

先進的計算基盤システム  
シンポジウム 2009

Symposium on Advanced Computing  
Systems and Infrastructures 2009  
(SACSYS 2009) 報告

尾上 浩一

# SAC SIS 2009

- 会場：広島国際会議場
- テーマ
  - 先進的計算システム
    - グリッド計算、オーバーレイネットワーク等
  - 先進的計算システムを支える基盤技術
    - プロセッサアーキテクチャ、OS・ミドルウェア、コンパイラ、並列アルゴリズム等
  - 実用的基盤技術
    - データベース、Webサービス、分散計算環境等

# SAC SIS 2009の概要

- 招待講演
  - Don Grice(IBM)
  - “The Roadrunner Project and the Importance of Energy Efficiency on the Road to Exascale Computing”
- 一般論文:38件
- ポスター発表:42件
- プログラミングコンテスト
  - Cellチャレンジ 2009
  - Clusterコンテスト
- チュートリアル:2件

# SACSYS 2009プログラム(1/2)

- 省電力コンパイラ
- ネットワークオンチップ
- 分散計算プラットフォーム
- 大規模ネットワーク
- セキュリティ基盤
- メニーコア技術
- シミュレーション
- ストレージ

# SACSYS 2009プログラム(2/2)

- 並列システム評価
- 専用アーキテクチャ
- マイクロアーキテクチャ
- GPGPU
- キャッシュ
- スケジューリング

リーク電力削減のための  
コンパイラによる  
細粒度スリープ制御

薦田登志矢 佐々木広 近藤正章 中村宏

省電力コンパイラ

# 概要

- コンパイラによる細粒度スリープ制御手法の提案
  - パワーゲーティング(PG)制御によるリークエネルギーの削減
  - 演算ユニットの使用間隔の期待値を解析し、該当演算ユニット使用後にスリープモードへ移行させるかどうかを命令で指定
    - Control Flow Graph (CFG)内のノード間の距離の解析
  - ハードウェアによる手法との組み合わせによるハイブリッドなPG制御も提案

# 計算機の高性能化に伴う消費電力や放熱の増加が深刻化

- マルチコアでは処理性能の向上に加え、増加する消費電力をいかに抑えるかが大きな課題である
- 将来の半導体プロセスのさらなる微細化に伴い、リーク電流の割合は増加すると予測される
- 組み合わせ回路はトランジスタ当たりのリーク電力が大きく、演算器のリーク電力を削減することも重要
  - キャッシュにおける実行時のリーク電流を削減する手法の提案  
[Flautner et al., 2002]
- 性能やエネルギー面において、オーバーヘッドが小さいパワーゲーティング手法が有望

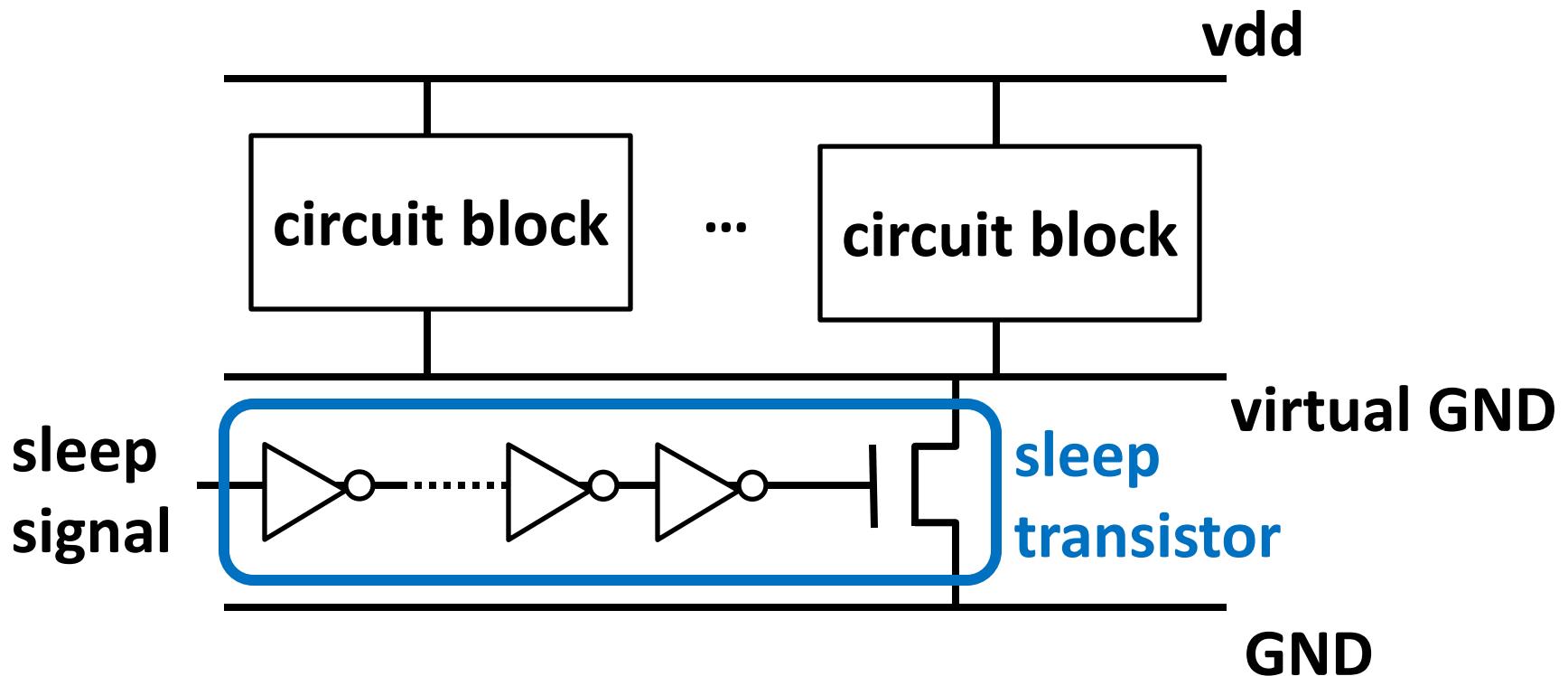
## cf.) CMOS LSIチップの消費電力

- ダイナミック消費電力
  - トランジスタのスイッチング
- リーク電流による消費電力
  - 電源電圧が供給されている限り発生

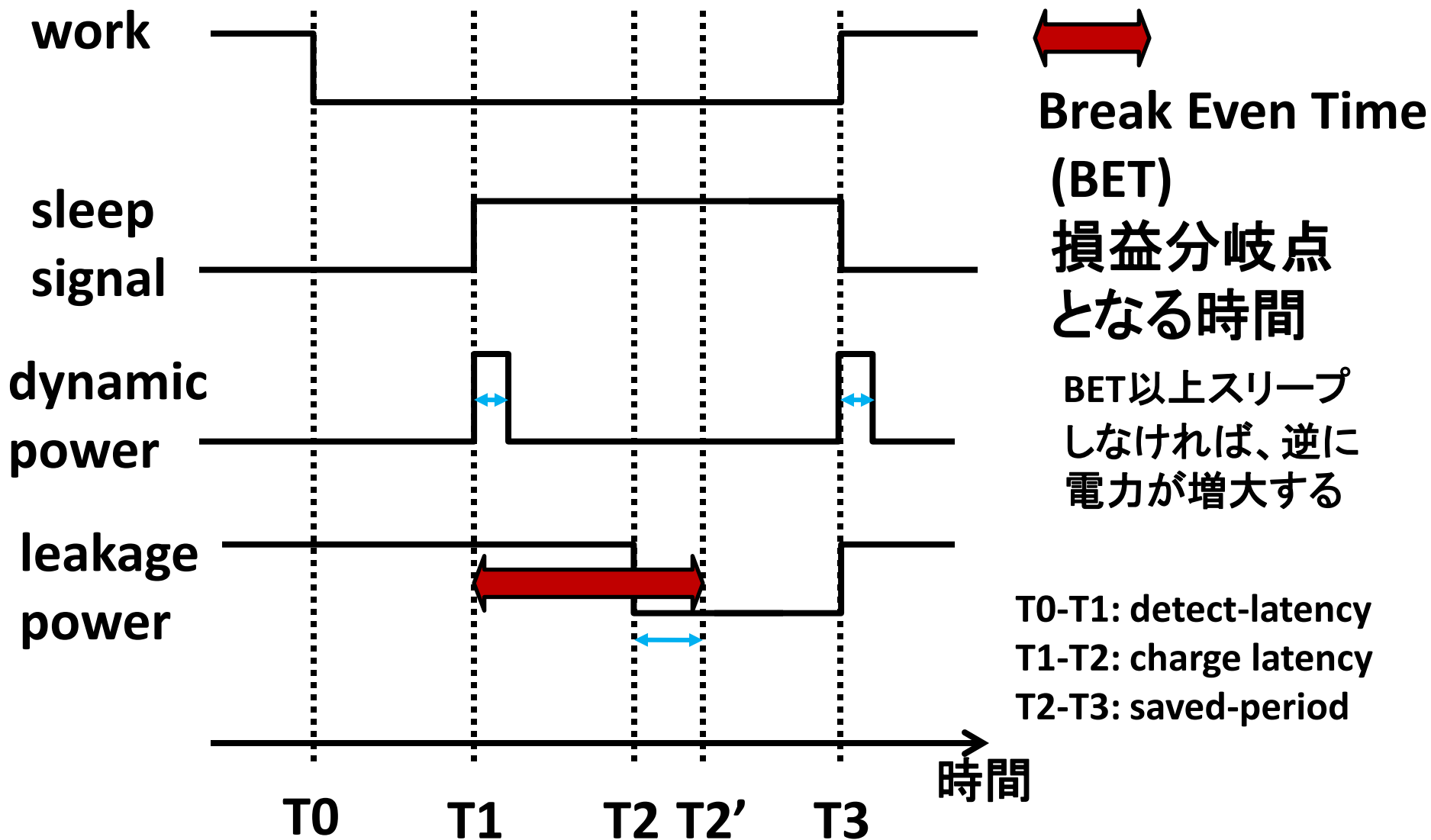


# パワーゲーティング(PG)制御

- 動作させる必要のない回路への電源供給を遮断し、リーク電流を削減する

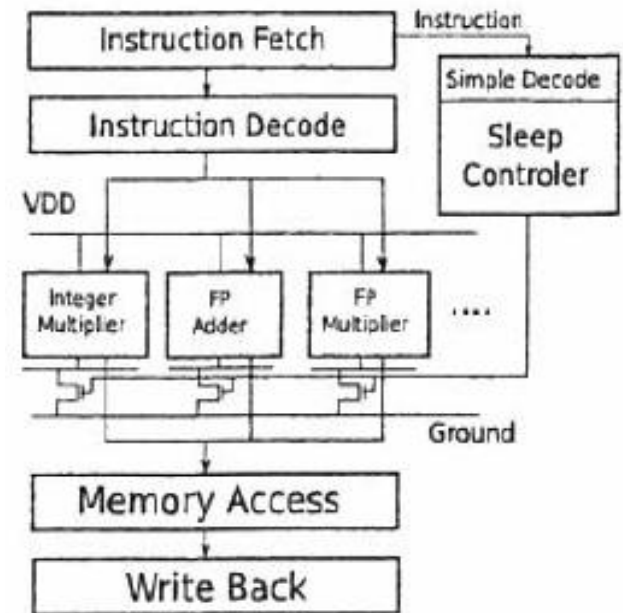


# PG制御によるスリープモード切り替え



# 対象アーキテクチャ

- Geysler(MIPS R3000互換)  
+ 浮動小数点ユニット
- 単一命令・インオーダー発行
- 演算ユニットごとにPG制御が可能



# 提案手法

- ハイブリッドなPG制御
  - コンパイラによるPG制御
  - キャッシュミスをトリガーとしたPG制御
- 効率的な実行時リーク電流削減のためには、「どの」回路に対して、「いつ」PG制御を適用するかを見極めることが必要

# コンパイラによるPG制御

- 演算実行後、スリープするかどうかの制御情報を命令に付加
  - 演算ユニットを使用する命令ごと
  - 命令セットアーキテクチャの変更が必要
- スリープ可能時間(の期待値)を予測

**BET = 20 cycle**

---

**mul.on (実行後スリープしない)**

↕ (5 cycle)

**mul.off (実行後スリープする)**

↕ (25 cycle)

mul: 乗算命令

# コンパイラを用いたPG制御の利点

- 追加のハードウェア機構をプロセッサに追加する必要がない
  - PG制御に関する予測をハードウェアで行うと、予測を行なうハードウェアによる消費電力が生じてしまう

# スリープ可能時間の解析 (1/2)

- コンパイラによるコード最適化で用いられている、Control Flow Graph (CFG)を解析し、演算ユニットのスリープ可能時間の期待値を求める
  - ノード:コード内の各命令
  - エッジ:制御の流れ
- 例)乗算器の休止可能時間の期待値を求める場合
  - CFG上の各乗算命令ごとに後続の乗算命令との間に存在する平均的なノード数を計算する
  - 各命令が1サイクルで終了すると仮定すれば、上述のノード数が、各乗算命令が実行された直後からの乗算器のスリープ可能サイクル数の予測値となる
    - パイプラインが理想的に流れている場合は、実質的に1サイクルごとに1命令が実行されるため、無理のある仮定ではない
    - スーバースカラ等への対応は今後の課題である

# スリープ可能時間の解析 (2/2)

- (関数呼び出しがない)関数内における解析
  - データの流れの解析[\*]と同様
- 関数間をまたぐ解析
  - リージョンベースな解析[\*]と同様
  - 呼び出す関数の先頭に目的命令が存在すると仮定し、CFGを利用する解析方法も考えられるが、この方法では精度の良い予測が困難

詳細は付録に記載

[\*] “Compilers: Principles, Techniques, and Tools”  
A. V. Aho, et al., 1986



# キャッシュミスをトリガーとした手法

- L2キャッシュミスが発生した時に、すべての演算器をスリープさせる
  - キャッシュミスはプロセッサをストールさせる大きな要因のひとつ

# 評価 (1/2)

- SimpleScalar Tool Set を利用
  - 消費エネルギーを評価した演算ユニット
    - 整数乗算器 (Integer multiplier)
    - 浮動小数点ALU (FP ALU)
    - 浮動小数点乗算器 (FP multiplier)
- ベンチマークプログラム
  - SPEC CPU2000
  - MiBench
- BETを20, 80に設定

# 評価 (2/2)

- 以下のPG制御手法を適用した場合の評価
  - compiler-inter
    - コンパイラのPG制御
  - compiler-intra
    - 関数間をまたぐ解析を行わないコンパイラのPG制御
  - L2
    - L2キャッシュミス発生時にスリープする手法
  - hybrid
    - コンパイラのPG制御とL2キャッシュミスによるPG制御
  - best compiler
    - 平均的なアイドル時間をコンパイル時に完全に予測できると仮定した場合のコンパイラ制御

# 評価結果

- Hybridは、BET=80におけるswim, mgirdのFP加算器を除く、すべてのケースにおいて、最も高いエネルギー削減効果を達成している
  - コンパイラまたはL2ミス制御を単独で適用した場合と比べ、大きなエネルギー削減効果を達成している
- compiler-interはcompiler-intraに対し、BET=80, mgird以外大きなエネルギー削減効果を達成している
- compiler-intraでは、equakeの整数乗算器のようにほとんど使用されていない場合に対し、スリープする機会を逃してしまう
- L2キャッシュミスによる制御は、swim, mgird, artにおけるFP加算器においてエネルギー削減効果が大きい
- 別紙の図8-13参照
  - 縦軸: スリープ制御を適用しない場合を1としたときの相対的なリークエネルギー

# 省電力化を実現するための既存手法

- 実行時の情報を利用した手法
  - プログラム全域にわたる、グローバルな最適化が困難
  - 実行時電力制御のための処理やハードウェア追加が必要となるため遅延が大きい
- 静的な情報を利用した手法
  - プログラムの解析による詳細な情報が利用できるため、きめ細かな電力制御が可能

# 実行時の情報を利用した手法

- Adaptive Processing [Albanesi et al., 2003]
  - プログラム中の各フェーズにおける付加を判断し、不要なりソースを停止させる
    - キャッシュミス回数測定用カウンタ、命令キュー等を利用
- Online Methods for Voltage and Frequency Control [Wu et al., 2004]
  - 実行時の負荷に応じた周波数・電圧制御 (FV制御)
- プロセッサ間の負荷不均衡に対し、Dynamic Voltage Frequency Scaling やタスク再配置を行なう手法 [高務ら, 2005]
  - 各プロセッサの負荷の分散度を計算
- Thrifty Barrier [Li et al., 2004]
  - ループ操作の不均衡に対する電力制御

# 静的な情報を利用した手法

- compiler-directed DVS [Hsu et al., 2003]
  - シングルプロセッサの低消費電力化手法の提案
- コンパイラによるPG制御
  - [Rele et al., 2002]
    - 議論の中心が性能的なオーバーヘッドについて
  - [Roy et al., 2007]
    - ループ階層構造に着目したプロファイリングを利用

# メッセージ衝突を防止する適応的な データ収集操作アルゴリズム

吉富翔太 弘中健 田浦健次郎

大規模ネットワーク



# 概要

- 大規模なネットワークに対しても適応可能で、効率的かつスケーラブルなデータ収集操作 (gather操作) を行なうアルゴリズムの提案
  - ネットワークのコンテンションが起きないように通信を制御
  - ノード間の同期操作の頻度を削減
  - 複数の既存手法のアルゴリズムに比べ、高い性能が得られることを確認
  - 大規模な環境でもスケーラブルであることを確認

# gatherにおける問題点

- コンテンションの発生により通信性能が悪化
  - ルータ・スイッチへの通信集中によるデータロス
  - データの再送
- コンテンションを防ぐために、ノード間で逐一同期しながら通信した場合、高遅延なネットワークでは性能がかえって低下してしまう
  - 同期に要する時間がノード間の遅延に依存

## cf.) gather操作

- ネットワークの複数のマシンからデータを収集する通信
- 例)・複数台の計算機で処理させたデータを1箇所に集積させる  
ような、MPIにおける並列計算処理
  - ・ NFSにおいて、複数ノードから1台のサーバへのデータ集積処理

# 既存の手法

- 既存のMPIライブラリ  
(OpenMPI, MPICH, MagPie)
  - ルートノードを頂点とするツリー構造を生成
  - 各送信ノードは同期をとらない
- 組み合わせ最適化問題としてのgather
  - バンド幅・遅延・メッセージサイズのパラメタを利用して、通信コストを算出
  - 実ネットワーク上のどの経路で通信するかを考慮していない

# 目標

- コンテンションを防止しつつ、ネットワークの遅延の大小を問わず様々な環境に適応する gather アルゴリズムの提案

## コンテンションが発生する条件

- あるルータ・スイッチにおいて、複数ノードからの同一時刻に同一宛先ノードへのメッセージが重複する場合

# 提案手法

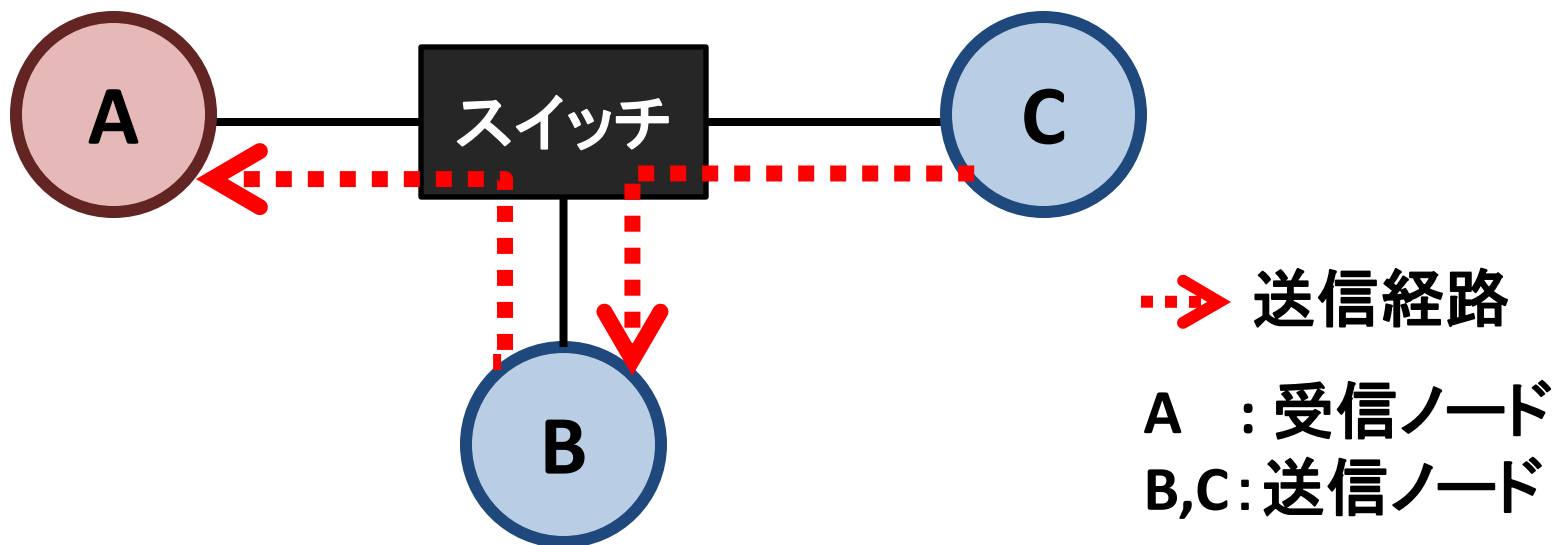
- 複数のノードから「同時」に「同一宛先ノード」へメッセージが流入しないようにする
  - 基本スケジューリング
    - パイプライン転送
    - タイミング制御(同期通信)
  - ネットワーク全体のスケジューリング
    - ネットワーク全体にパイプライン転送を適用し、初期のメッセージの流れ(通信ツリー)を決定
    - 初期の通信ツリーの性能向上の余地があれば、タイミング制御を用いて、通信ツリーを再構築

# 想定しているネットワーク

- 通信経路が事前に決定されたネットワーク
  - トポロジー・遅延・バンド幅
  - 構成情報の取得に要するコストは0とする
- ネットワーク構成が途中で変化しない
  - ノードの参加・脱退等は考慮しない
- 1ノードは受信ノードが相異なる複数のgatherを同時に行わない
  - 他の通信に影響されない

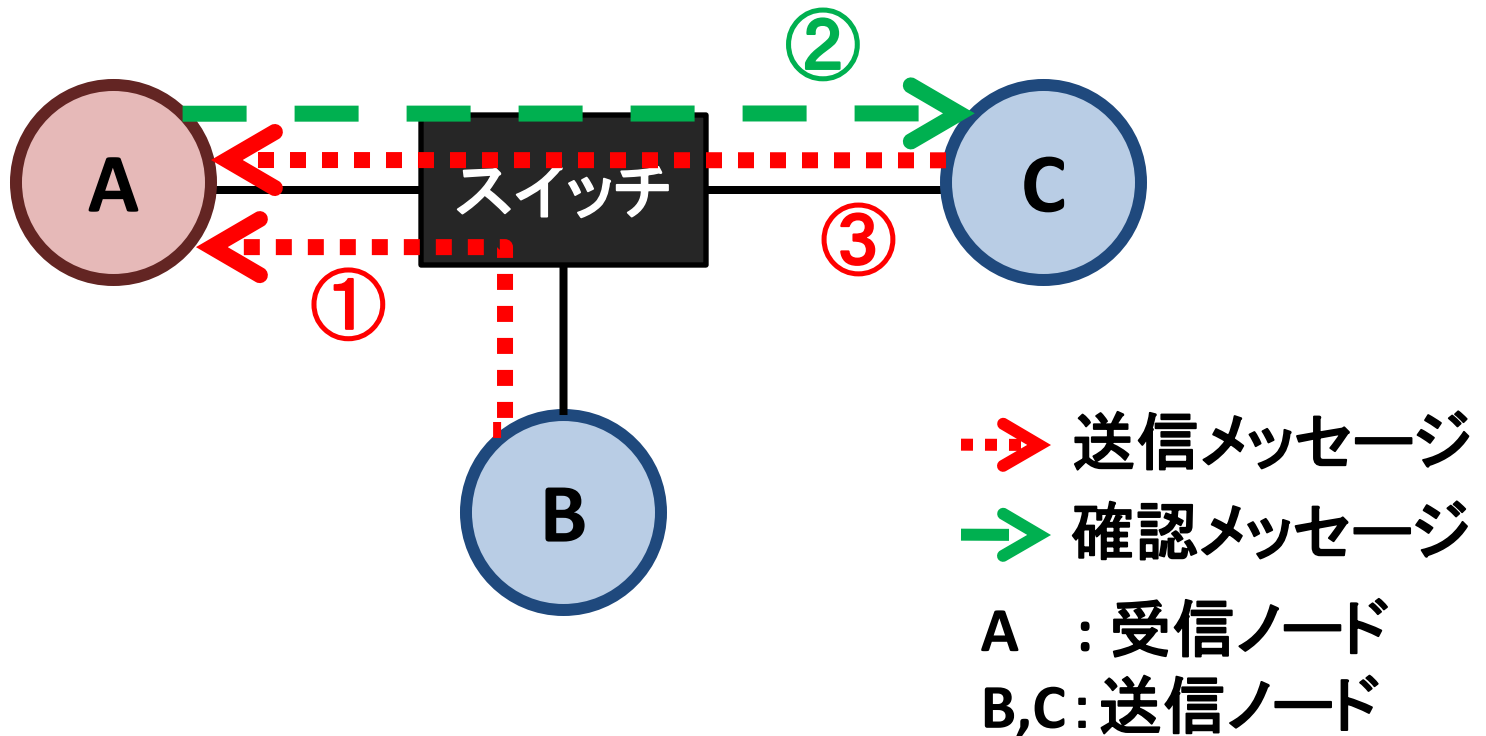
# パイプライン転送

- ノードBはノードAに対してのみ独占的にメッセージを送る
- AはB以外の別のノードからメッセージを受け取らない



# タイミング制御（同期制御）

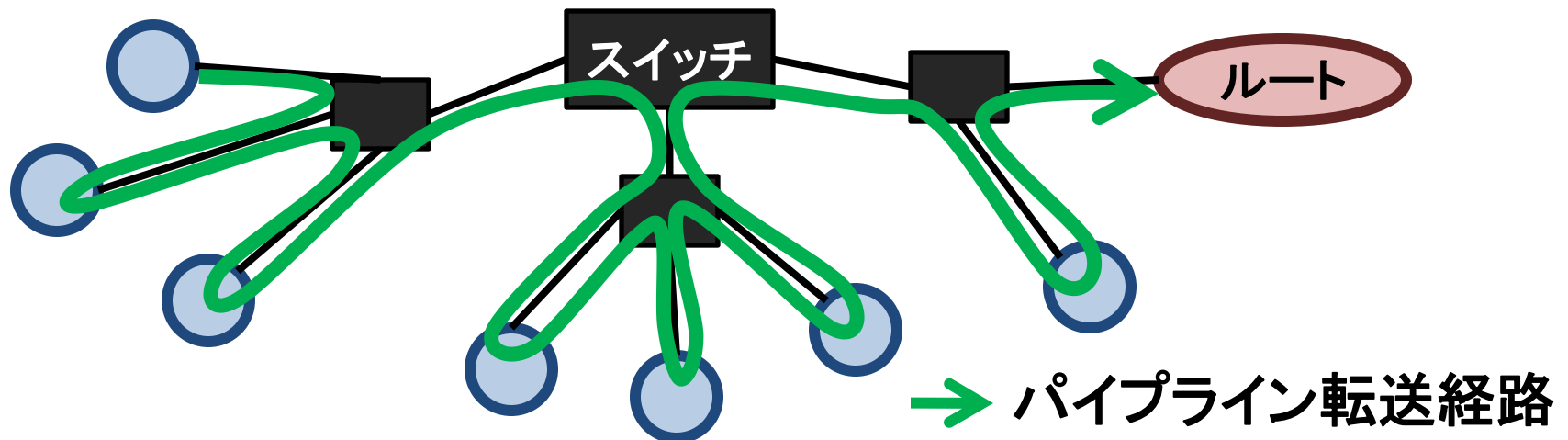
- 複数の送信ノードから同時にメッセージを流入させない





# 初期の通信ツリーの構築

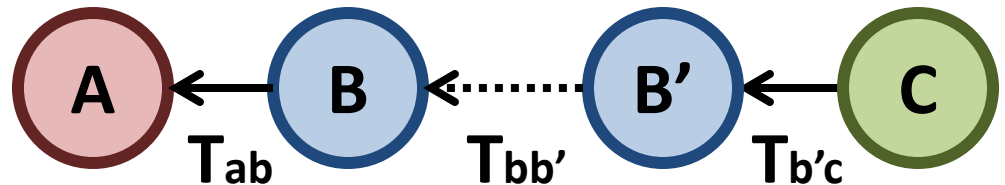
- 以下の手順で、ルートノードに対する通信ツリーを構築
  - ルートノードから深さ優先探索で走査
  - 逆順をパイプライン転送順とする
- 探索で複数のノードの選択肢が存在する場合、ルートノードとのバンド幅が大きいノードを選択
  - バンド幅が小さいリンクがボトルネックとならないようにするため



# 通信ツリーの再構築 (1/2)

- ノードAからノードCへの最小の通信所要時間
  - パイプライン転送

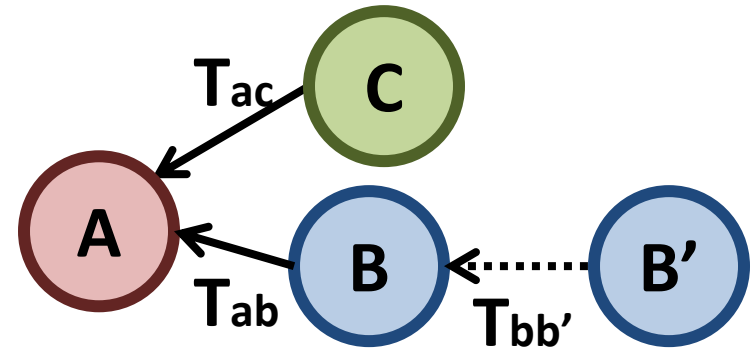
$$T_p = T_{ab} + T_{bb'} + T_{b'c}$$



- タイミング制御

$$T_t = T_{ab} + T_{bb'} + T_{ac} + L_{ac}$$

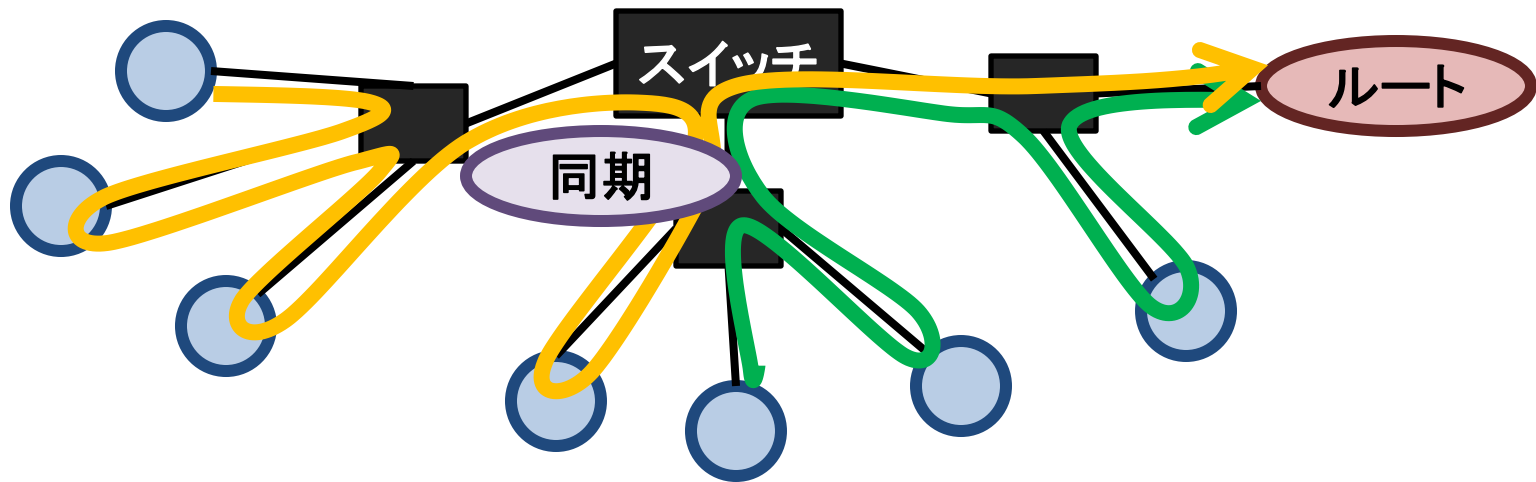
$L_{ac}$  : 確認メッセージ



- タイミング制御を導入する条件
  - $T_p > T_t \iff T_{b'c} > T_{ac} + L_{ac}$

# 通信ツリーの再構築(2/2)

- 再構築は初期の通信ツリーの先頭から順に行なう
  - 後続のノードのツリーの形を維持したまま再構築

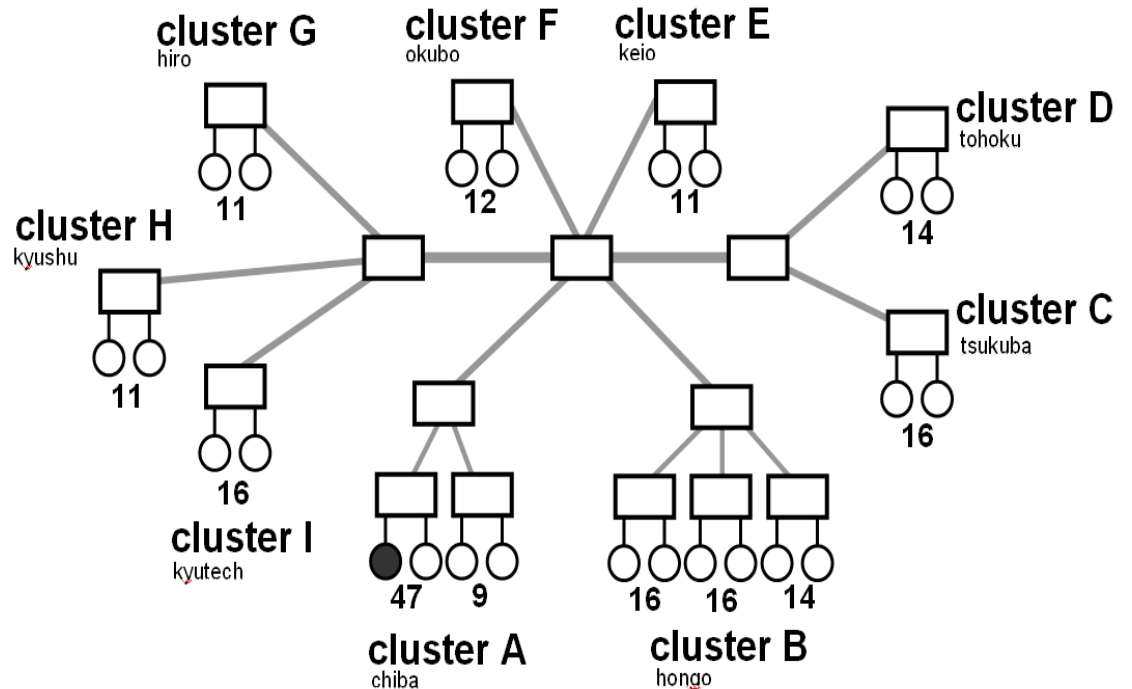


→ 初期のパイプライン  
転送経路

→ 再構築されたパイプライン  
転送経路

# 評価

- 9クラスタ
- バンド幅
  - Cluster Bの16ノードx2のみ500Mbps
  - 残りは1Gbps



# 単一クラスタ内での性能

- Cluster A内でのgather操作
- 実験結果(別紙図8参照)
  - 縦軸: gather実行時間の理論値を1として正規化した総実行時間
- 他の3手法と比べ、OURSは総じて同等以上の性能
- メッセージサイズが大きい場合、Concurrent, MPICHではコンテンションが発生
- メッセージサイズが小さい場合、Sequentialでは同期のコストが無視できない

Concurrent : 一斉通信

Sequential : 逐次同期通信

MPICH: MPICHの利用

OURS : 提案手法

Theoretical: gather実行時間の理論値

# 複数クラスタを利用した場合の性能

- 通信に参加するノードを増減させたときの台数効果（結果は別紙図9を参照）
  - OURSは台数に応じて実行時間が増加
  - Concurrent, MagPleは、実行時間がほぼ一定
    - Cluster A内でのコンテンションが影響
  - Sequentialは、性能が極端に劣化
    - クラスタ間の大きな遅延が影響

Concurrent : 一斉通信

Sequential : 逐次同期通信

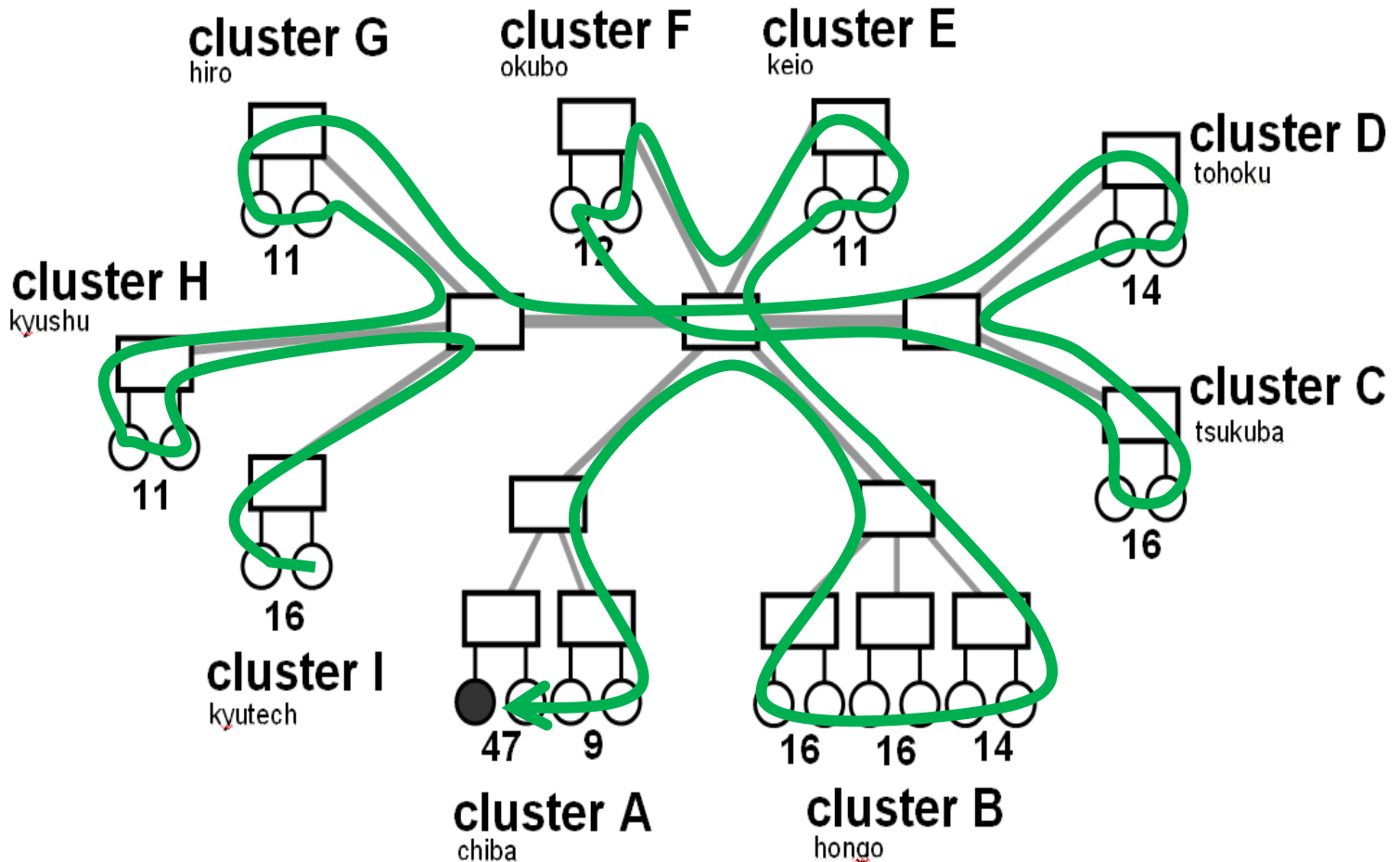
MagPle: LANの内外で異なる

通信パターン

OURS : 提案手法

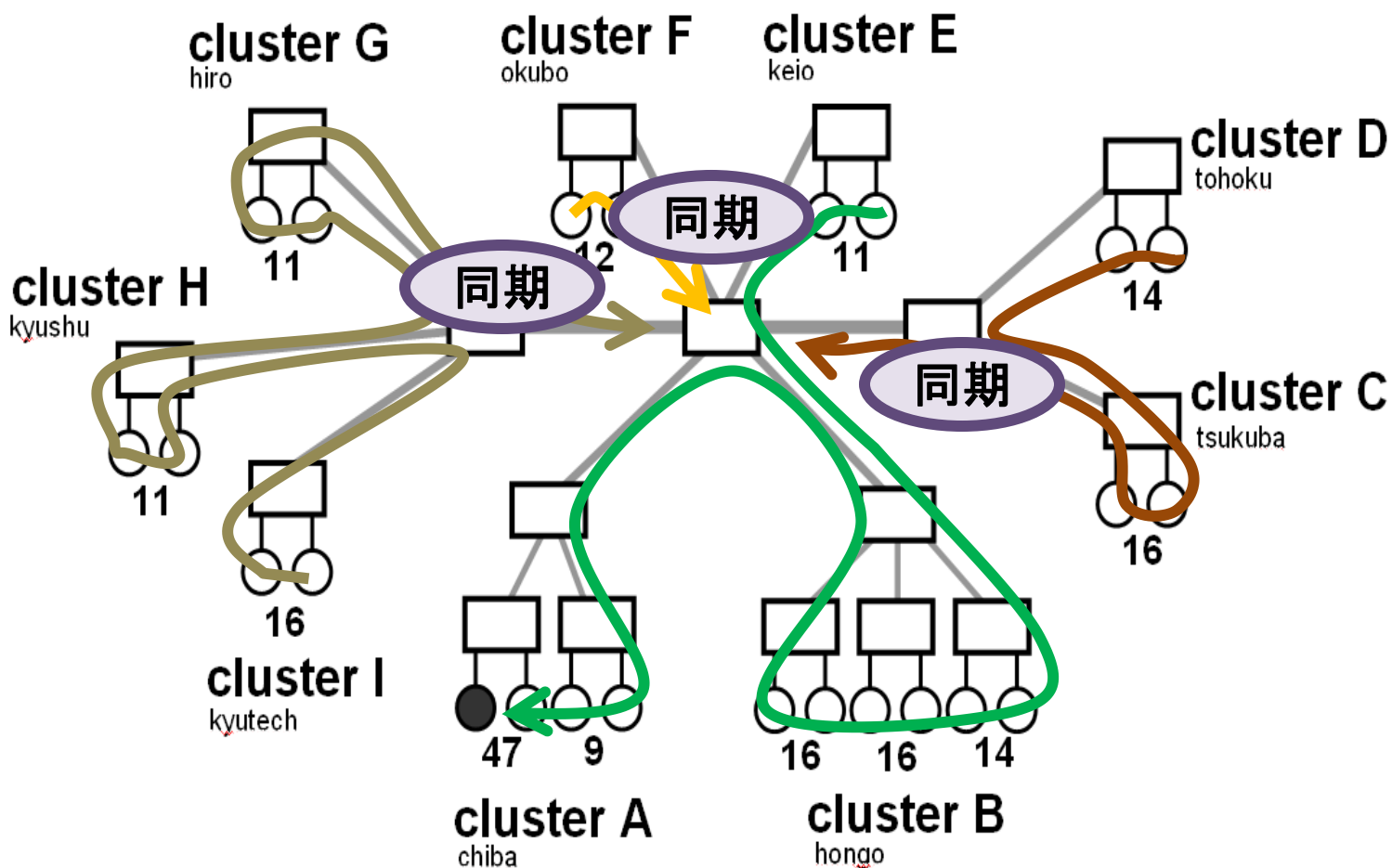
# 評価時に生成された通信ツリー (1/2)

- パイプライン転送の適用後



# 評価時に生成された通信ツリー (2/2)

- タイミング転送の適用後





# 実装

- 現状
  - 擬似的なgather APIを実装
    - 各ノードごとに転送経路を決定
    - 各ノードが適宜send(), recv()を実行
- 今後の予定
  - MPIの実装への組み込み
    - MPI\_Gather()に提案する手法を組み込む
      - 転送経路の決定は、MPI\_Gather()の中で行なう
- 転送経路の決定
  - メッセージサイズのパラメータにより、転送経路が変化するため、gather操作ごとに行なう

HyperShield: 動作中のOSを  
安全な仮想マシン上に移行するための  
仮想マシンモニタ

野元 励 大山 恵弘

セキュリティ基盤

# 概要

- 動的に組み込むことが可能な、  
セキュリティ向上のための仮想マシンモニタ  
HyperShieldの提案
  - 適用するOSに対し、変更や再起動が必要ない
  - OSのカーネルバッファに対する  
バッファオーバーフロー攻撃を防御する機能を実現

# セキュリティ向上のためのVMMを用いた既存手法の問題点

- OSを修正する必要がある
- VMMをあらかじめ起動する必要がある
- VMMによるオーバーヘッドが大きい

# 提案手法

- 動作中のOSに対して組み込みや取り外しが可能なセキュリティ向上のためのVMMを提案
  - VMMのアップデートを、OSを停止させずに行なうことができる
  - 稼働中のサーバマシンに適用できる
  - VMMによるオーバーヘッドの影響を必要最小限にとどめられる

# VMMを用いることによって 実現可能なセキュリティ機能

- OSカーネルによるバッファオーバーフロー攻撃に対する防御
- キーロガー等のルートキットの検出
- I/Oの監視による侵入検知
- カーネルコード等の特定のメモリ領域の書き換え防止
- 重要なレジスタ値やセグメントの書き換え防止

# HyperShieldの防御対象

- OSカーネルによるバッファオーバーフロー攻撃に対する防御
  - カーネルスタック上のリターンアドレス書き換え
  - ユーザ空間にある攻撃コードへの書き換え
    - カーネルスタックに注入された攻撃コードを実行する攻撃は、XDビットを利用したOSの機能で防げる
  - OS機能を拡張することでも実現可能であるが、複雑なOSのコードに修正をする必要がある
    - OSのバージョン等の実装が依存してしまう

# CPU・OSの前提条件

- CPU
  - Intel VT-xに対応している
  - eXecute Disable (XD)ビット機能を有する
  - Physical Address Extension (PAE)に対応している
- OS
  - PAEに対応している

(実装の都合)

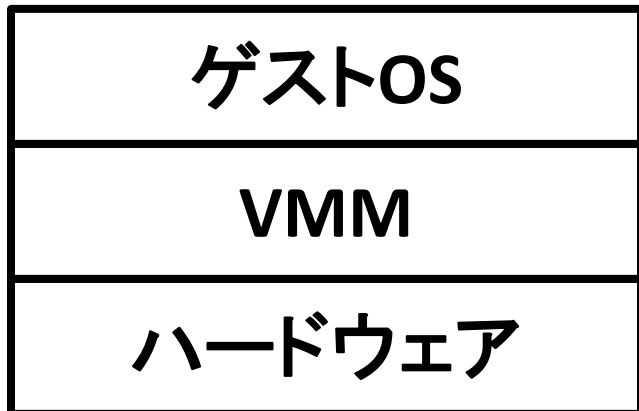
マルチプロセッサには未対応  
同時に稼働させるOSは1つ



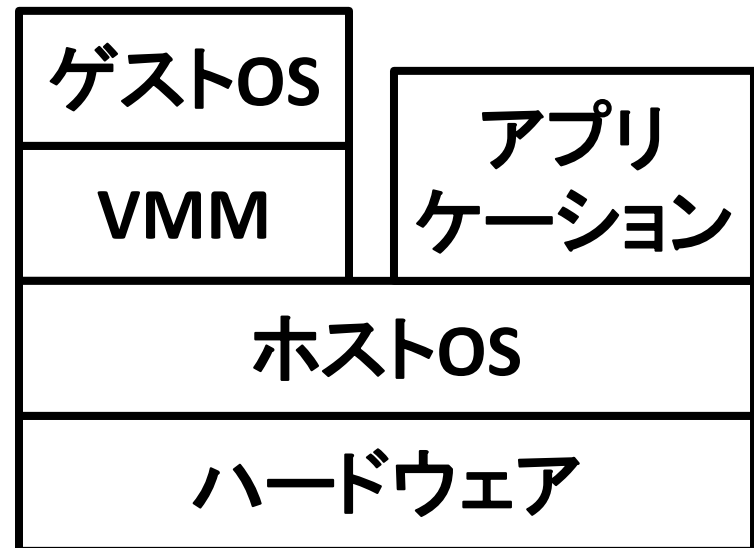
# VMMの選択

- Type-I VMMを利用
  - Type-II VMMではホストOSを保護できない

Type-I VMM



Type-II VMM



# VMMの組み込み・取り外し

- 既存のルートキット技術[BLUE PILL]を利用
- VMMはカーネルモジュールとして実装
  - VMMの組み込み
    - カーネルモジュールの初期化処理
  - VMMの取り外し
    - ゲストOSからの特権命令(VMCALL)の発行

# VMMの組み込み

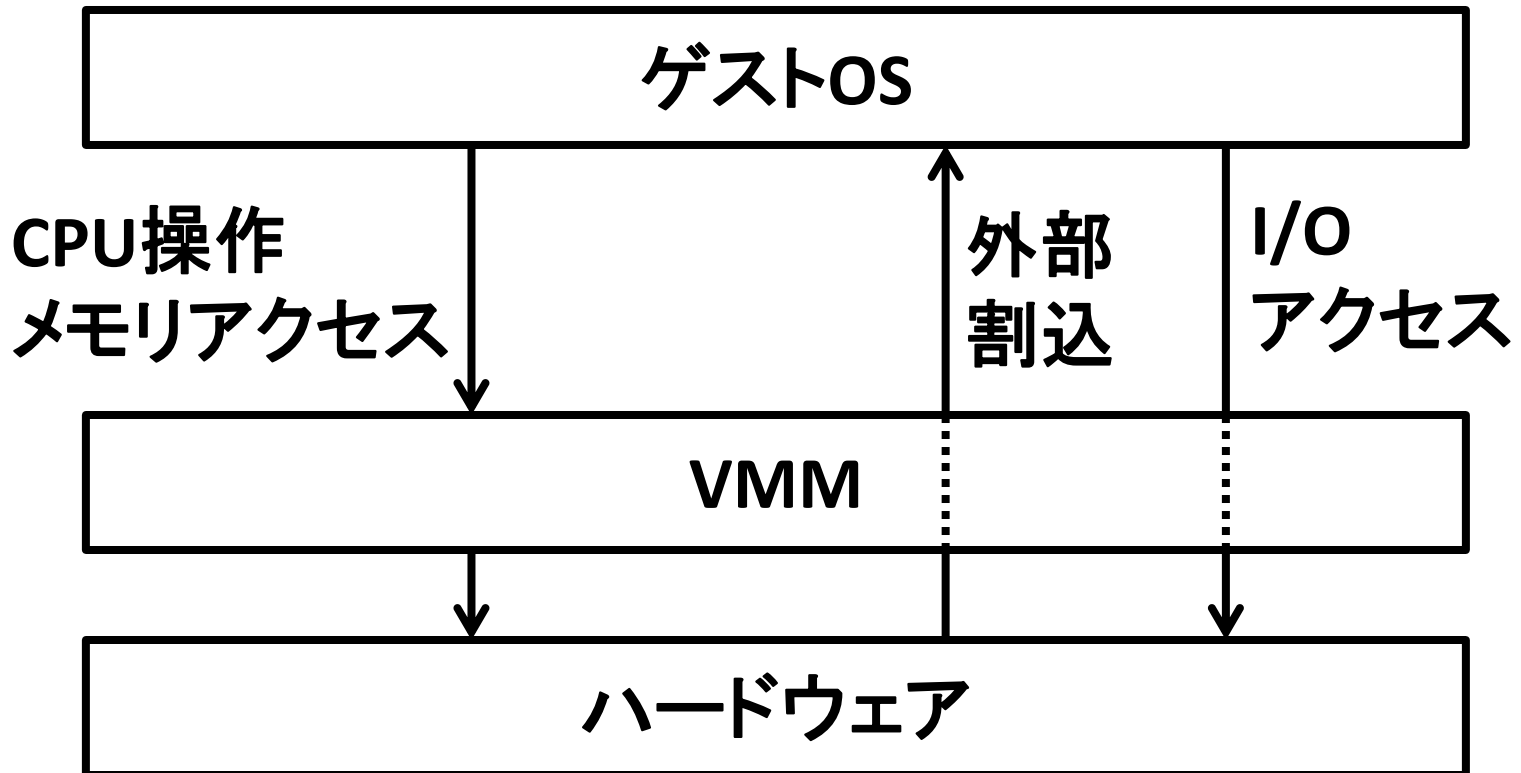
- VMMのスタック領域・メモリ管理用のページテーブル等の確保
- VMのCPU情報の設定
  - VMの起動時のインストラクションポインタには、カーネルモジュールの初期化関数内をアドレスを設定
- VMを起動する命令(VMLAUNCH)を発行

# VMMの取り外し

- VMのCPU情報を実CPU情報に設定
- VMMのスタック領域・メモリ管理用のページテーブル等の解放
- カーネルモジュールの終了処理の実行時に、VMMをアンインストール

# 設計

- CPUとメモリ管理のみを仮想化

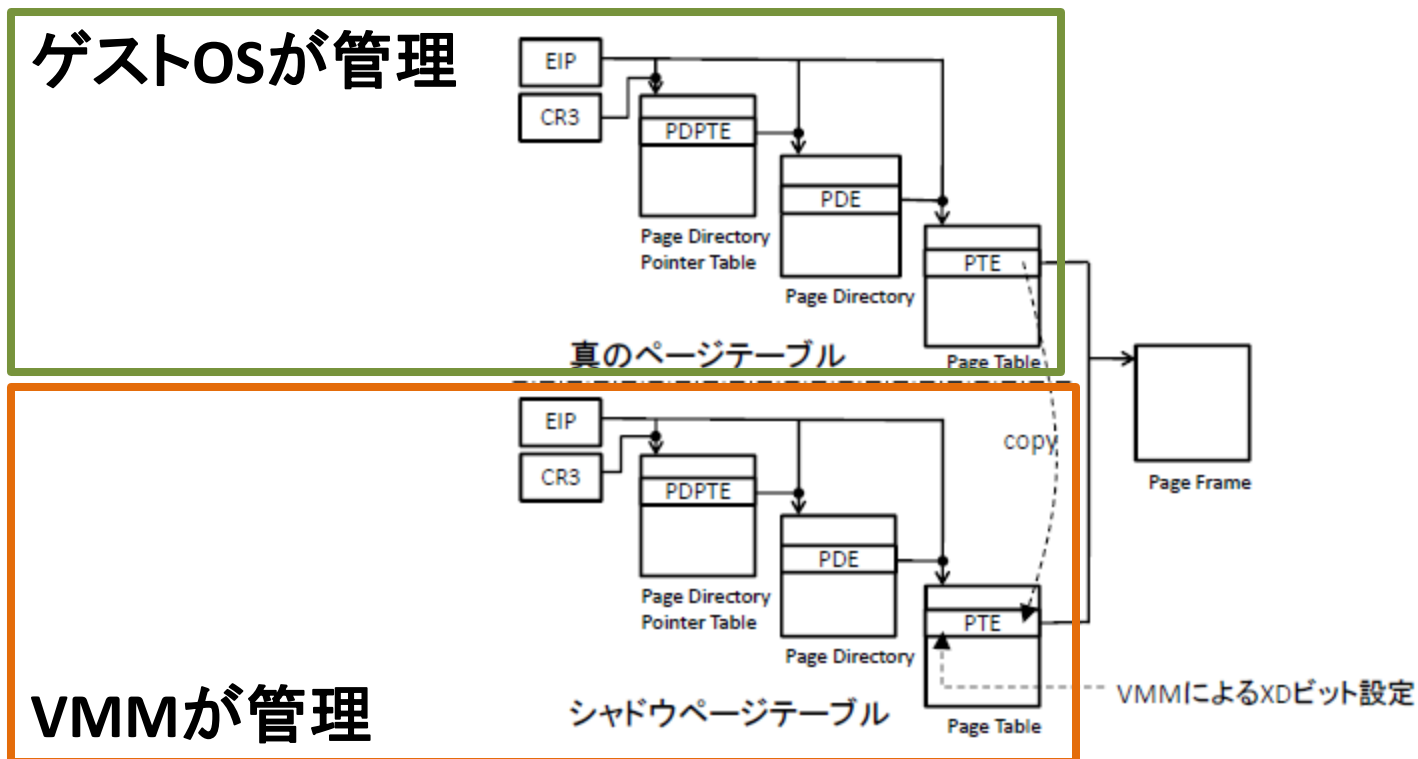


# CPUの仮想化

- ページテーブルやハードウェアに関する一部の命令はVMMがエミュレート
- その他はIntel VT-xの機能を利用
- VMのCPUでは、VMMの組み込み前のスタックやレジスタの値をそのまま利用
  - ページテーブル(CR3)には、VMMが用意したページテーブル(シャドーページテーブル)を設定
  - インストラクションポインタには、カーネルモジュールの初期化関数内をアドレスを設定

# メモリ管理の仮想化

- 実際のメモリアクセス時にはシャドウページテーブルが参照される



# 防御機能の実現

- XDビットの利用
- カーネルモード移行の検知



# XDビットの利用

- ゲストOSカーネルモードに移行後、シャドーページテーブルのユーザ空間領域にXDビットを設定
- ユーザ空間領域の実行時にページフォルトが発生し、動作モードに基づき処理
  - ユーザモード: XDビットをクリアし、実行を継続
  - カーネルモード: バッファオーバーフロー攻撃であるとみなし、ページフォルトのイベントを挿入し、ゲストOSを再開させる

# カーネルモード移行の検知

- カーネルモードへの移行をVMMが検知できるようにするため、必ずページフォルトが発生するように設定する
  - SYSENTER命令による移行
  - 割り込みによる移行
    - Windows: INT 0x2E
    - Linux: INT 0x80

# 評価 (1/2)

- 防御実験
  - バッファオーバーフロー脆弱性を持つデバイスドライバを作成し、攻撃を模擬
- オーバヘッドの計測
  - Unixbench
    - 実マシンと比べ、  
syscallは約25倍、context1は約20倍、  
execlとspawnは約5倍のオーバーヘッドを生じた

execl : execl呼出

syscall : システムコール呼出

spawn : プロセス生成

context1 : コンテキストスイッチ

# 評価

- オーバヘッドの計測

- カーネルコンパイル

- シャドーページテーブル操作、システムコール呼出によりオーバーヘッドが生じる

- PostMark

- 100万個のファイルの生成・読み書き
    - read, writeシステムコールによるVM exitとXDビットの設定によりオーバーヘッドが生じる

表 2 カーネルコンパイルの実行時間 (s)

Table 2 Execution times of kernel build (s)

	実行時間
実マシン	249.56
Xen	257.56
KVM	319.98
HyperShield	396.52

表 3 PostMark の実行時間 (s)

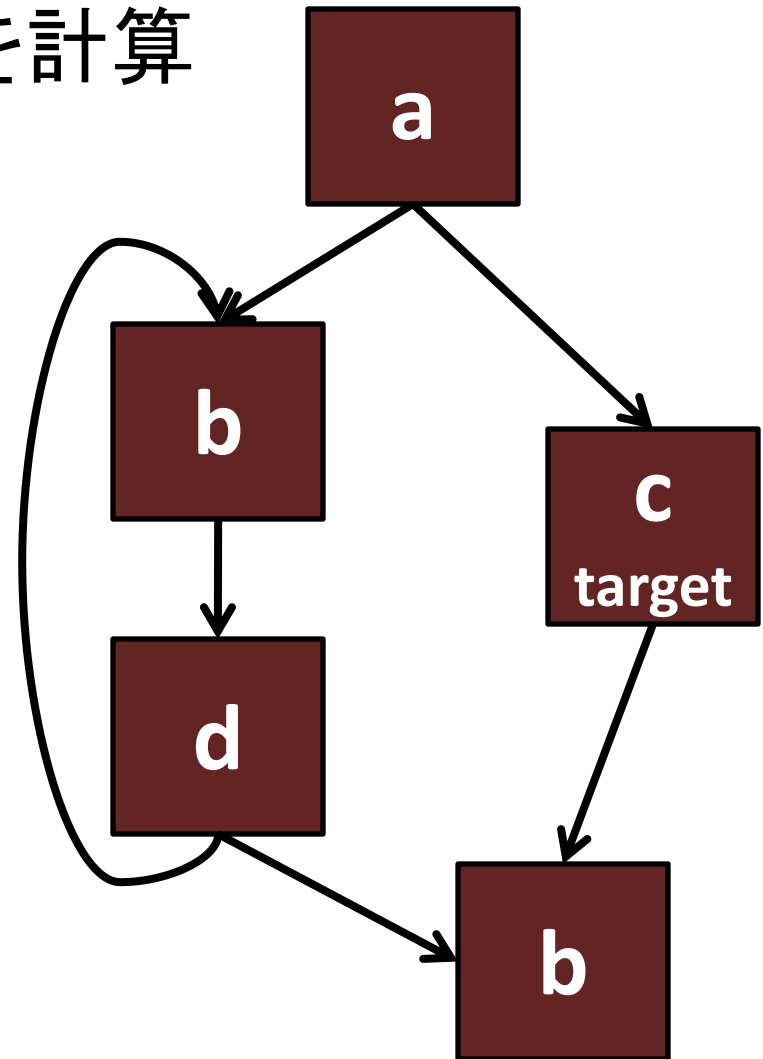
Table 3 Execution times of PostMark (s)

	実行時間
実マシン	72
Xen	98
KVM	124
HyperShield	363

# 付録

# 関数内における解析(1/2)

- targetまでの平均ノード数を計算
  - ノード: コード内の各命令
  - エッジ: 制御の流れ
  - 各命令が1 cycleで終了すると仮定すると、ノード数が命令が実行された直後からのスリープ可能サイクル数の予測値となる



# 関数内における解析(2/2)

- 各ノードの前後での変数の定義
  - 演算器の平均的な使用間隔を表す実数値
- 満たすべき等式群の定義
  - 初期値から、等式群を用いて反復計算
- プログラムの流れと逆方向に解析
  - 実行パスの後続情報を解析するため

# 関数内における解析(2/2)

- 各ノードの前後での変数の定義
  - 演算器の平均的な使用間隔を表す実数値
- 満たすべき等式群の定義
  - 初期値から、等式群を用いて反復計算
- プログラムの流れと逆方向に解析
  - 実行パスの後続情報を解析するため



# 各ノードの前後での変数

- $OUT_D[s]$ 
  - ノード $s$ から次にtargetを使用するノードまでのノード数の期待値
  - ノード $s$ が使用する演算器のスリープ可能時間の期待値を表す
- $IN_D[s]$ 
  - $s$ の直前のノードで定義された $OUT_D[s]$ と同値
- $OUT_P[s]$ 
  - $s$ の後ろの地点からtargetを使用する命令を実行せずに関数の出口に達する確率
- $IN_P[s]$ 
  - $s$ の直前のノードで定義された $OUT_P[s]$ と同値

# 満たすべき等式群 (1/2)

- 命令の伝達定数:  $T_D[s]$ ,  $T_P[s]$

$$T_D[s] = \begin{cases} 0 & (s \text{が} \hat{i} \text{target}) \\ 1 & (\text{その他}) \end{cases} \quad T_P[s] = \begin{cases} 0 & (s \text{が} \hat{i} \text{target}) \\ 1 & (\text{その他}) \end{cases}$$

- 流れの等式群

$$IN_D[s] = T_P[s]OUT_D[s] + T_D[s]$$

$$IN_P[s] = T_P[s]OUT_P[s]$$

$$OUT_D[s] = \begin{cases} q[s]*IN_D[s_{suc1}] + (1-q[s])*IN_D[s_{suc2}] & (s \text{が} \hat{i} \text{branch}) \\ IN_D[s_{suc}] & (\text{その他}) \end{cases}$$

$$OUT_P[s] = \begin{cases} q[s]*IN_P[s_{suc1}] + (1-q[s])*IN_P[s_{suc2}] & (s \text{が} \hat{i} \text{branch}) \\ IN_P[s_{suc}] & (\text{その他}) \end{cases}$$

$s_{suc}$ :  $s$ がbranchでないときの後続命令

$s_{suc1}, s_{suc2}$ :  $s$ がbranchのときの後続命令

$q[s]$ : branchの結果が $s_{suc1}$ の場合の確率

# 満たすべき等式群 (2/2)

- 初期値

$$\begin{array}{ll} \text{IN}_D[s] = T_P[s] * T_D[s] & \text{IN}_P[s] = T_P[s] \\ \text{OUT}_D[s] = 0 & \text{OUT}_P[s] = 0 \end{array}$$

- 関数出口 ( $s_{exit}$ ) での境界値

$$\text{IN}_D[s_{exit}] = 0 \quad \text{IN}_P[s_{exit}] = 1$$

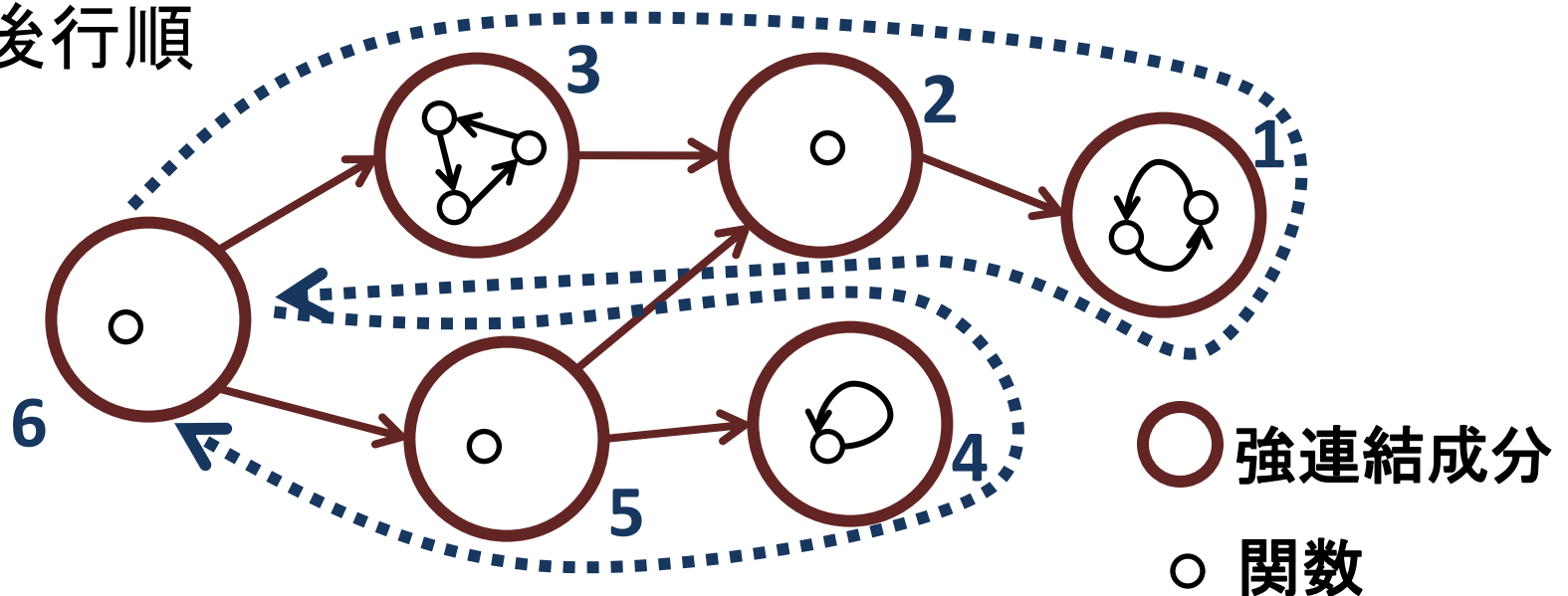
- 前出のCFGの例に、変数、等式群などを適用し、反復計算を行なった例を別紙 (図6) に掲載

# 関数間をまたぐ解析

- 呼び出す関数の先頭にtargetが存在すると仮定し、CFGを利用した解析方法も考えられるが、この方法では精度の良い予測が困難
- Call Graph (CG)を用いた解析
  1. CGの前処理・解析
  2. 各関数の伝達定数の計算
  3. プログラム全体の出口からの解析
    - 2.,3.で各関数を解析する順序が重要となるため、1.で適切な解析順序を決定する

# CGの前処理・解析

- CGの強連結成分分解 (生成されたCG: CG')
  - ループ構造を取り除くため
- CG'の各ノードのラベル付
  - 深さ優先探索
  - 後行順



# 伝達定数の計算

- ラベル順に各関数を解析
- 関数の伝達定数:  $T_D[prc]$ ,  $T_P[prc]$

$$T_D[prc] = IN_D[Sent] \quad T_P[prc] = IN_P[Sent]$$

- 命令の伝達定数:  $T_D[s]$ ,  $T_P[s]$

$$T_D[s] = \begin{cases} 0 & (s \text{が} \hat{i} \text{target}) \\ T_D[prc_{called}] & (s \text{が} \hat{i} \text{jal}) \\ 1 & (\text{その他}) \end{cases}$$

$$T_P[s] = \begin{cases} 0 & (s \text{が} \hat{i} \text{target}) \\ T_P[prc_{called}] & (s \text{が} \hat{i} \text{jal}) \\ 1 & (\text{その他}) \end{cases}$$

*prc*: 関数

*s*: 命令

*Sent*: 関数*prc*の入口の命令

*jal*: 関数呼出命令

*prc<sub>called</sub>*: *jal*が呼び出す関数

# プログラムの出口からの解析

- ラベルの逆順に解析情報を伝達し、プログラムの全ての位置における演算器の使用間隔を計算
  - 境界値  $IN_D[S_{exit}]$  を  $OUT_D[s'_{call}]$  から決定していく

$s'_{call}$  :  $s$ を呼び出す命令