全体ミーティング
(2006/5/23)

M2 佐藤秀明

今回の内容

- 自分の研究について
  - コード複製の高速な解消
- 論文サーベイ
  - コンパイラ支援によるマルチプロセッサの電力消費削減

自分の研究について

Code Clone

- 機能または文面が似ているコード
  - ソースをコピー & ペースト
  - 設計の洗練不足
- 大規模なシステム開発において有害
  - メンテナンス性の低下
- ソースコードの類似性を解析して発見
  - どうやって発見するか？

CCFinder[Kamiya et al.]

- トークン列をマッチング判定
- Parameterized suffix tree[Baker] を採用
  - 同じ変数の出現を直前の出現位置からの距離に置換
    - 例 :xbyyxbx ⇒ 0b014b2
  - 変数名の違いを吸収したマッチング

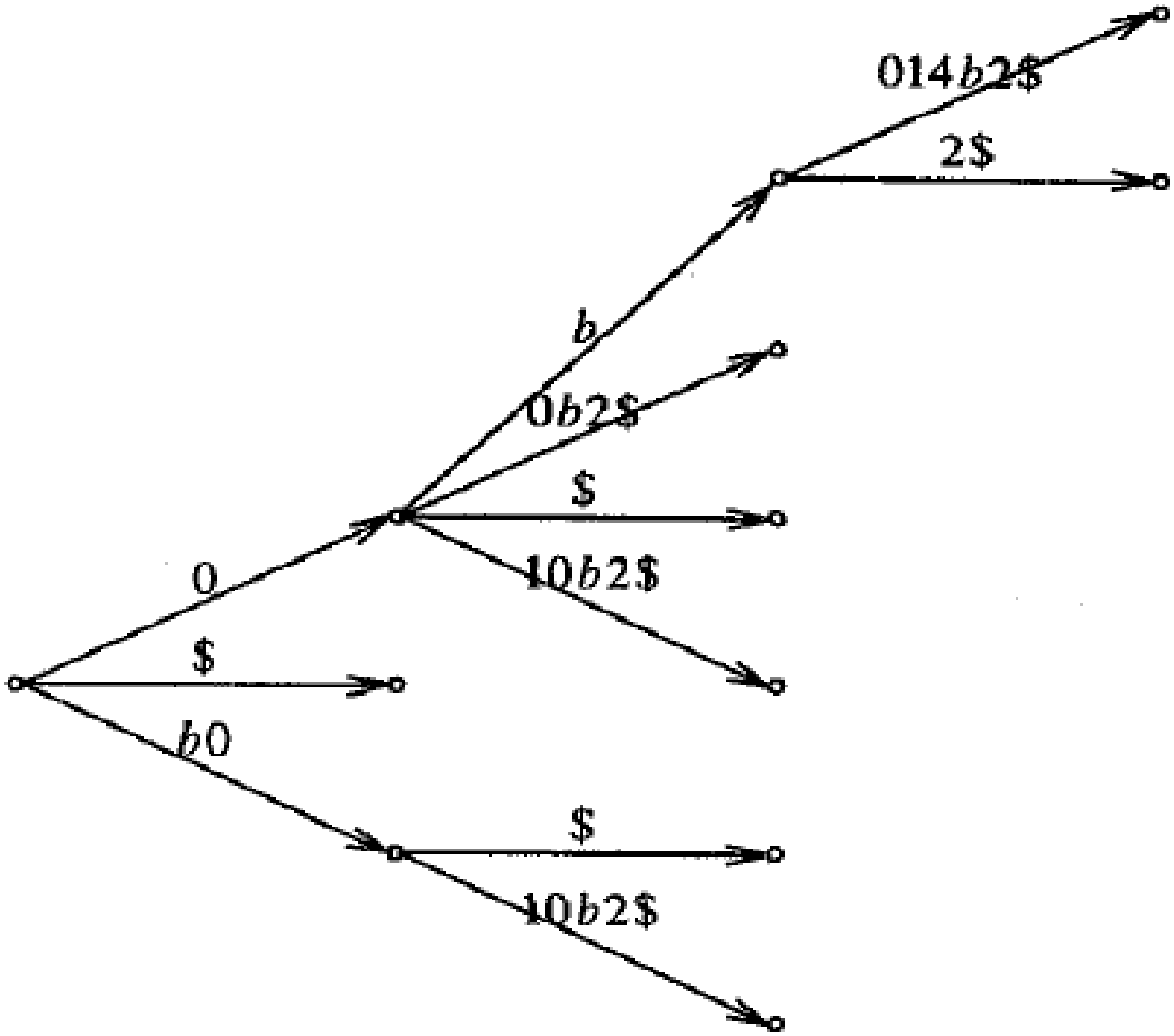


Figure 5: A p-suffix tree for the p-string $S = xbyyxbx\$$.

問題

- 一度に提示するクローンの数が多すぎる
  - 膨大な情報に溺れる
- クローンをどう除去するかについてはノータッチ
  - 除去まで考えると解析時間がかかりすぎる

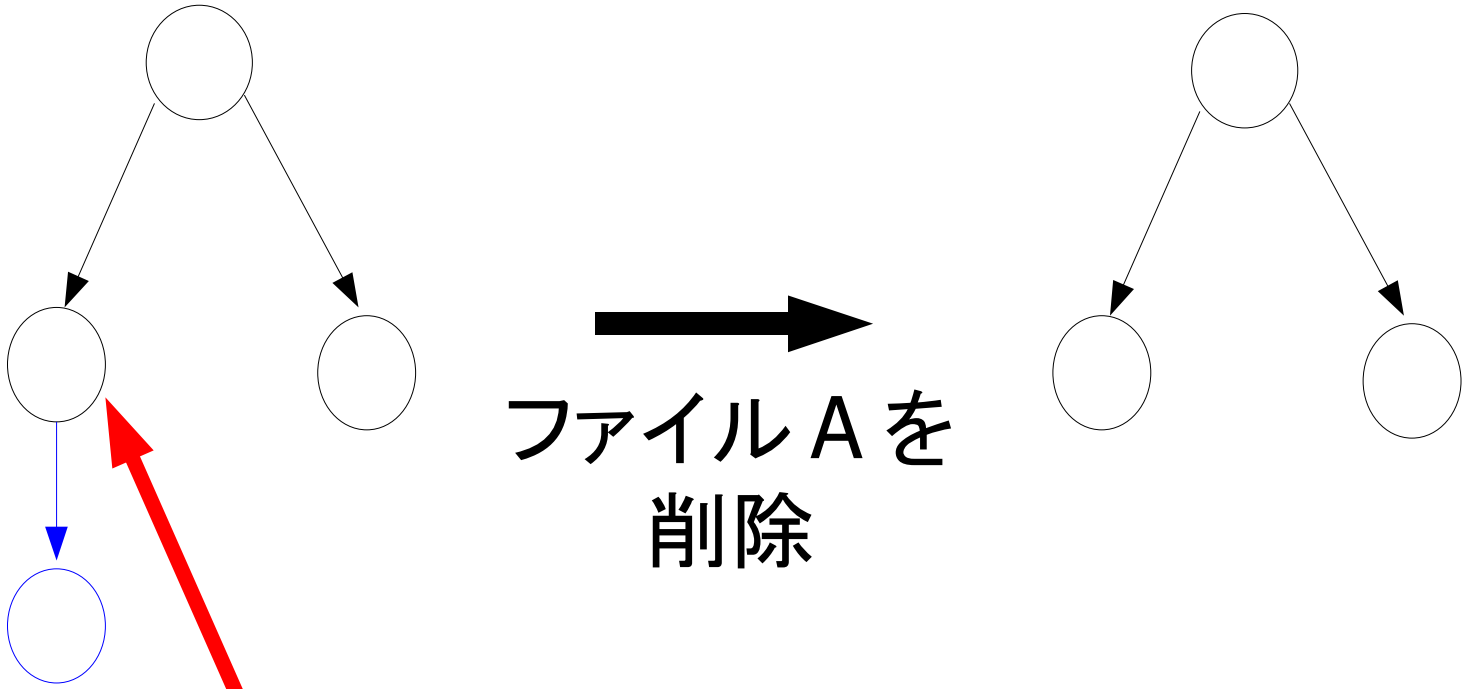
基本的なアイデア

- 毎日少しずつリファクタリング
  - 昨日からの差分のみを解析対象にする
  - コードを常にきれいに保つ
- 2 つのステップでクローン候補を段階的に選抜
  - 重い解析を行う対象をできるだけ減らす
- クローン除去を視覚的に支援
  - 瑣末な書き換えミスを防止
  - 「面倒」というイメージを改善

Updatable Suffix Tree

- 毎回 suffix tree を一から作り直すのは無駄
  - 昨日からの差分のみを更新すれば済むはず
- ツリーの内容を後から改変できるしくみを導入
  - 不要になった枝はカット
- ツリー構造は普段はファイルに書き出しておく
  - 解析時に読み込み

Tree 更新の例



ファイル A しか使わない
枝の先頭を保持

バージョン管理システムとの連携

- リビジョンを上げるときにクローン解析を行う
  - プログラマにとって妥当なタイミングであろう
- バージョン管理システムから必要な情報を取得
  - 更新されたファイルの一覧

現状

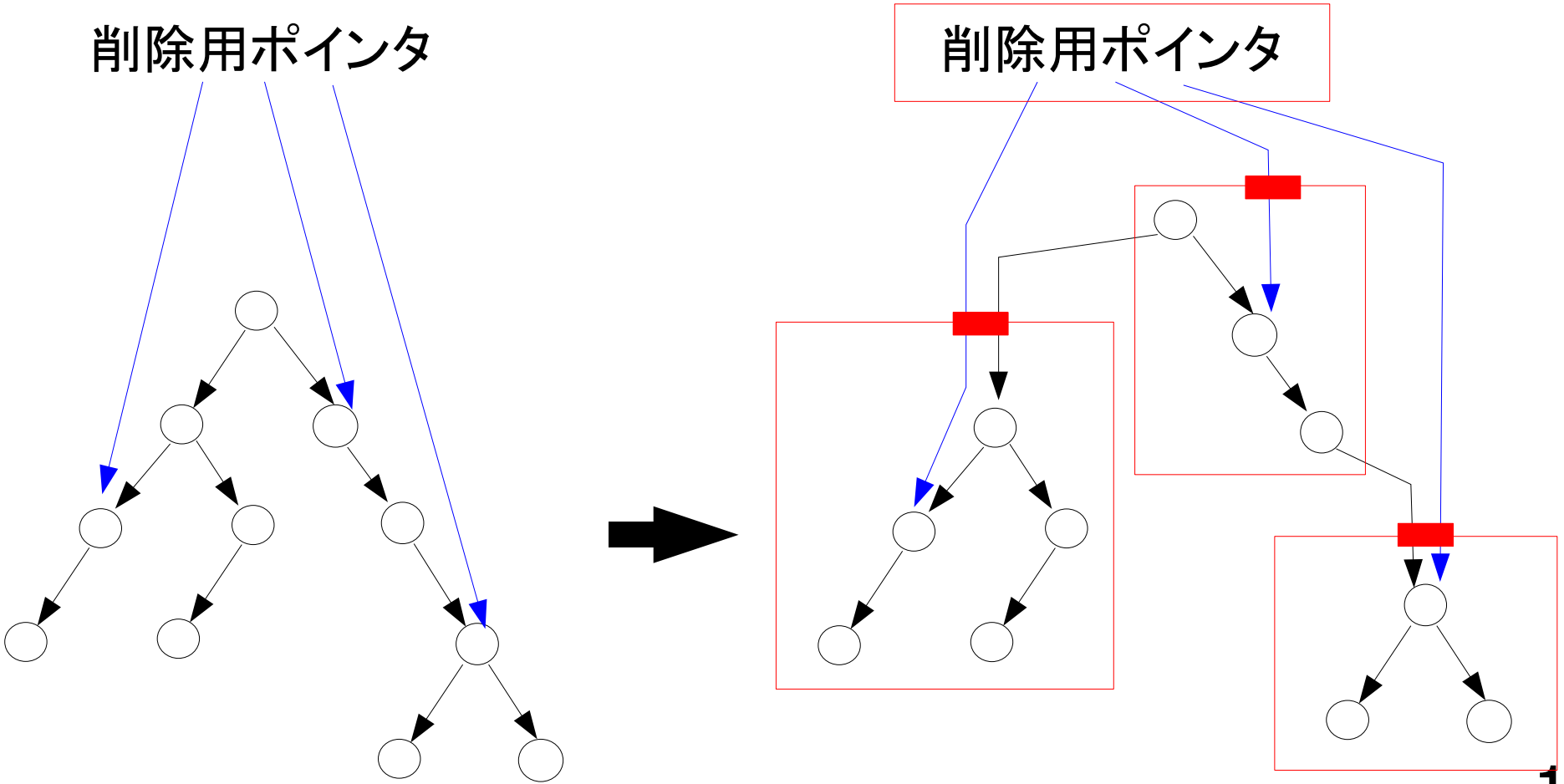
- ここまでの内容をプロトタイプとして実装
  - バージョン管理システムの活用
  - 字句解析
  - ツリー操作
  - クローン候補発見
- さらなる最適化を予定

ツリーの分割管理

- 毎回全てのツリー構造を読み込むのは無駄
  - 改変されるのはツリーのほんの一部分
- 部分ツリーごとに分割して保存すれば効率的
  - 改変のなかった部分は読み込まれない
- 最適な分割方法は考え中
  - ツリーへのアクセスに局所性を見出すことができれば…

ツリーの分割例

- DAG 構造を適切に処理する必要がある
  - 削除用ポインタの存在



構文解析

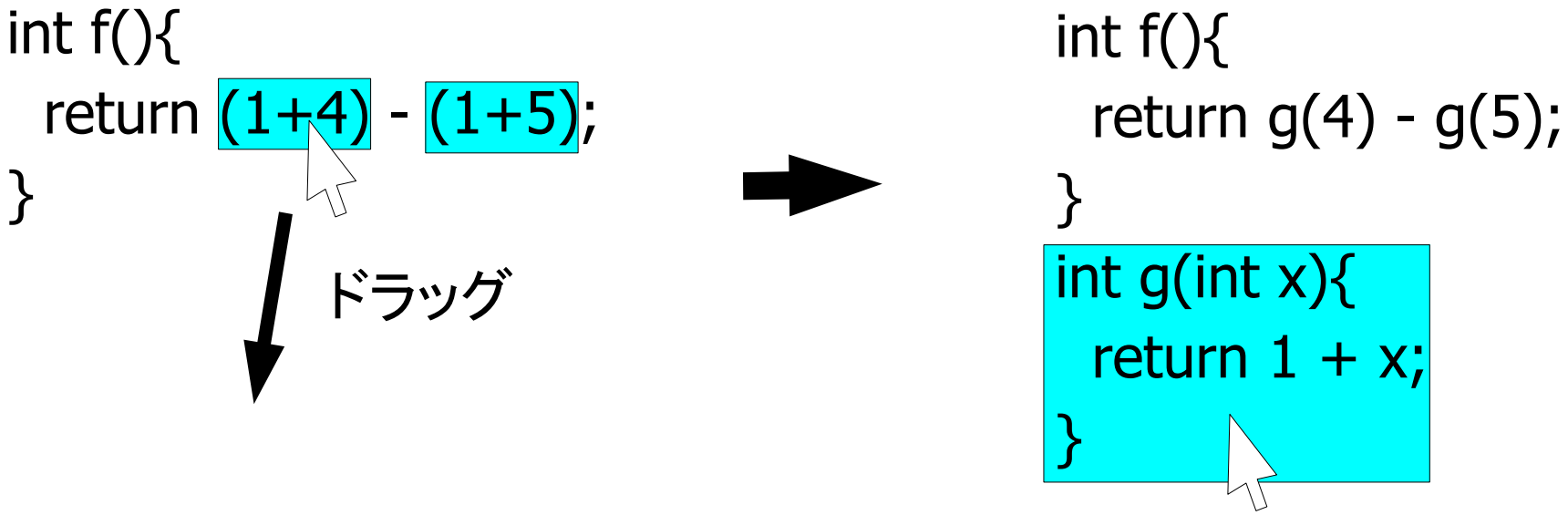
- suffix tree で絞られたクローン候補に適用
  - 全体の計算時間は小さいはず
- プログラムの変換に必要な情報を取得
  - パーズ
  - 型チェック
  - 生存期間解析
  - 自由変数解析

コード書き換え

- 実際に書き換えるか否かの判断は人間に任せる
  - プログラムの構造は設計者のポリシーに依存
- 全自動ではなく人間自身に書き換えさせるべき
  - 改変を加えた場所を覚えてもらうため
- 書き換えが簡単かつ面白い作業ならなおよい
  - リファクタリングしたくない人を説得

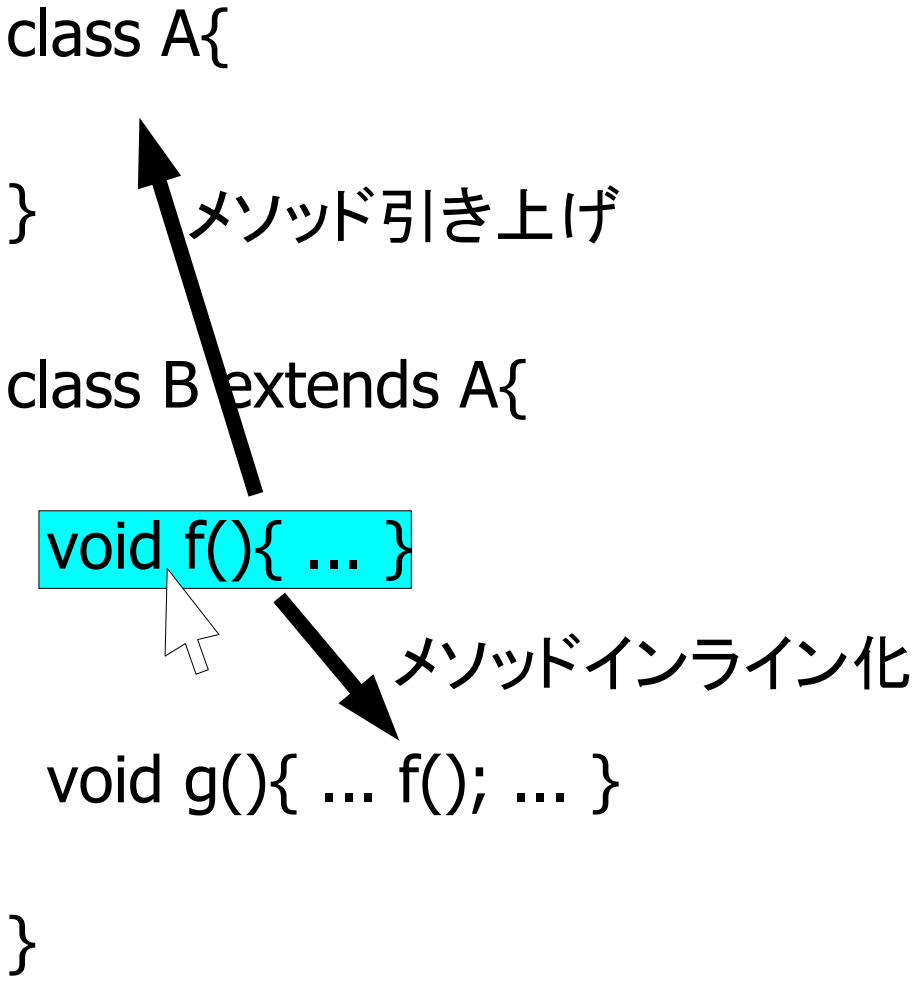
視覚的なコード書き換え

- クローンの抽出先をドラッグ＆ドロップで指示
  - どこへ抽出するかによって抽出手順を柔軟に調整
  - プログラムの意味を変えずに正しく変換



発展：視覚的リファクタリング (1)

- クローン抽出以外の手法もドラッグ＆ドロップで
  - どの手法を適用するかを自動的に判断



発展：視覚的リファクタリング (2)

- ドラッグ＆ドロップの表現力は妥当か？
  - 複数の手法が候補として挙がる可能性
  - 複雑すぎてドラッグ＆ドロップでは表現できない手法
- プログラマの範囲指定が不正確な場合は？
  - 構文木にならない範囲を指定してきたら
  - 融通を利かせてうまくパーズできるか

まとめ

- コード複製の高速な除去
  - 更新可能な suffix tree の導入
  - クローン候補の 2 段階選抜
  - ドラッグ＆ドロップによる抽出先の指定

論文のサーベイ

題材

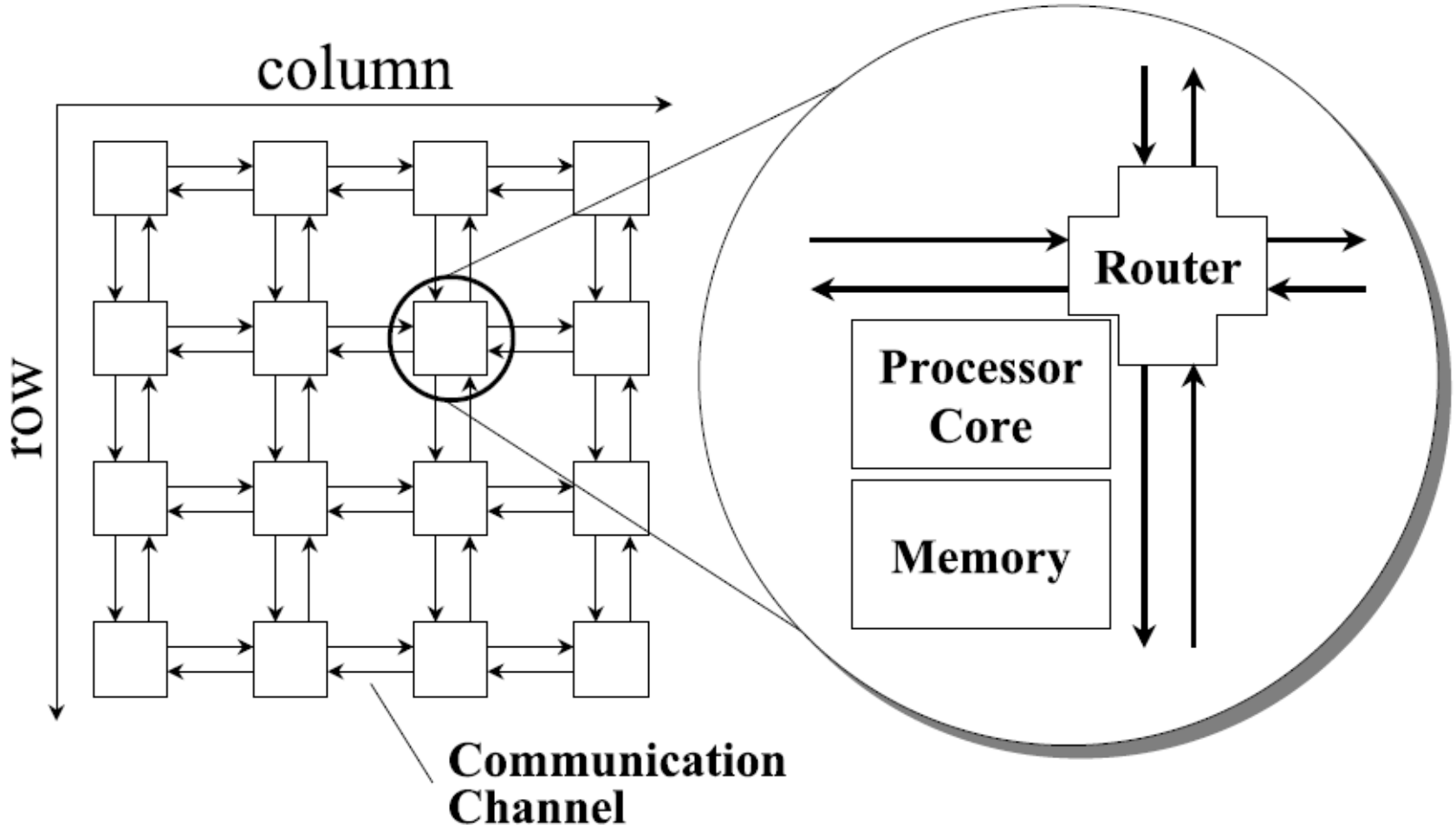
- Compiler-Directed Channel Allocation for Saving Power in On-Chip Networks
  - By Guangyu Chen, Feihui Li and Mahmut Kandemir
  - In Proc. ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL'06), January 2006
- 組み込みマルチプロセッサの電力消費削減
  - プロセッサ間の通信に必要なチャネル数を削減

組み込みシステムの最新動向

- 超マルチプロセッサ時代がやってくる
  - 16 プロセッサ、32 プロセッサ、…
- プロセッサ間の接続トポロジを考慮する必要性
  - Network-on-Chip (NoC) アーキテクチャの概念

接続トポロジの一例：メッシュ型

- 画像 / 動画処理アルゴリズムとの親和性
- 隣接プロセッサ同士を 1 対の通信チャネルで接続



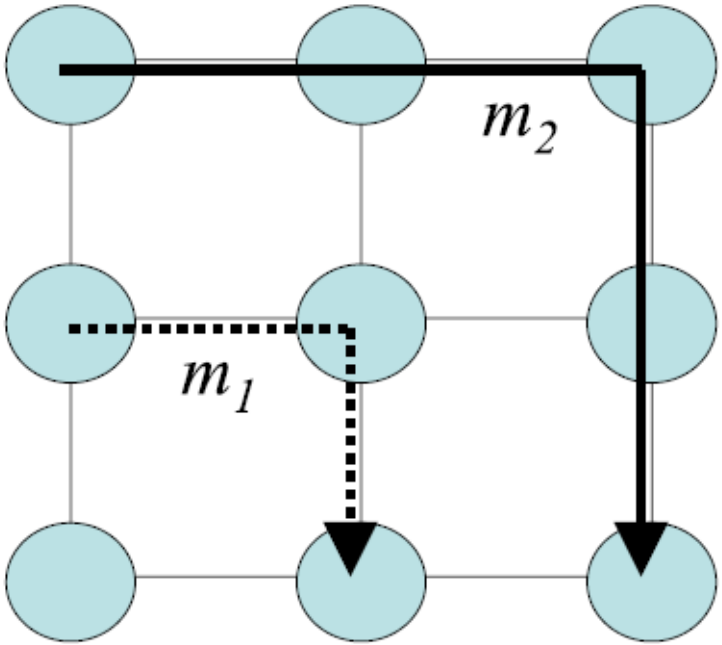
消費電力への対処

- 問題：プロセッサ数が増えると消費電力も増える
  - 組み込みシステムにとっては致命的
- 有効な策：使っていない通信チャネルは電源を切る
  - 効果は既存研究によって実証済み

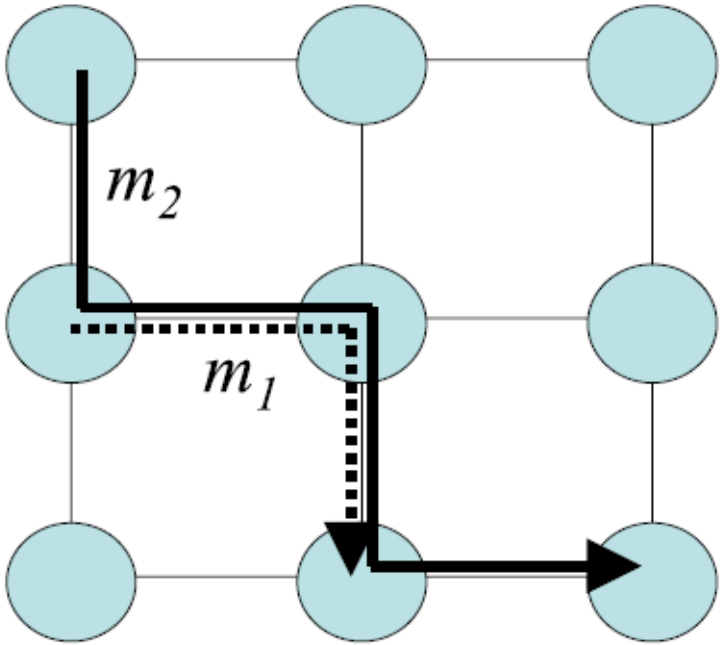
- では、その電源を切る機会はどうすれば増えるか？
  - 使用するチャネル数を少なく抑える
  - 各チャネルを使用しない期間をできる限り長くする

我々の提案する手法

- 複数の connection 間で通信チャネルを共有
  - ただし各 connection が互いに衝突してはいけない



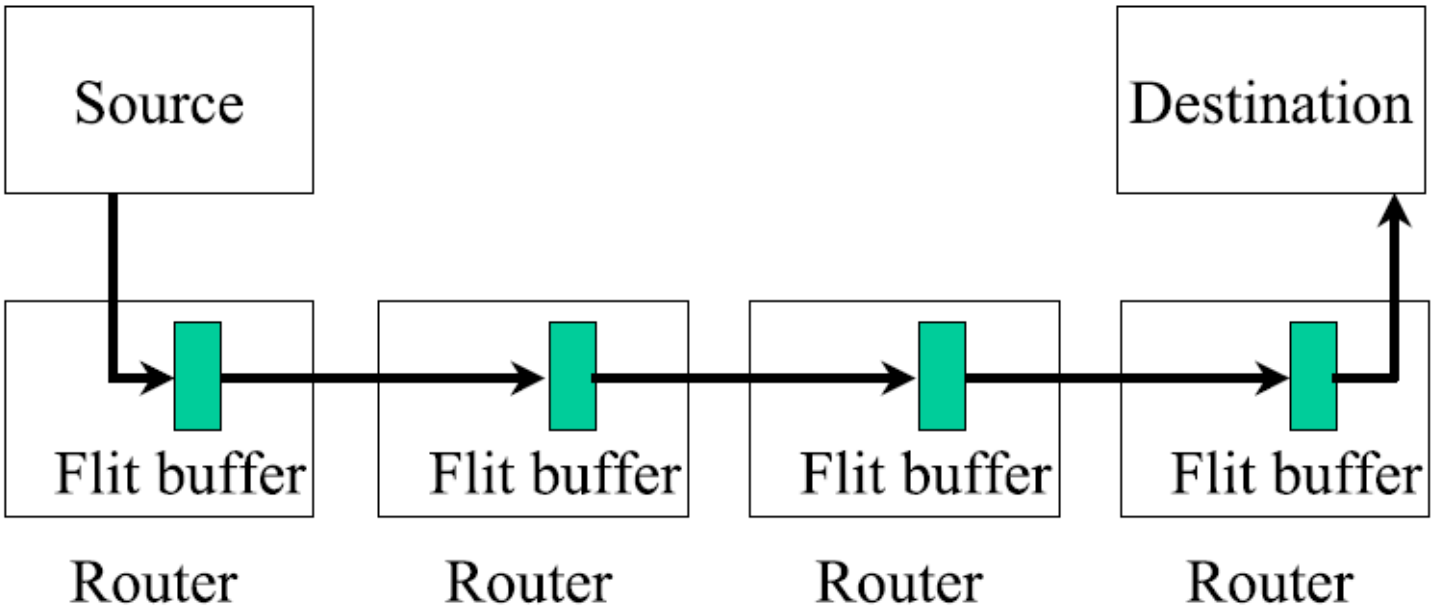
X-Y 法
(X 方向、Y 方向の順に進む)



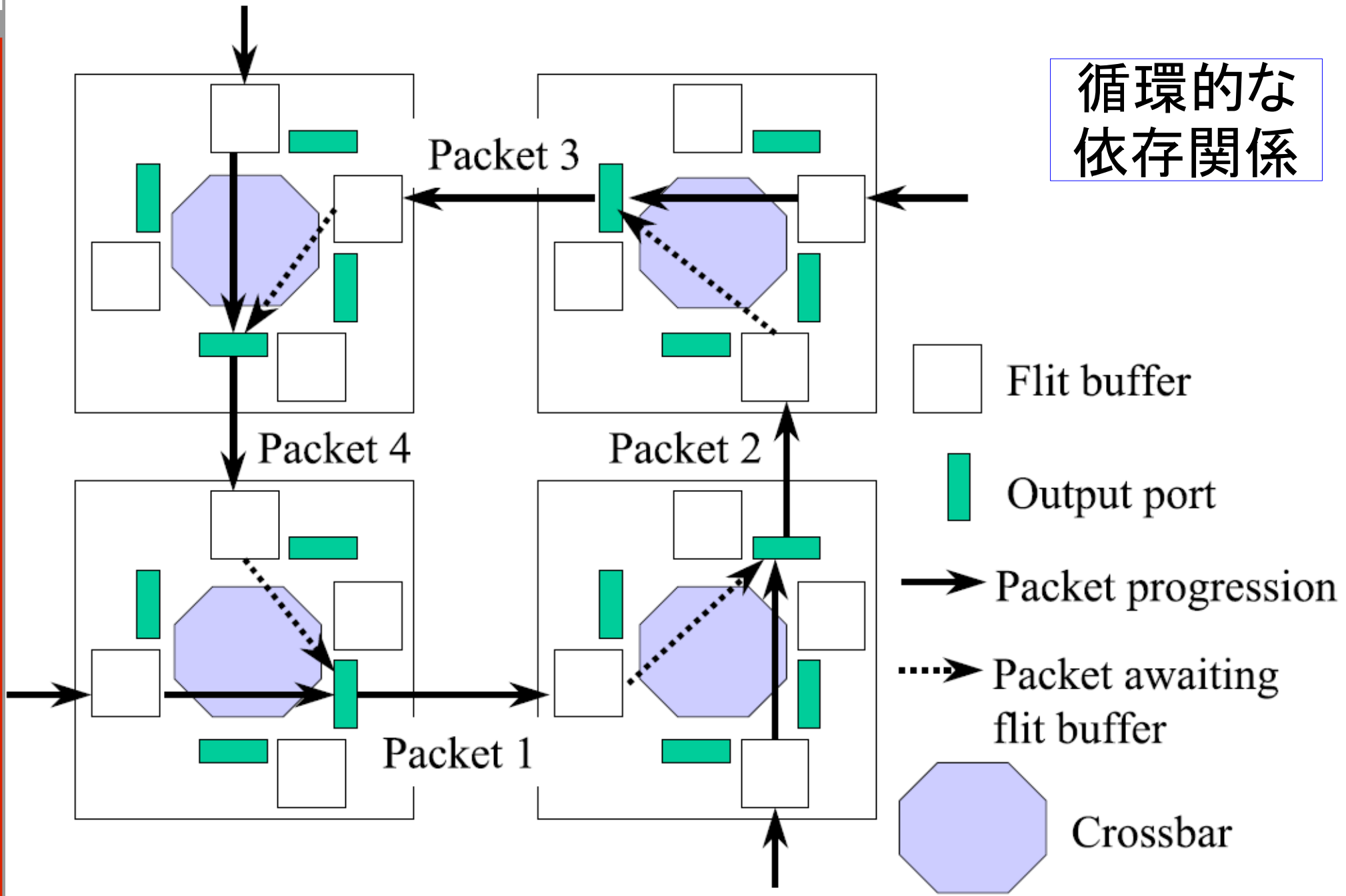
我々の手法
(チャネルをできるだけ共有)

前提知識 :Wormhole Switching

- 1 つの packet は複数の flits に分割して順に送る
  - ムカデの行進
- 次に進む flit buffer が空くまで進行が block される
  - deadlock の発生する危険性



Deadlock の例

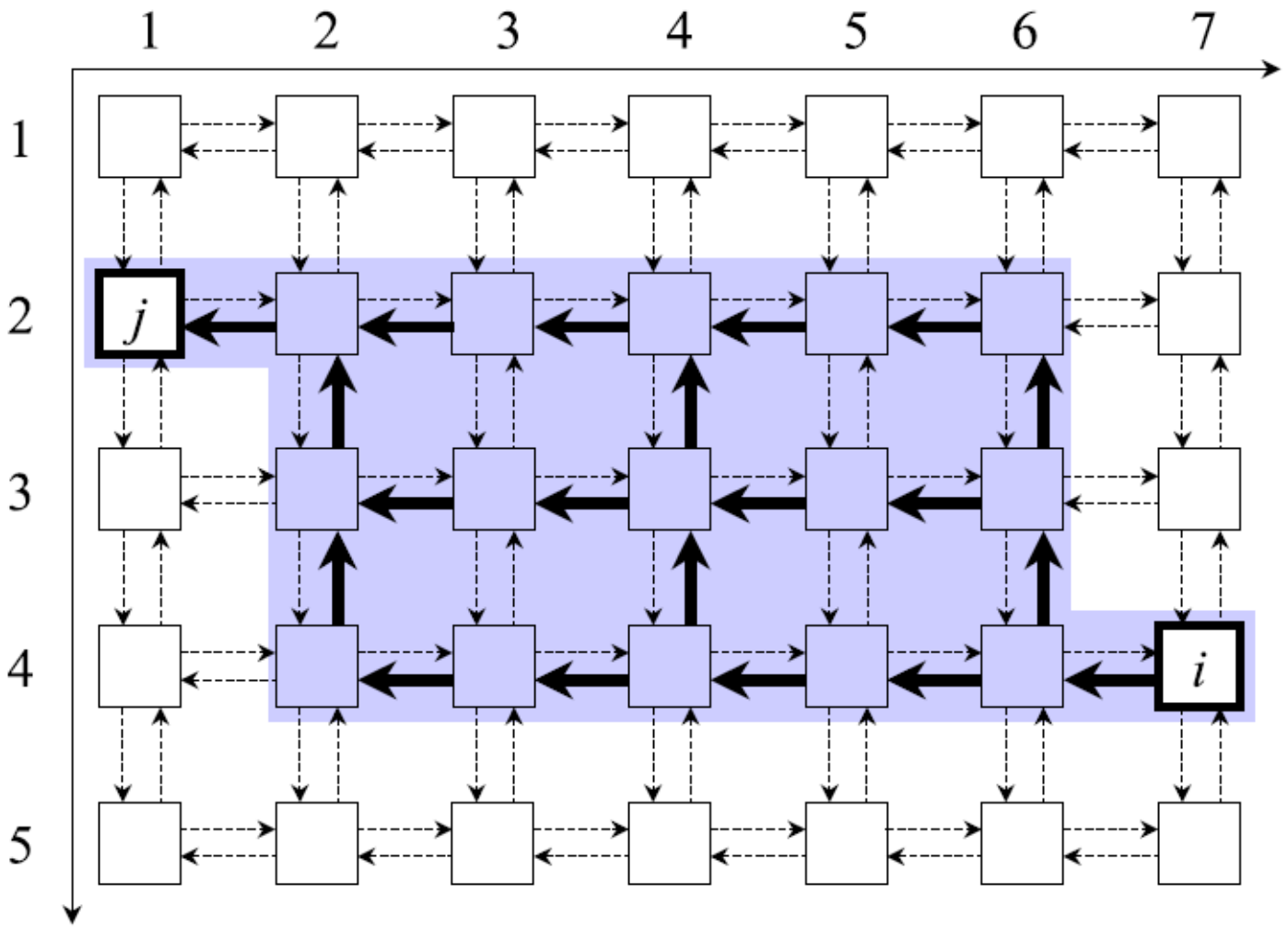

循環的な
依存関係
Packet 3
Packet 4
Packet 2
Packet 1
Flit buffer
Output port
Packet progression
Packet awaiting
flit buffer
Crossbar

前提知識 :Deadlock 防止アルゴリズム

- (1)packet は shortest path で routing する
- (2) 東方向への packet 送信では偶数列の南北方向チャネルを使用しない
- (3) 西方向への packet 送信では奇数列の南北方向チャネルを使用しない

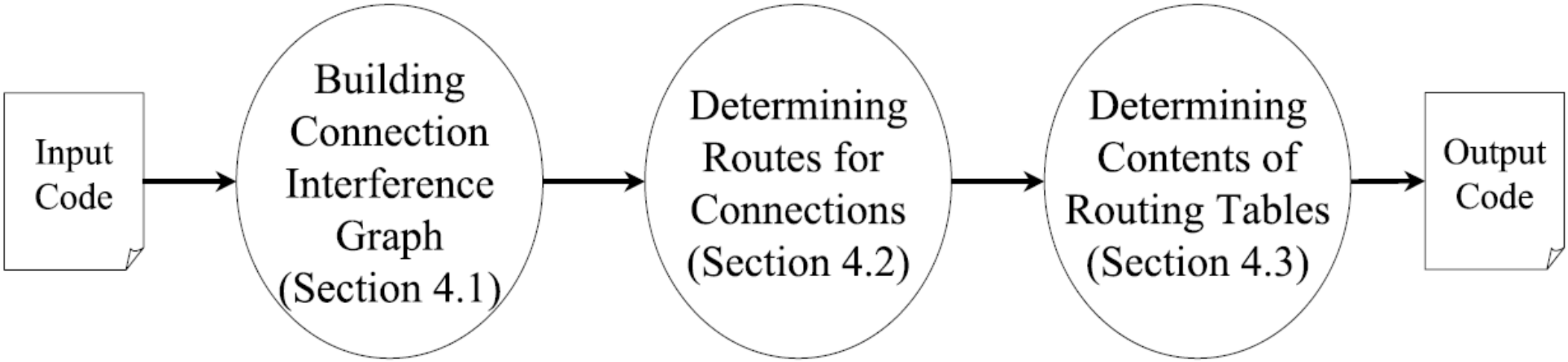
Deadlock 防止アルゴリズムの適用例

- 1/3/5/7 列目の南北方向チャネルは使用できない



我々のアプローチ

- (1) 各 connection 間の干渉グラフを作成
  - conflict する可能性を列挙
- (2) (1) を参考に各 connection を routing
  - できるだけ channel を共有させる
- (3) (2) を基に各 router の routing table を作成

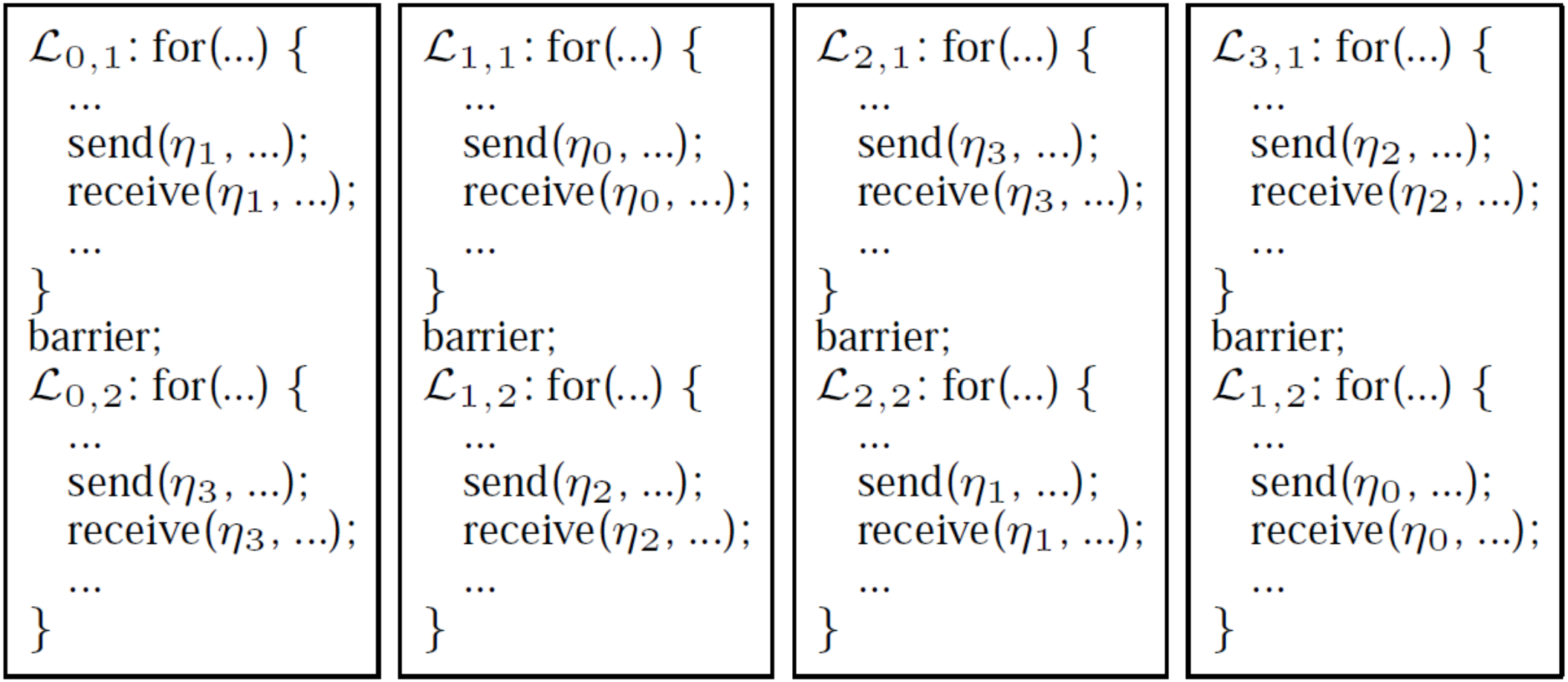


(1) 干渉グラフの作成

- 実行コード中の「互いに並列なループ」を発見
  1. 一方から他方への message passing がある、
  2. 同じ同期ポイントの直後に実行される、または
  3. 推移的に「互いに並列」を導ける
- 各 connection をノードとする干渉グラフを構成
  - 2 つのノード間にエッジが存在
    = 各送信元が互いに並列なループの一方ずつを持つ

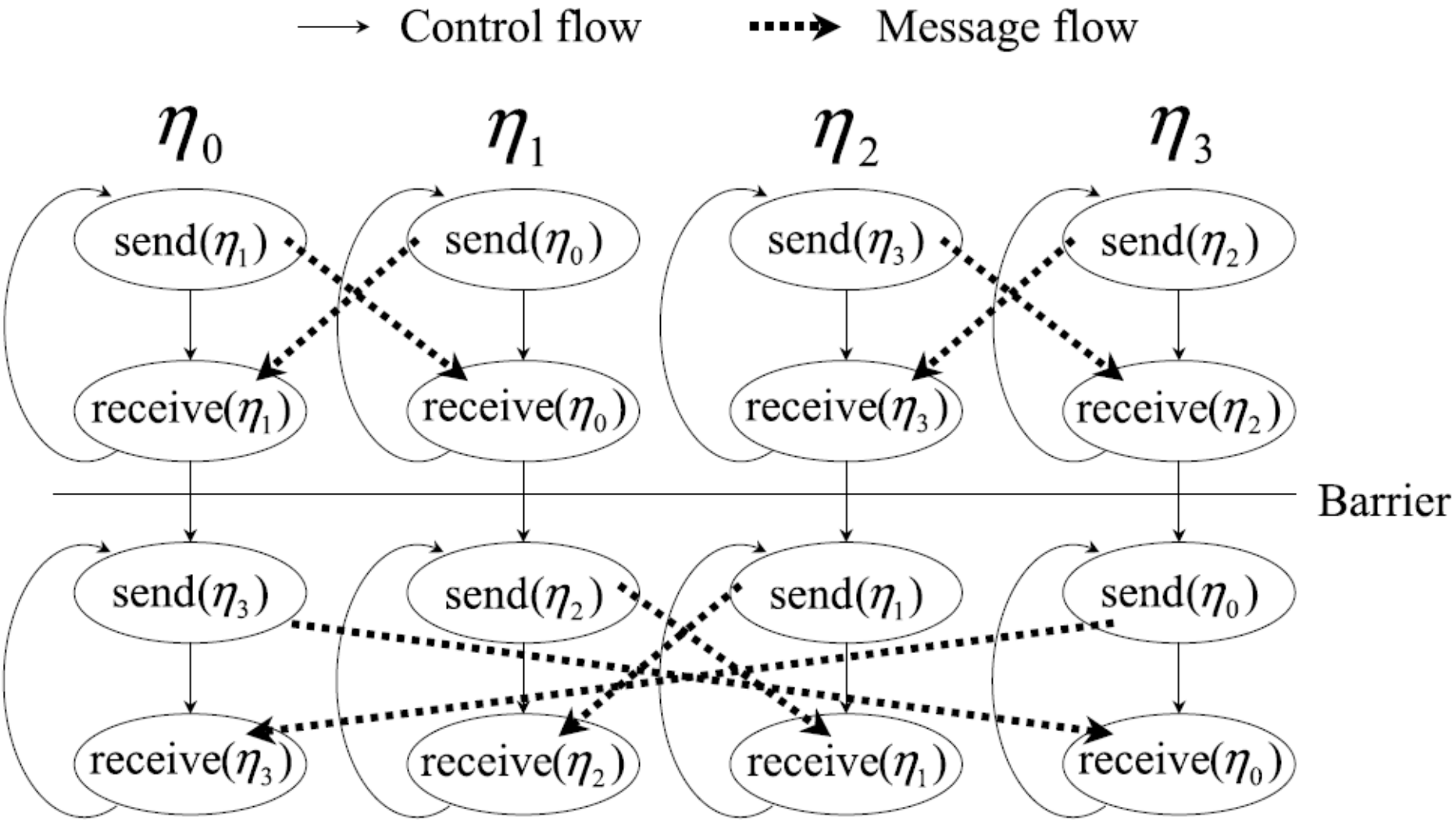
干渉グラフの例 (1)

- 各プロセッサが実行するコード



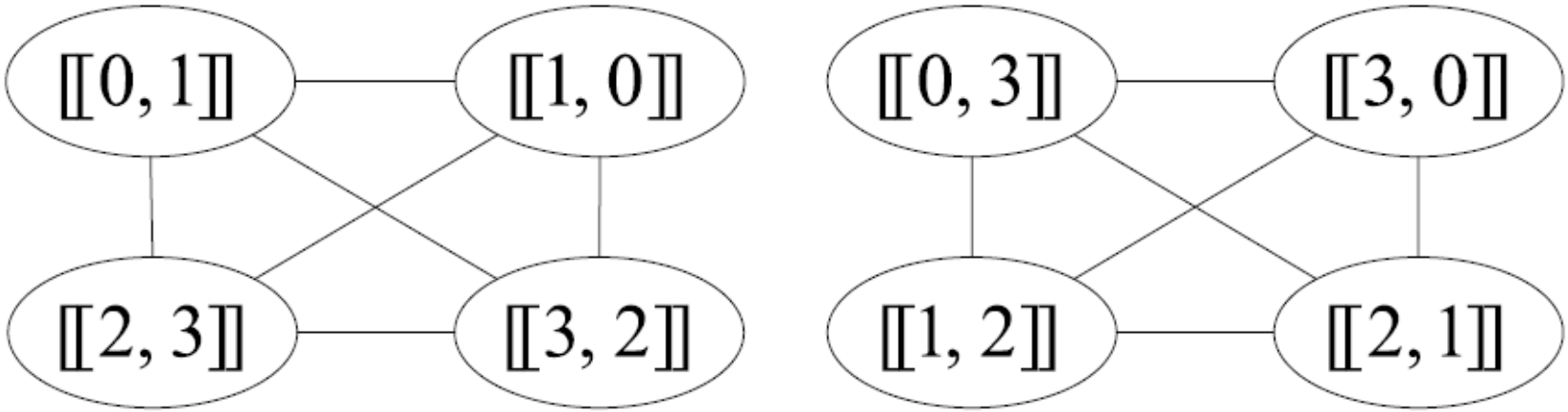
干渉グラフの例 (2)

- 制御フローとメッセージフロー



干渉グラフの例 (3)

- connection に関する干渉グラフ

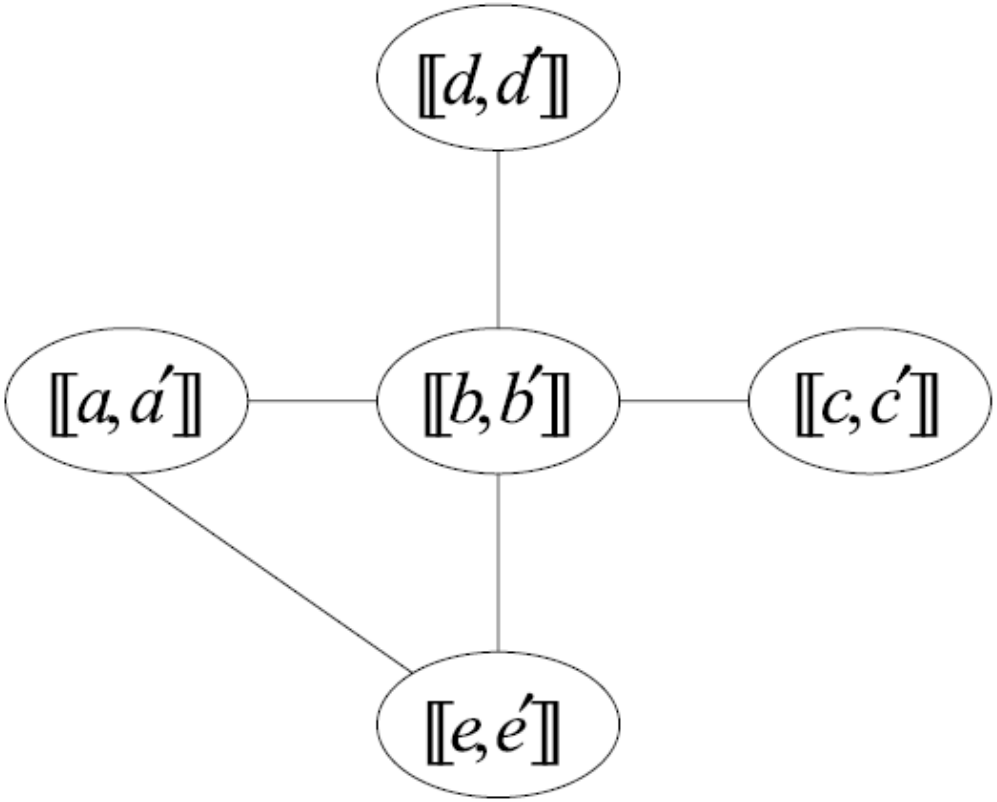


(2) 各 connection の routing

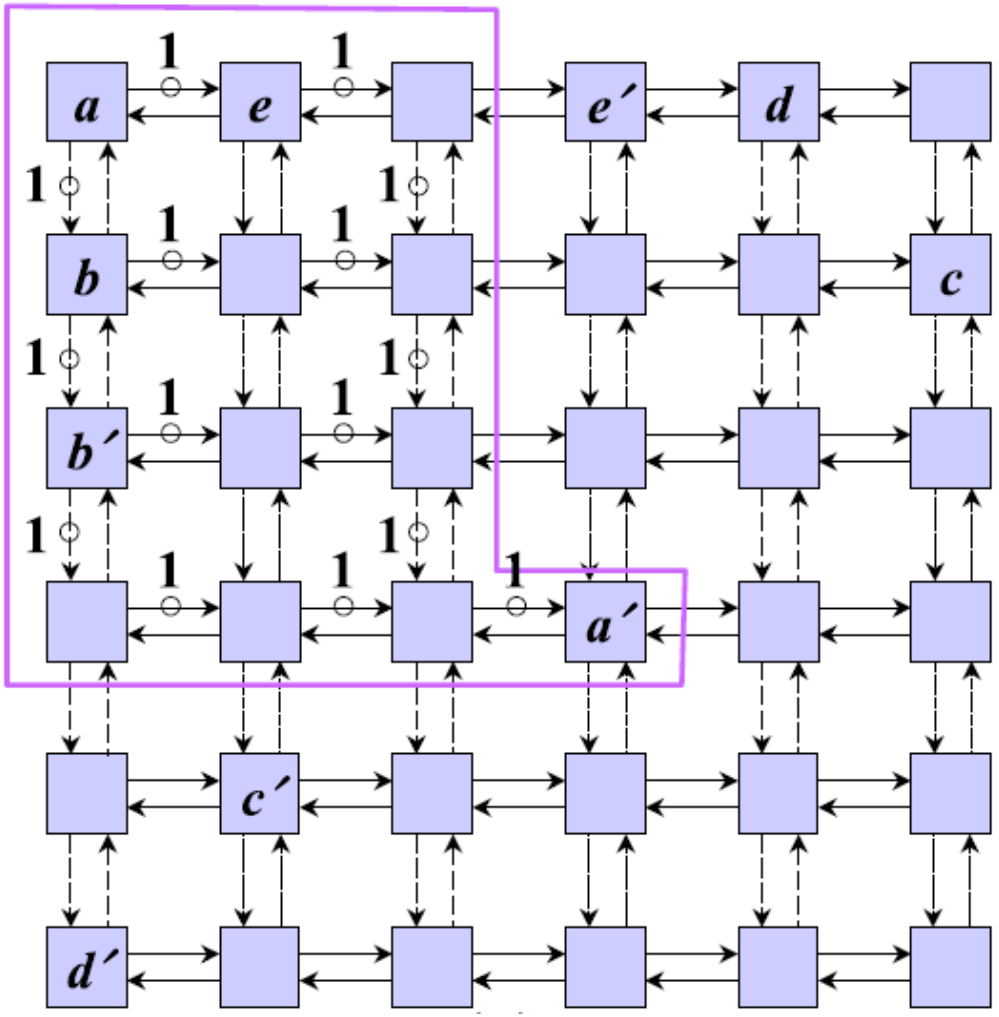
- 干渉グラフを真面目に解くと NP-hard
- heuristic に解く

1. 各通信チャネルにカウンター設置
2. 各 connection が通り得るチャネルのカウンタに
   それぞれ 1 を加算
3. routing の選択肢が最少の connection を 1 つ選ぶ
   - 最多のものから選んでいくとチャネルを共有させ辛くなる
4. 通るチャネルのカウンタ合計が最大になるように、
   かつ conflict しないように、routing を決定
5. 通り得るチャネルのうち実際に採用されなかったもの
   カウンタから 1 を減算
6. 3. に戻る

Routing の例 (1)

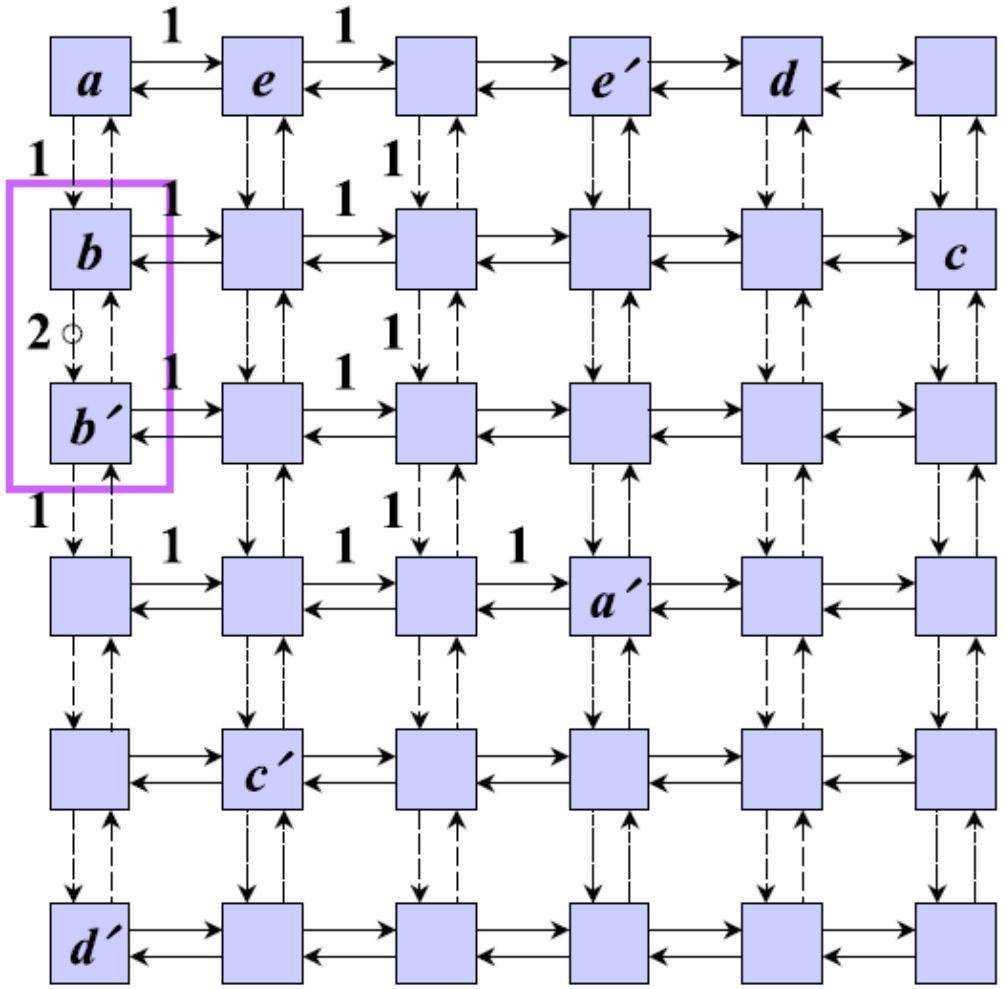


干渉グラフ

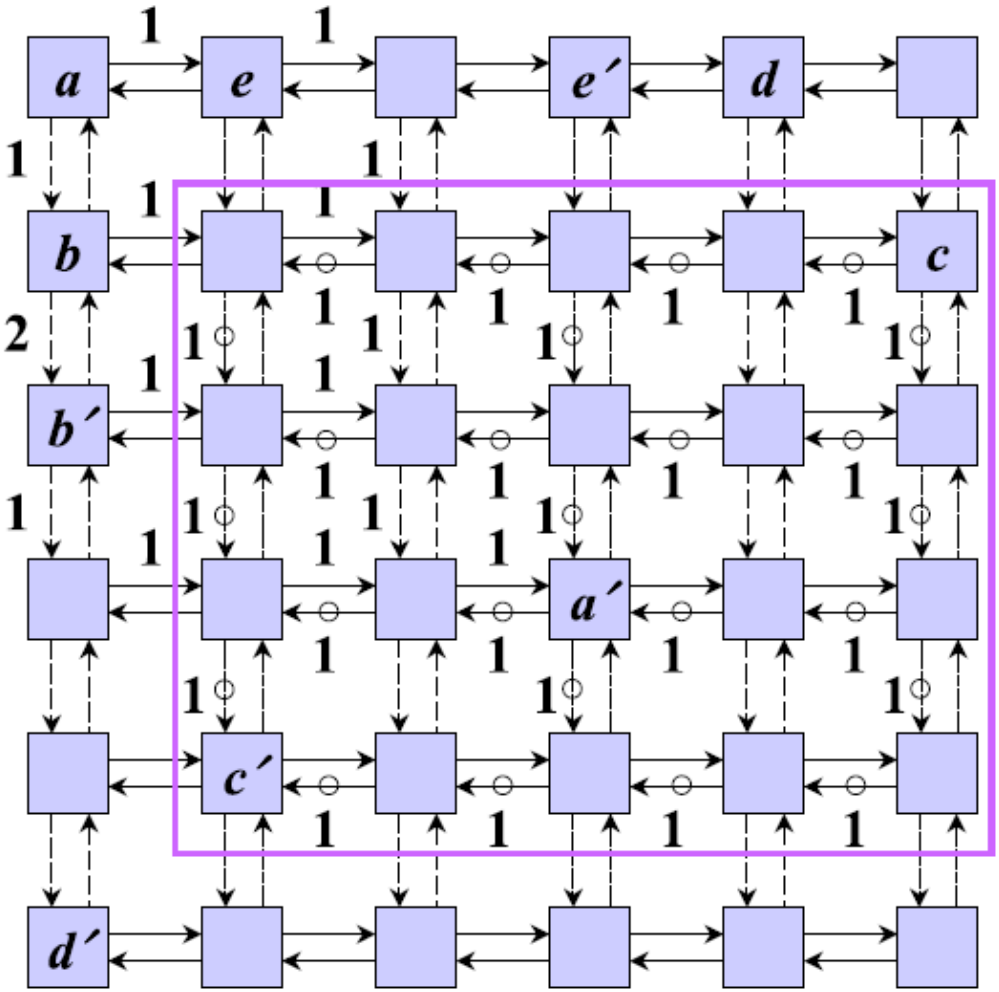


[a,a'] のカウンタ加算

Routing の例 (2)

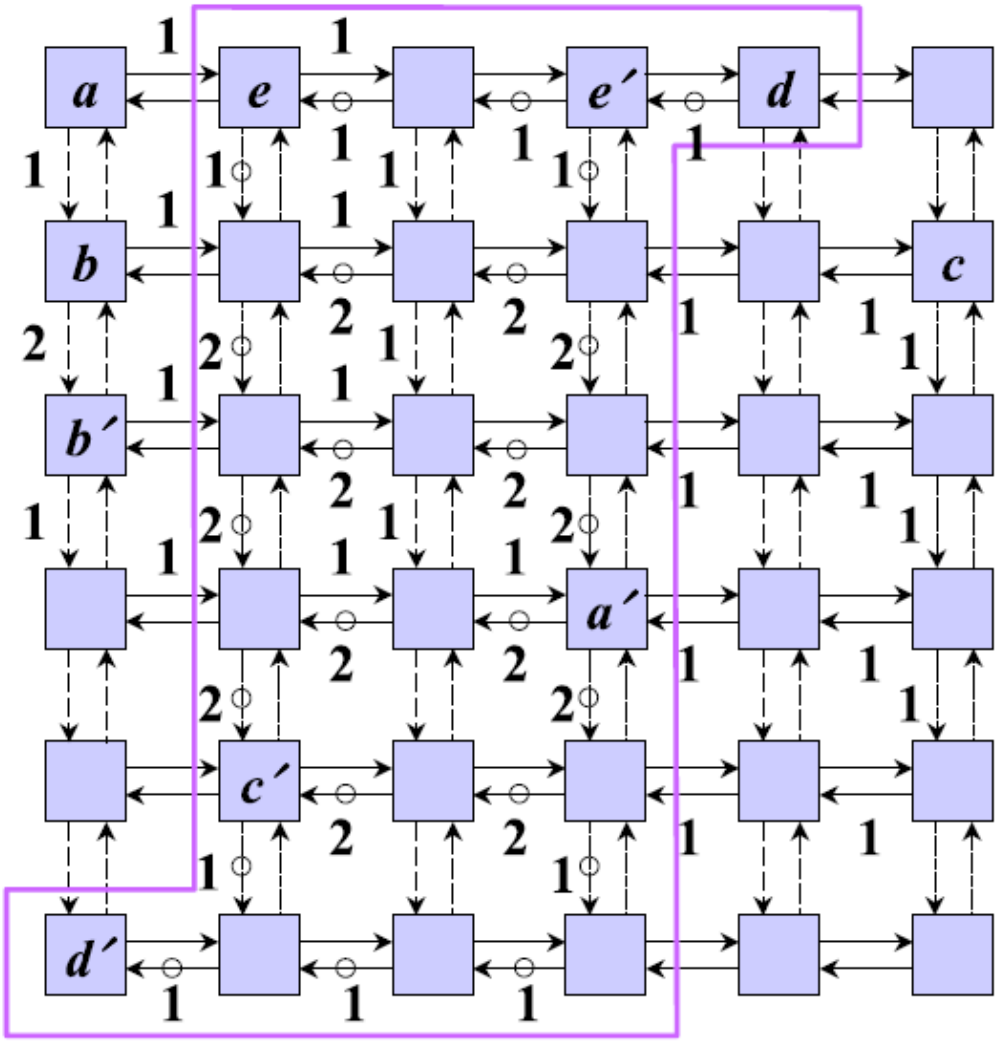


[b,b'] のカウンタ加算

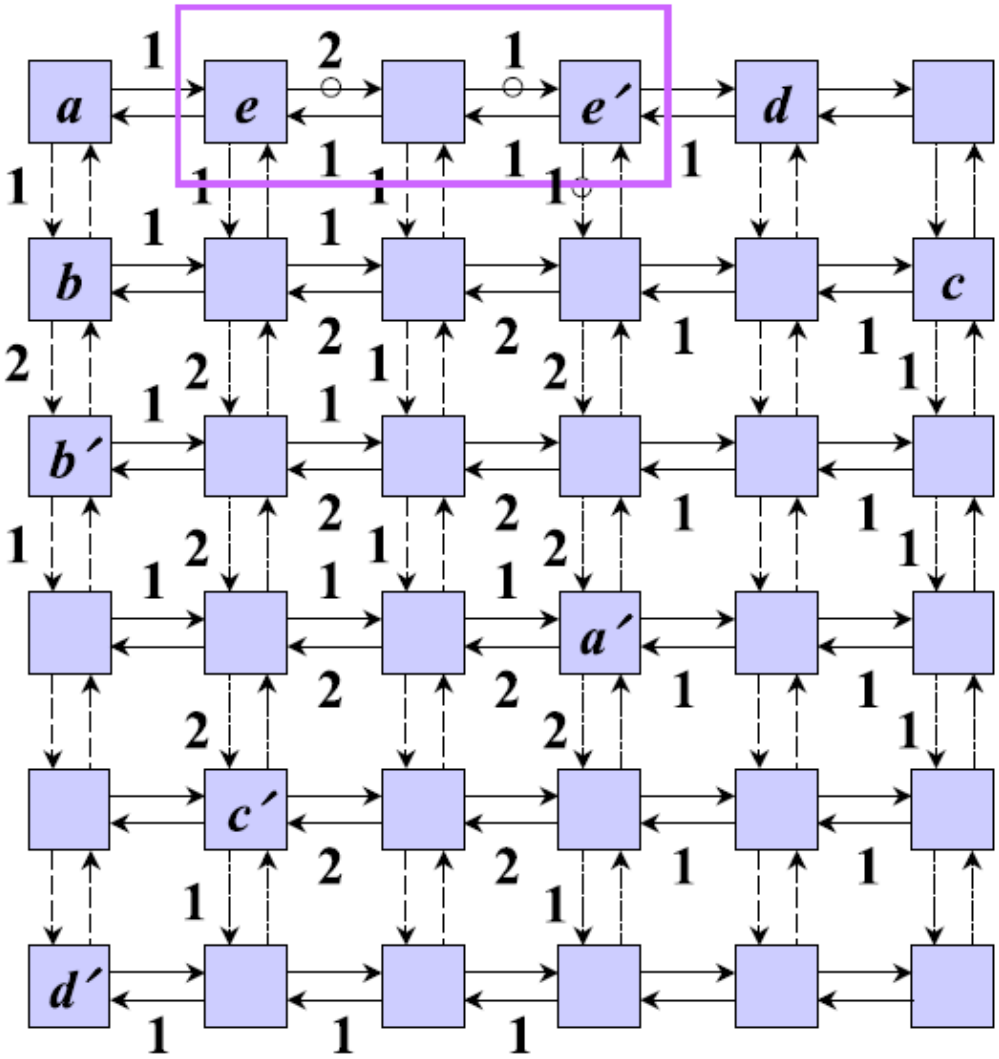


[c,c'] のカウンタ加算

Routing の例 (3)

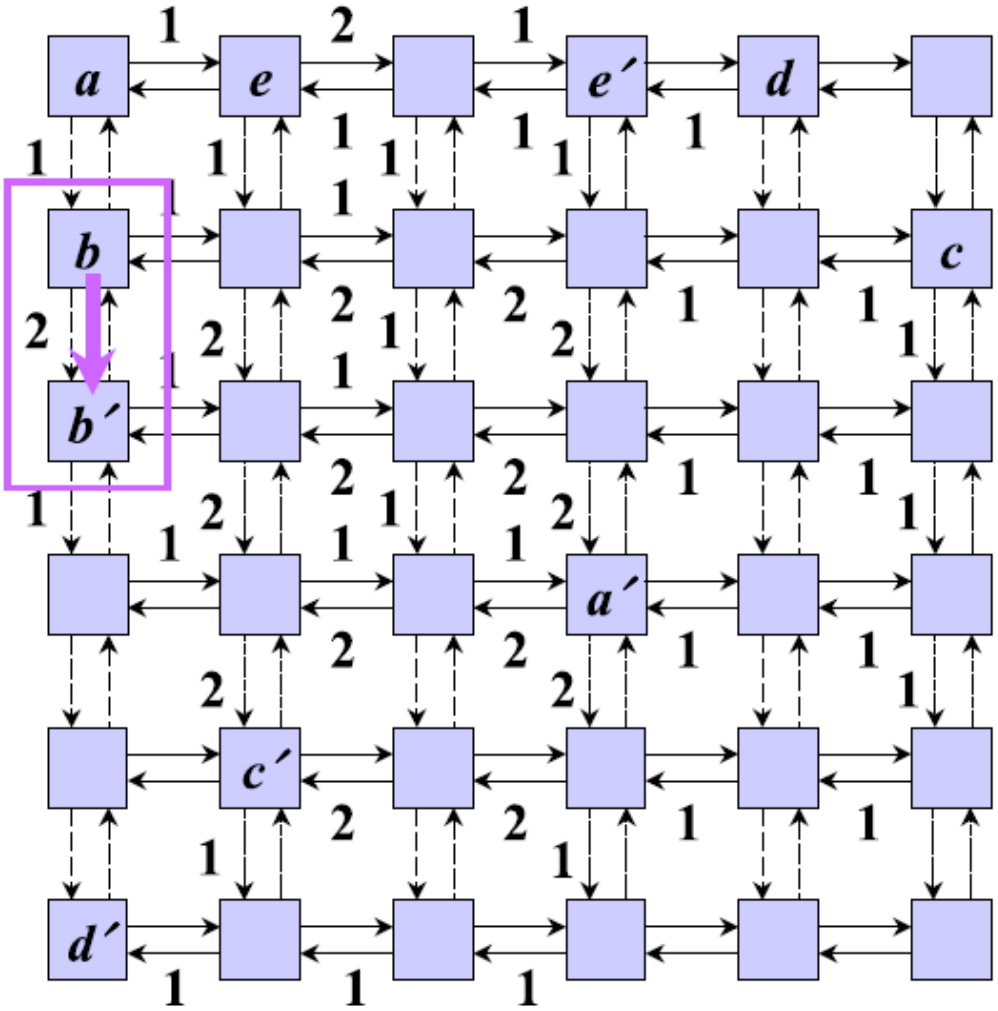


[d,d'] のカウンタ加算

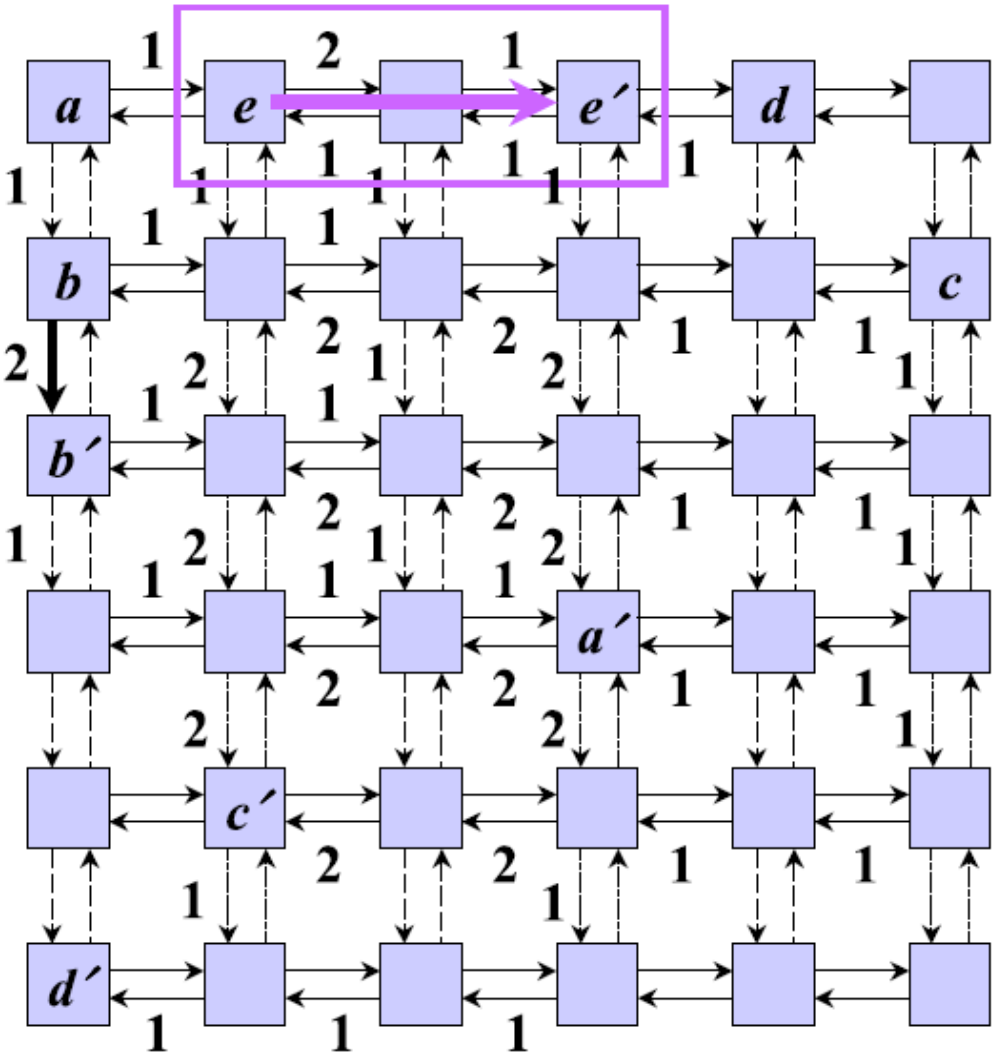


[e,e'] のカウンタ加算

Routing の例 (4)

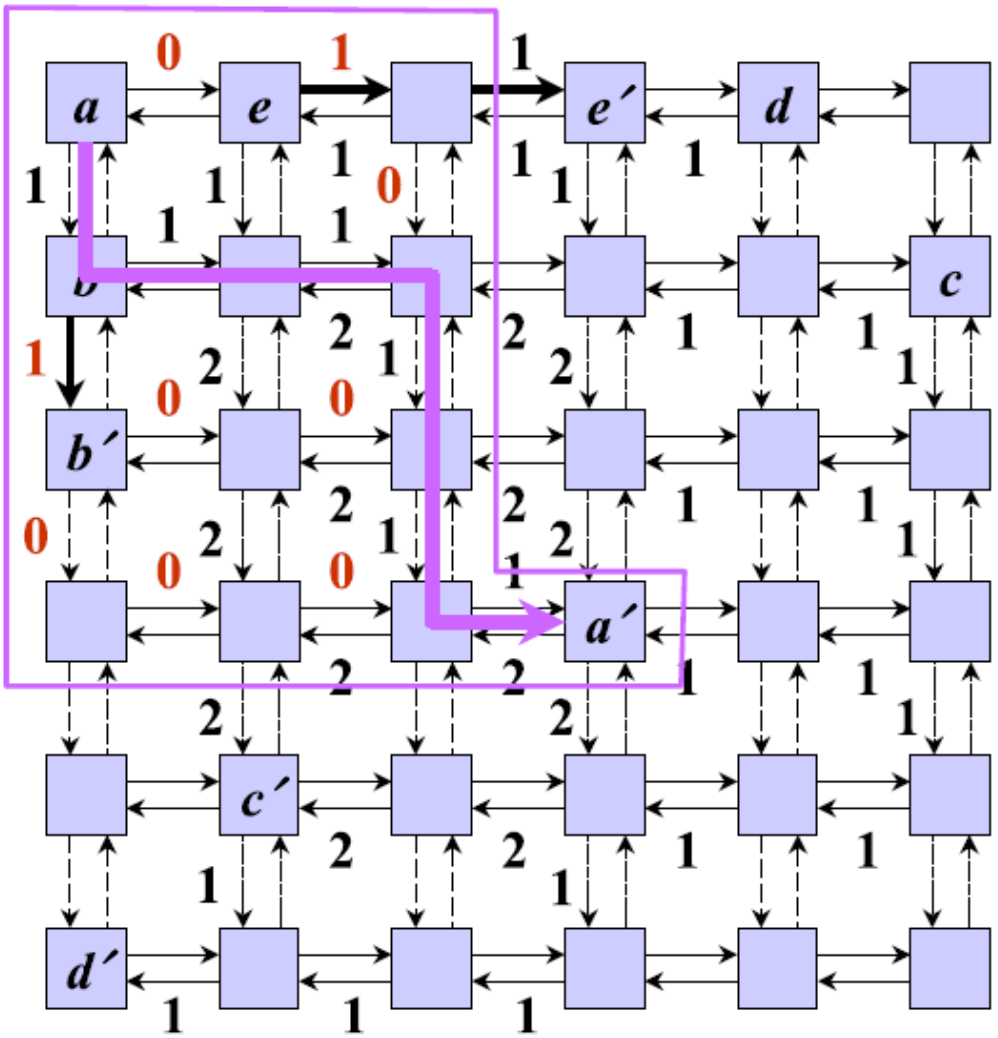


[b,b'] の routing

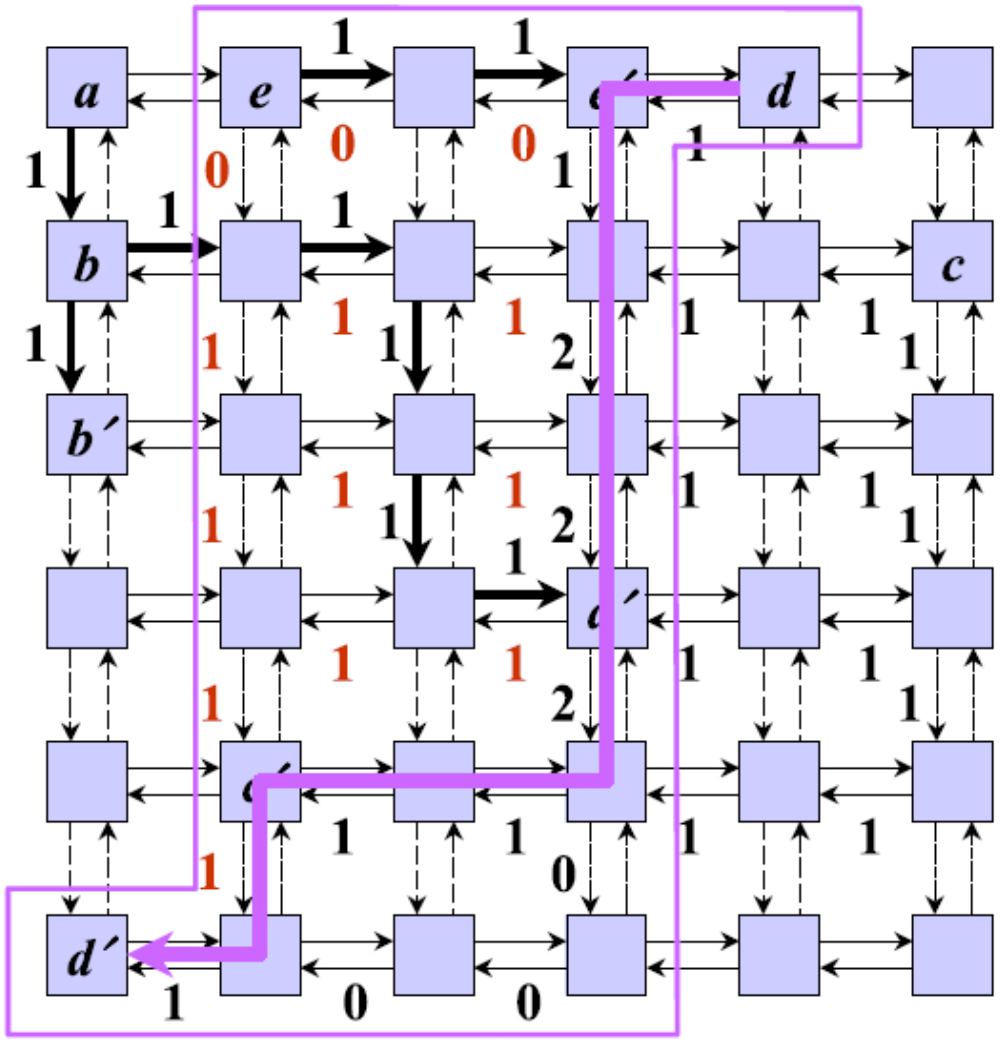


[e,e'] の routing

Routing の例 (5)

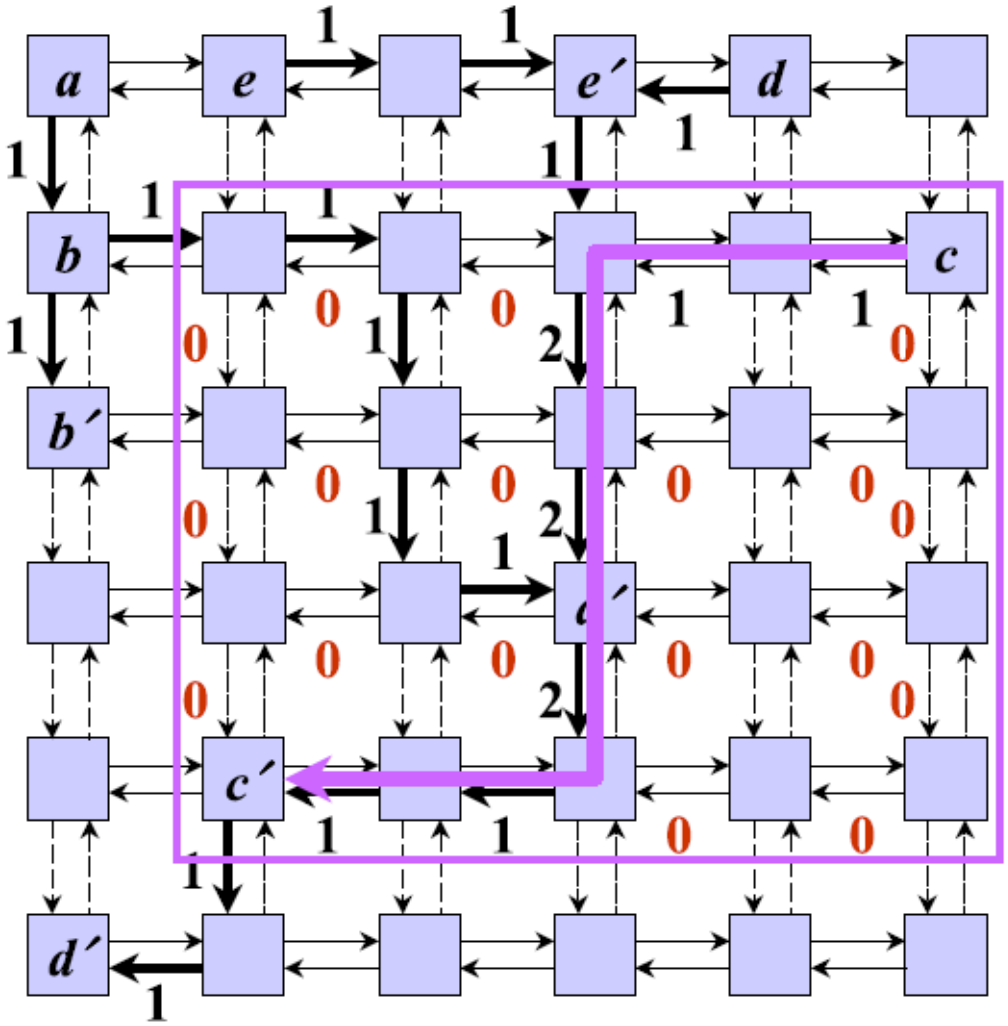


[a,a'] の routing
＋ カウンタ減算



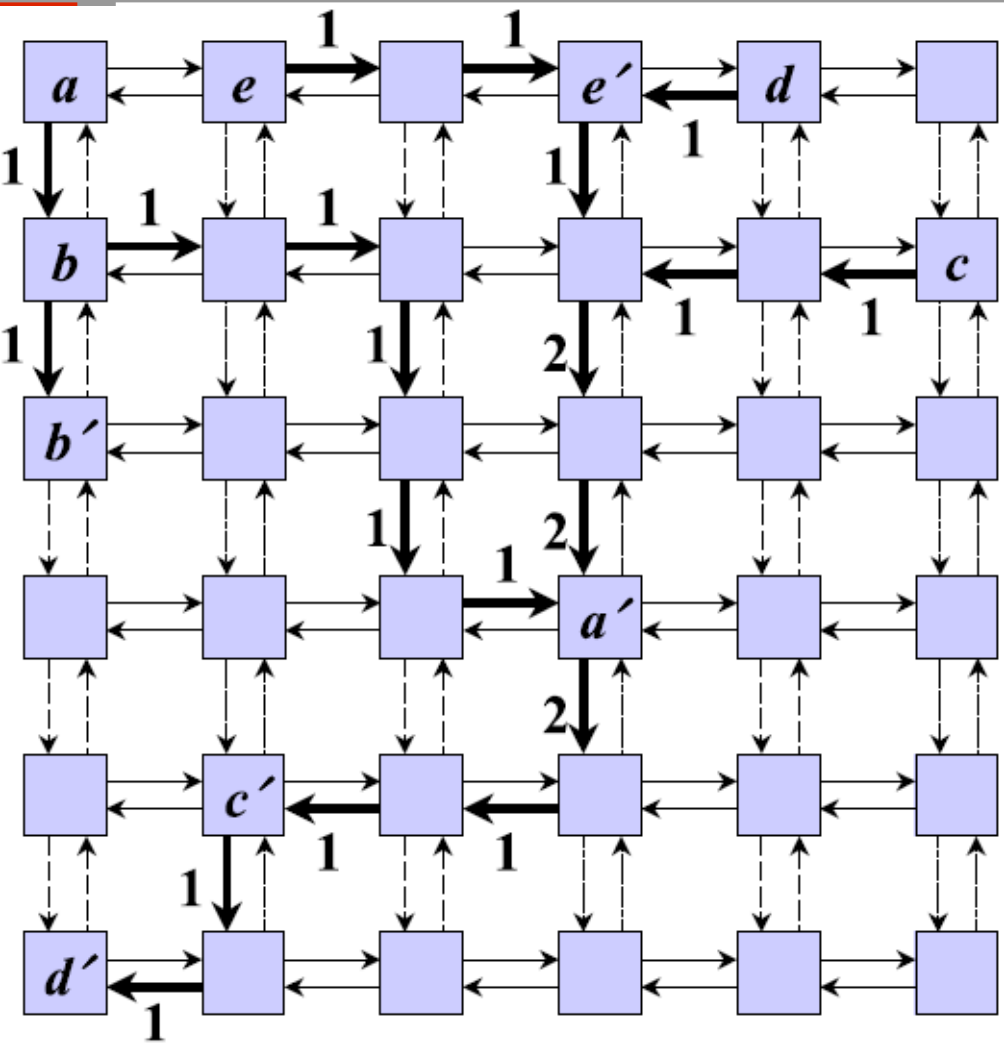
[d,d'] の routing
＋ カウンタ減算

Routing の例 (6)



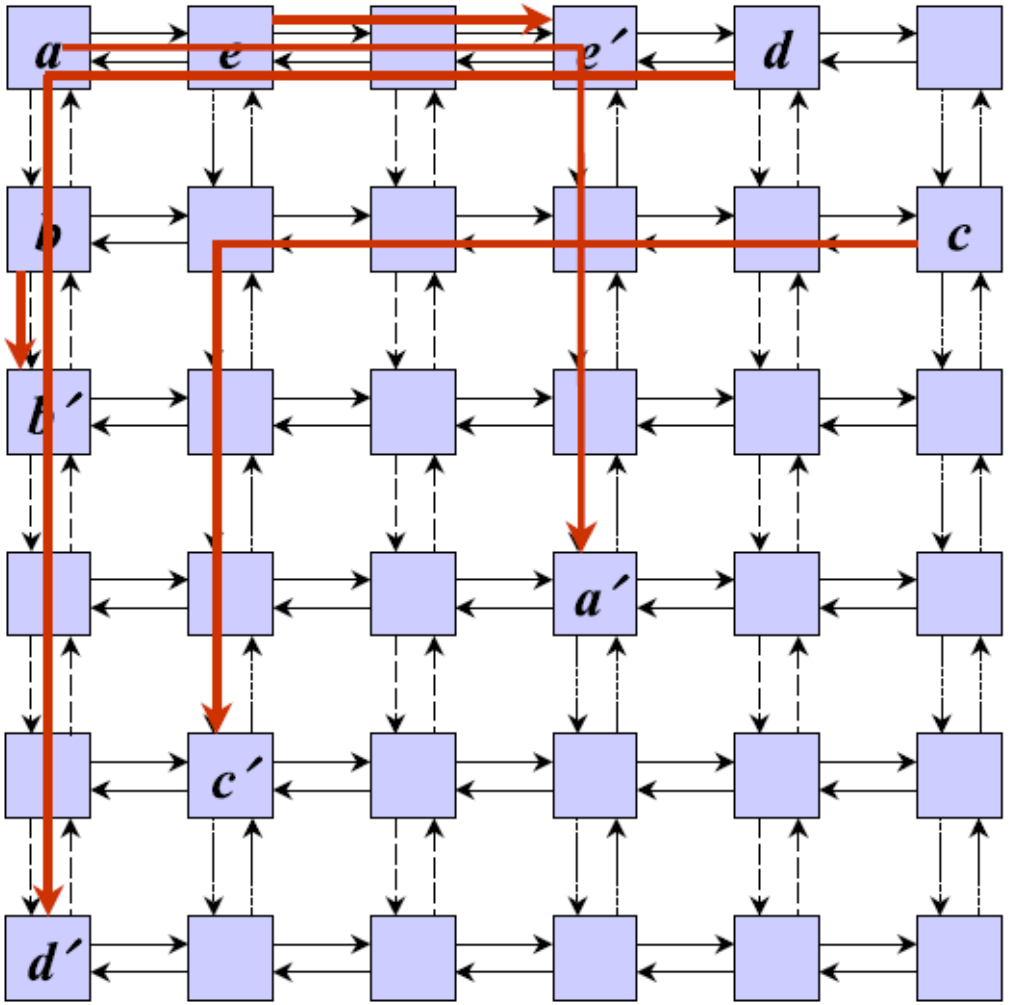
[c,c'] の routing
＋ カウンタ減算

Routing の例 (7)



最終結果

(conflict なし、使用チャネル数 20)



[参考]X-Y 法による結果

(confilct2 回、使用チャネル数 22)

(3)routing table の作成

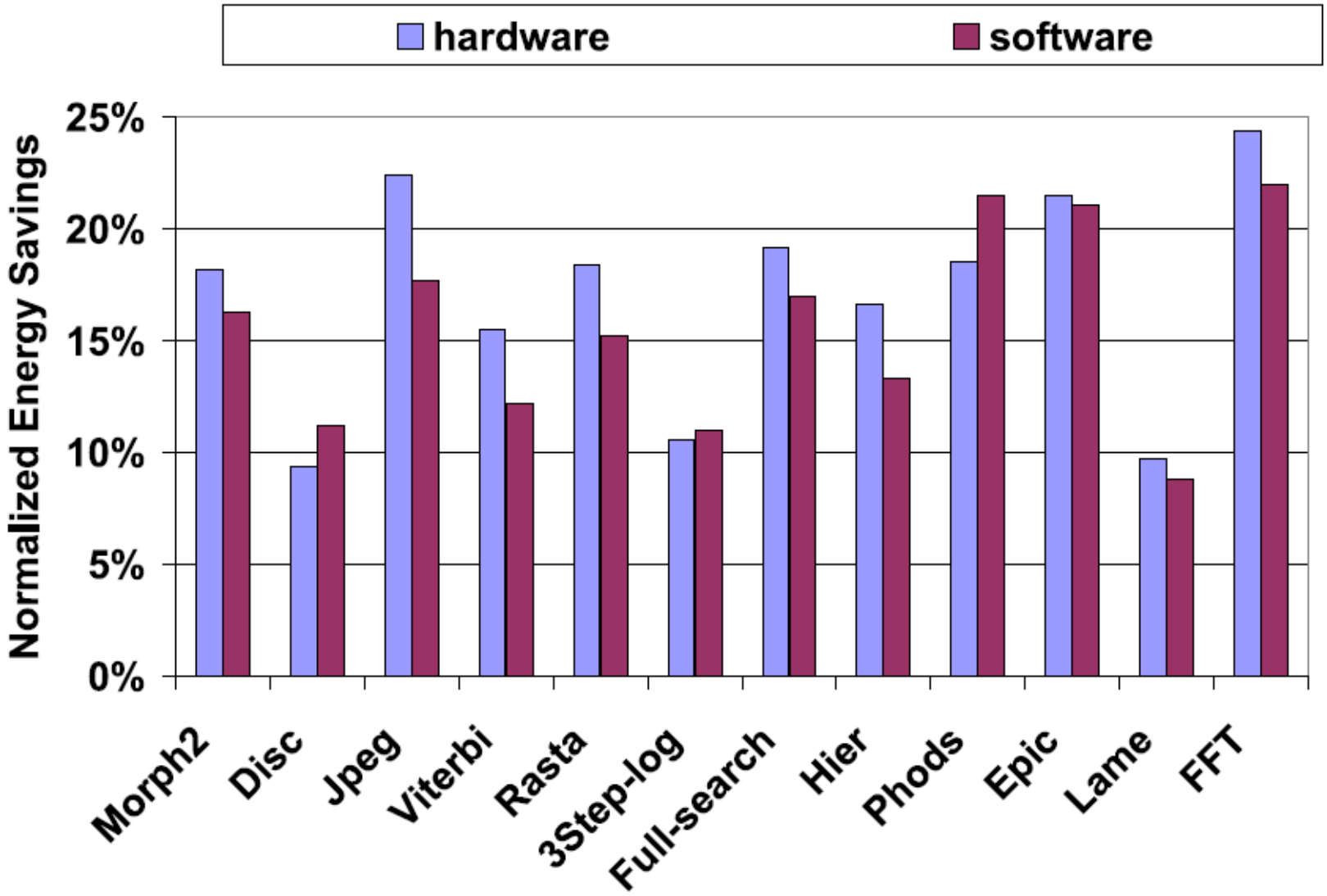
- 最終目的地から次に進むべき方角を索く表を作成
  - router 毎に正しい表を持たせる
- 表の内容は全て静的に決定
  - 実行時に書き換わったりしない

評価方法

- アレイベースのアプリケーションを用意
  - メッシュ型アーキテクチャに適したアルゴリズム
- 並列最適化後のコードに対して我々の手法を適用
  - コンパイル時間の増加は約 55%
- 2 つのチャネル電源操作方法について評価
  - ハードウェア的手法
    - しばらく通信がなかったら自動で off
    - off 中に通信が来たら自動で on
  - ソフトウェア的手法
    - コンパイラが on/off 命令を明示的に挿入
- シミュレータ上で評価

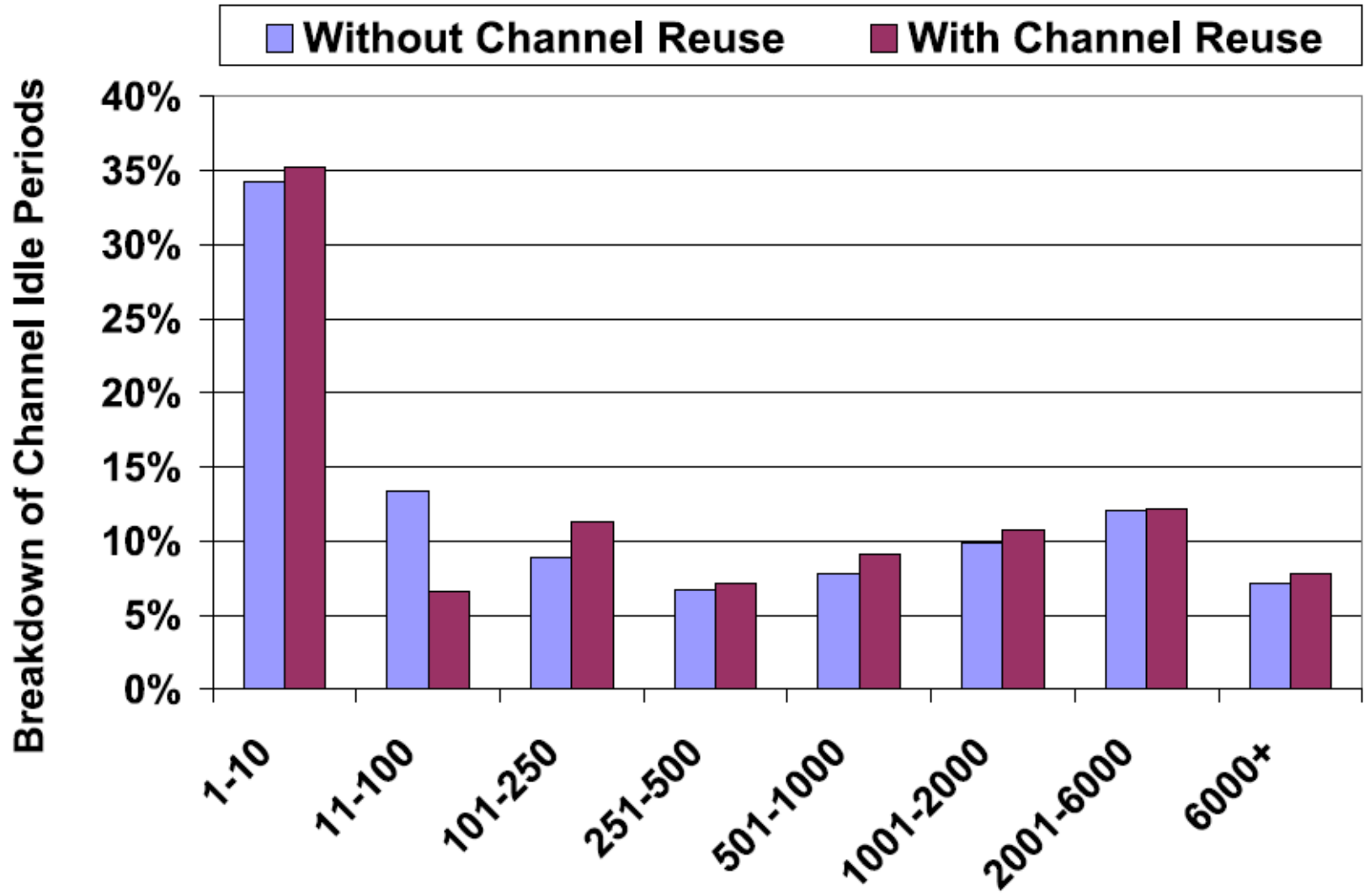
電力の削減

- チャネル操作手法に拘わらず効果的



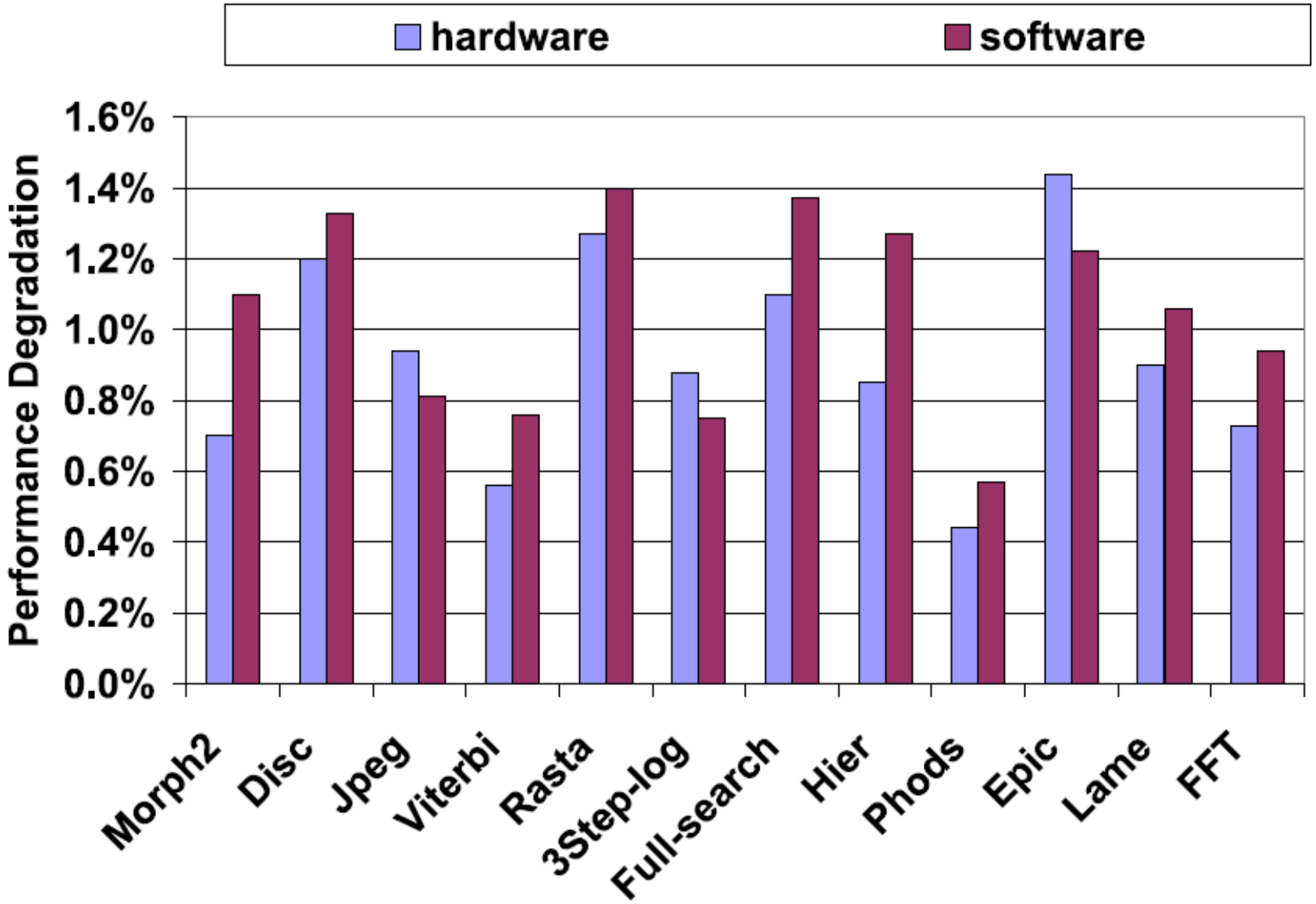
アイドル期間のサイズ別分類

- アイドル期間の破片がより大きい塊へと連結された
  - チャネル電源の切替えによるオーバーヘッドを削減



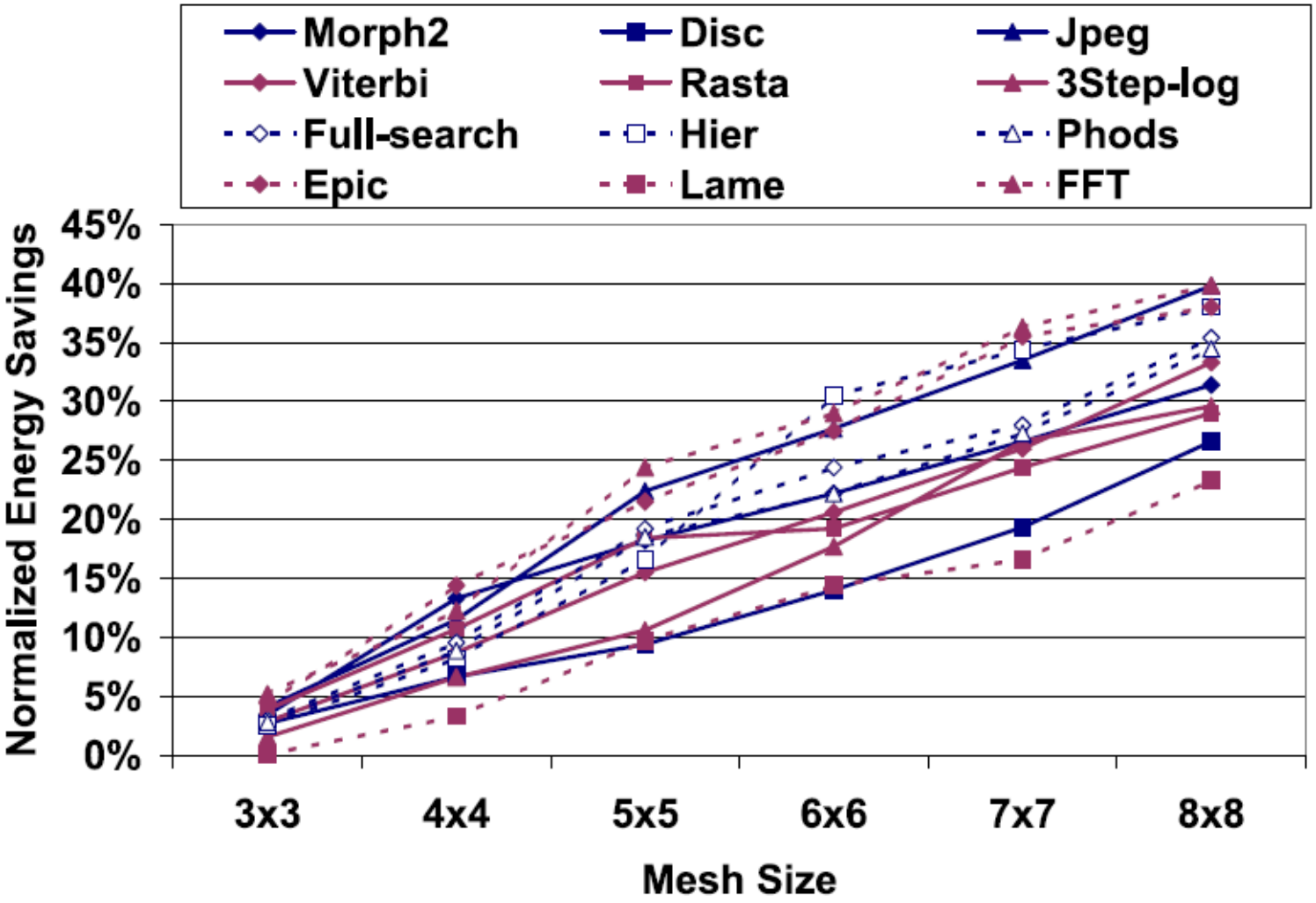
実行性能の低下

- ほとんど無視できる



メッシュサイズによる効果の変化

- サイズが大きいほど効果も大きい
  - とり得る routing の選択肢が膨大になる



まとめ

- コンパイラ主導で NoC に対する電力管理を行った
  - 複数の connection 間で通信チャネルを共有
  - チャネル割り当て問題を heuristic に解決
- 15% 以上の電力削減に成功した
  - チャネル電源の操作手法に依存しない結果