリへのデータ転送速度が模擬環境と同等であれば，次世代スパコンへ適用できる
    - マルチコアとメニーコアを内部バス（QPI）で接続したアーキテクチャが必要

2012/09/11　JSASS13

まとめ/今後の課題　23

まとめ

- ✓ 管理OSと演算OSの協調による資源管理方式を提案
  - 管理OSによる演算プロセスの生成・実行管理
  - 演算OSで発生したI/Oアクセスを管理OSで代行処理
- ✓ 汎用OSと同等のプロセス生成・I/Oアクセス性能を確認
  - ゼロコピーのデータ転送と，低遅延なOS間通信

今後の課題

- 模擬環境からの脱却
  - 模擬環境では再現できないコア単体性能・CPU間通信方式を検証
- 演算OSによる擾乱の軽減具合をベンチマークで確認
  - 並列演算とファイルI/Oが含まれるベンチマークでAPP実行性能を評価
- MPI-IO，並列ファイルシステムへの対応
  - 高度なI/Oアクセスの集約，I/Oアクセス並列度の向上を図る

2012/09/11　JSASS13

編集後記

今年で第 13 回となる JSASS2012 は農工大での開催でした．農工大での開催は，2001 年と 2005 年に引き続き 3 回目・7 年ぶりとなります．まずは，ご多忙にもかかわらず，会場の準備や懇親会の準備に骨を折って頂きました並木先生に心から深くお礼申し上げます．ありがとうございました．

また，今回は，農工大，名工大，拓殖大，立命大から計 13 件の発表がありました．発表された方々，ご討論に参加された方々を始め，参加して頂きました皆様に深くお礼申し上げます．JSASS が，皆様の研究推進と交流に少しでも役立っていたらいいなと，心から願っております．

ということで農工大を訪れるのは私にとっても 7 年ぶりだと思うのですが，久々に訪れて驚いたことが 3 つありますので，紹介したいと思います．まず 1 点目ですが，次ページ上側の写真にあるように東門にアーチができていたことです．同時に，東門入ってすぐのあたりがおしゃれな感じになっています．特に左手のところにはエリプスという名前の素敵な建物ができており，きれいなカフェがあるとともに研究・研究交流の拠点にもなっているようです．とてもいいですね．以前は何でしたっけ．駐車場だったような気がします．実は 7 号館の変化にもびっくりしました．ほとんどは以前のままだったのですが，なんと，トイレがリニューアルされていました！このリニューアルを決断された人はとてもえらいなと感じました．限られた予算と時間の中で，最大限の効果を得る決断をされたと思います．その絶大な効果は，普段 7 号館にいらっしゃる方々こそが実感されているに違いないと思います．閉塞感が漂う日本を救うキーワード「スピード」と「英断」がそこにあると思います．素晴らしい．

次に 2 点目ですが，やはり JR 中央線の高架化でしょうか．いえ，その前に中央線の車両が，ステンレス製の若々しい車両になっている点が実は私にとっては新鮮です．山手線や京浜東北線，中央・総武線などは

かなり前からステンレス車両でしたが，なぜかいつまで経っても普通鋼だった中央線．老体に鞭を打ちながら重そうに走っているなぁと感じながら乗っていたものです．今は軽快に走っています．日本を救うキーワード「若返り」がここにあるんだなと実感します．

そして最後がJR中央線の高架と東小金井駅のリニューアルです．鉄道関係の話ばかりで恐縮ですが．特に駅北側（農工大と反対側）は注目に値します．2009年に高架化が完了し，もう3年経つにもかかわらず何もない…．いや，あのスタ城（居酒屋．スタミナの城の略）はあるんですよ．でもスタ城以外に何もない…．なんということでしょう．では，南側（農工大側）はどうかと言うと，やっぱりあの狭いエリアにラーメンと中華のお店ばかりがあって，分野の偏り具合が相変わらずでした．ちなみに，スタ城は創業40周年だそうです．日本を救うキーワード「根性」がここに感じ取れます．

編集後記だからと言って，好き放題書いておりますが…．このシステムソフトウェアの分野，世界的にはいろいろ盛り上がっています．例えば国際会議ではSOSP, OSDI, ASPLOS, EuroSysにAPSysなど徐々に数を増やしていますし，参加者も増加しているようです．今ホットな話題のスマートフォンも，iOSだのAndroidだの，OSの話に事欠きません．にもかかわらず国内はパッとしないようなそんな感じはありませんか？私はそう感じていたのですが，最近はそうでもないのかなと感じています．大学のシステムソフトウェアの研究者は数が多くないとしても活躍していますし，聞いてみると，企業でもLinux等のコミッターがいたり，基盤技術を商品化していたりします．あれ？国内も結構盛り上がってるよね？そうだと思うんです．ただ，そういった人たちをうまく評価し，表に出てきてもらえる枠組みが必要かなと感じています．今後やらないといけないのは，その辺かな．

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 2012 (JSASS2012)

6巻1号（通号6号）オンライン版　2012年10月29日発行
© JSASS実行委員会

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