Empowered by Innovation NEC

pgbenchによるEarly Lock Releaseの評価
（ + ついでにPostgres 9.2の強化点）

2012年 9月29日

NEC
情報・ナレッジ研究所
堀 川 隆

主旨

pgbenchによりEarly Lock Release（ELR）の効果を測定

- 前回(1/22)発表時，笠原さんより，「DBT-1はトランザクションの性質を考えるとELRの効果が出にくい．pgbenchだと効果が出るのではないか」とのコメントを頂いた．
- 鈴木さんより，「pgbenchの結果が出たら，5分～10分で良いので追加報告してね」と依頼された．

前回との違い

- ベンチマーク
  - DBT-1 → pgbench
- PostgreSQLのバージョン
  - 9.0.4 → 9.1.2
- ELRの実装方法（trivial）
  - XLogFlush()の呼び出しをより後に移動 （詳細は付録）

+α：PostgreSQL 9.2 β2（当時）

- 測定実験中にリリース ⇒ これも測定してみた

Empowered by Innovation NEC

測定結果

測定条件，方法 など

pgbench DB の scale factor = 100 ⇒ ~1.6GB

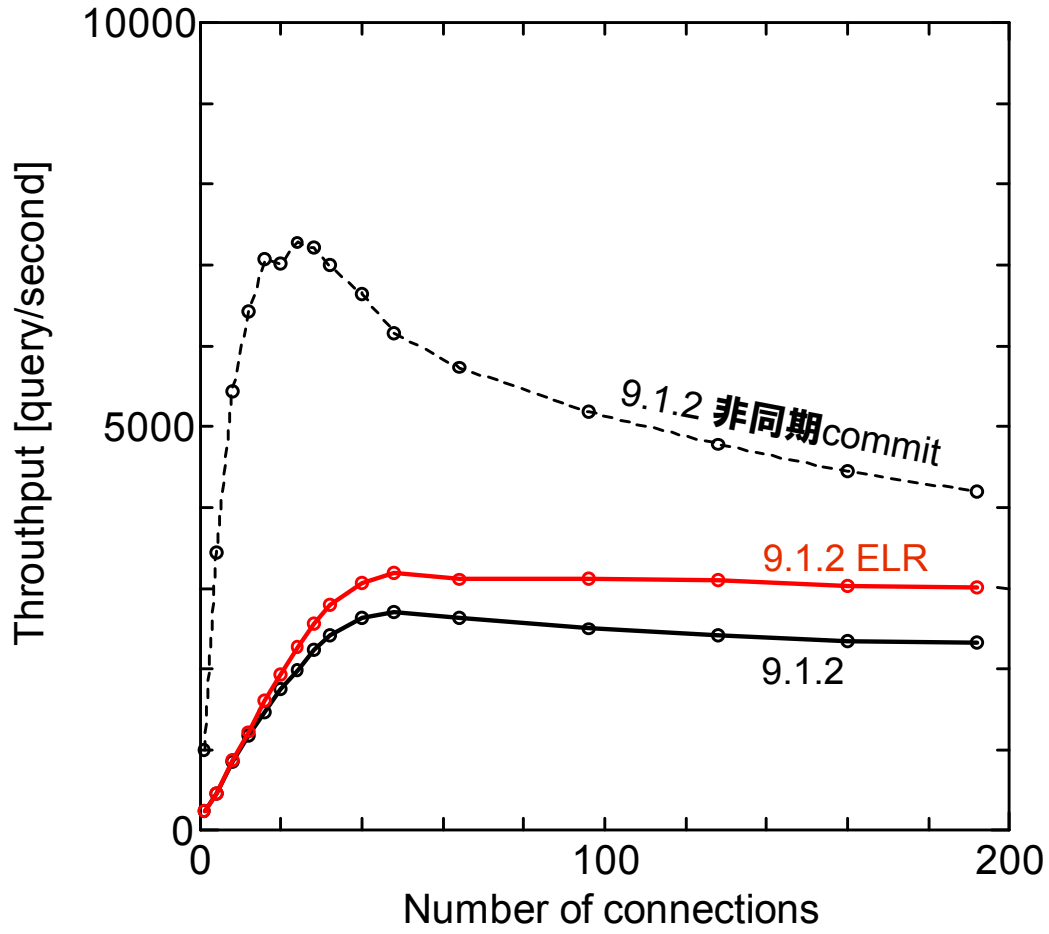
ベンチマーク方法

- ワークロード：標準（r/w あり）と Select only（readのみ）の2種類
  - Select only は，9.1.2と9.2βの比較で使用
- 5分測定 × 3回実施 ⇒ 平均値をベンチマーク結果（TPS値）とした
  - Robert Haasの測定方法に準拠（cf. PostgreSQL Conf. 2012）

測定環境

- Hardware
  - CPU: Intel E7310 (4cores/chip) × 4, 1.6GHz, FSB 1033MHz
  - memory: 16GByte
  - Disk: 1) OS and AP, 2) DB data, 3) WAL
- Software
  - DBMS: PostgreSQL 9.1.2
    - NUM_BUFFER_PARTITIONSを256, NUM_LOCK_PARTITIONSを64に増やす
    - shared_buffer = 4GB
  - OS: Linux 2.6.18-164.6.1.el5 (CentOS)

EARの有無（+非同期commitの効果 -参考として-）



⇒ Early Lock Releaseの効果はある

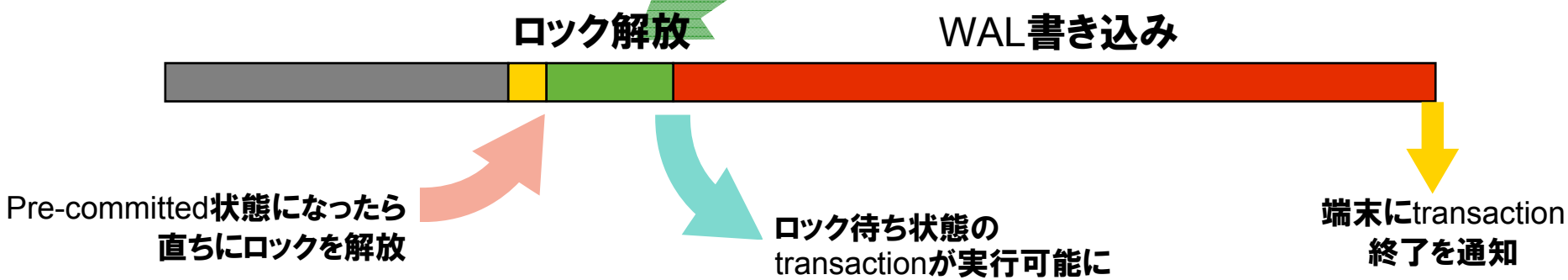
Early Lock Release の効果

従来

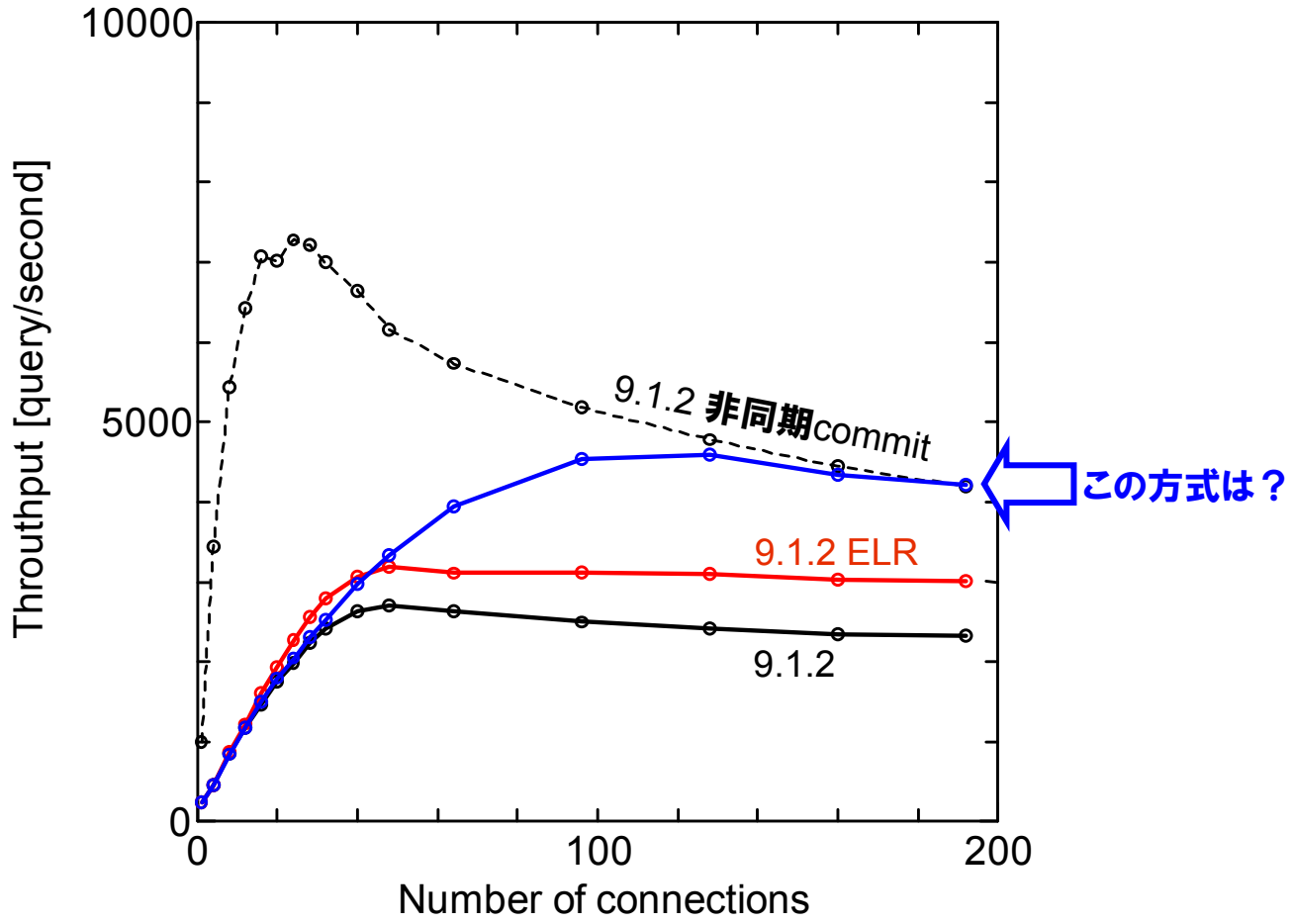
ロック解放：WAL書き込み
時間の分，早くなる

⇒ これで総てなのか？

Early Lock Release

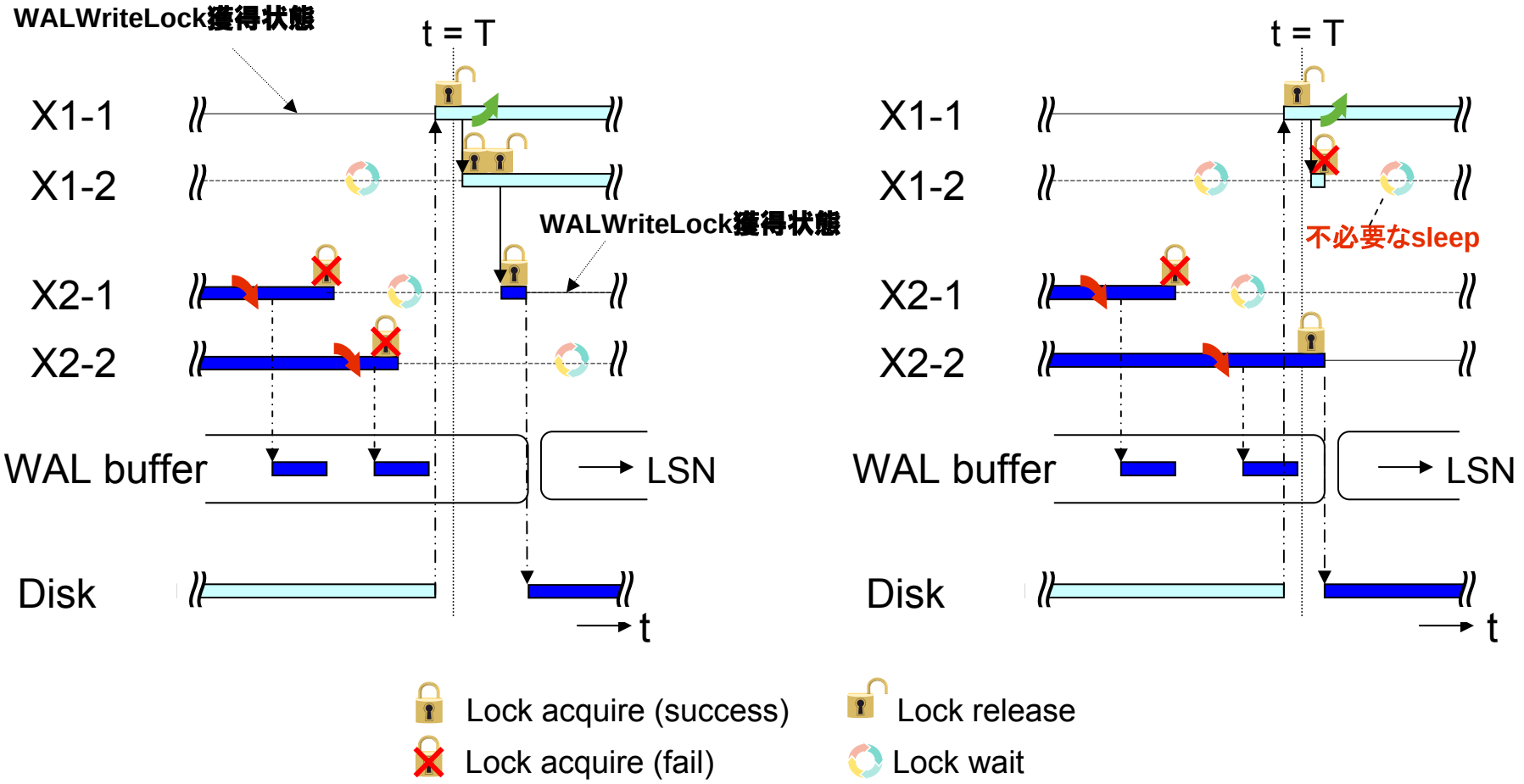


EARでも非同期commitでもない



⇒ Early Lock Releaseより効果のある方式とは？

複数のWAL書き込み待ちプロセスをwake up



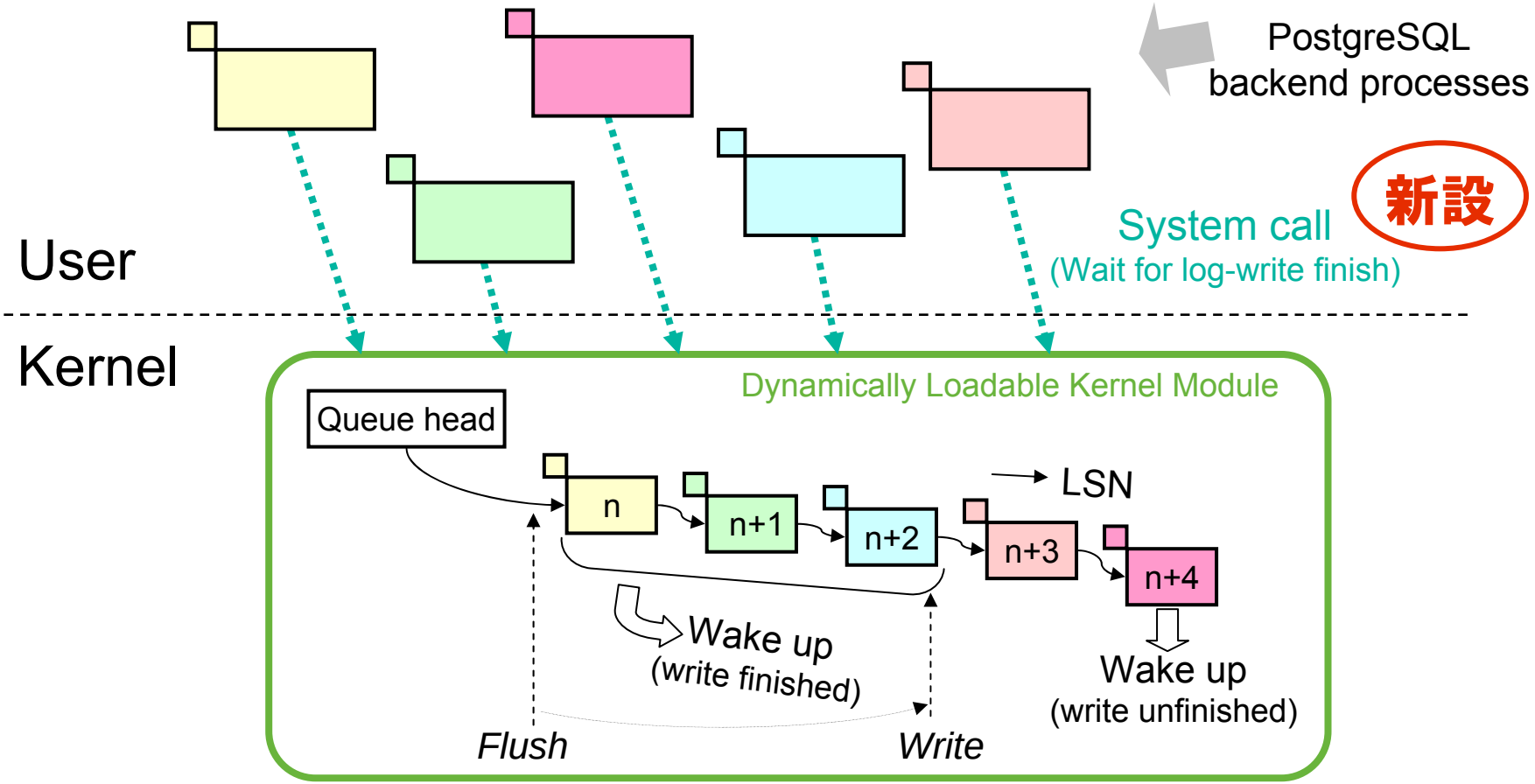
期待どおりの順番でwake up

Wake upの順番が狂うと. . .

Linux カーネルモジュールを作成

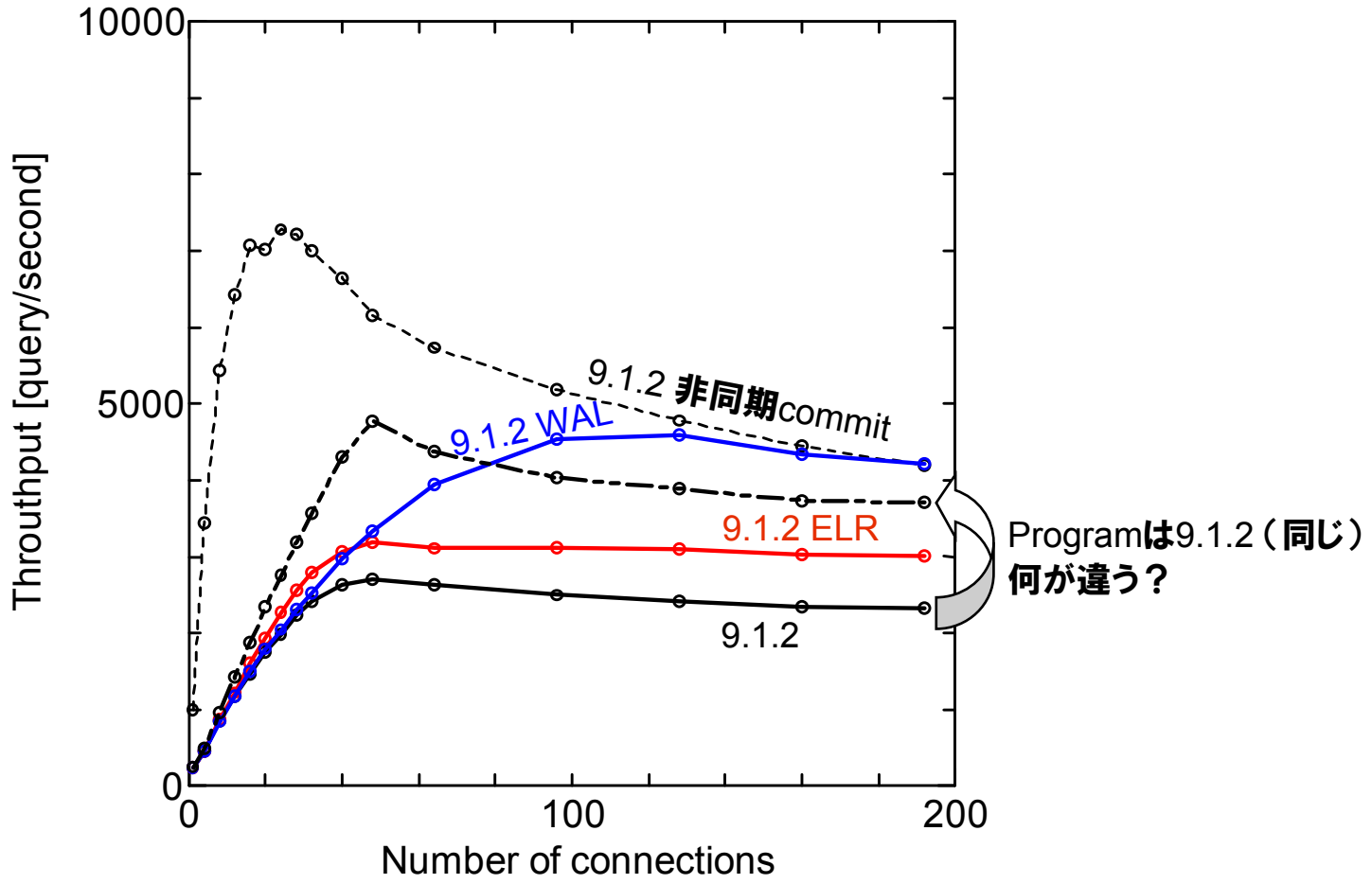
DLKMとして，WAL待ち用のqueueをインプリメント

- 書き込み完了時，そのLSN以下(＋LSNの最大値)のプロセスをwake up



cf. Horikawa, T., An Unexpected Scalability Bottleneck in a DBMS: A Hidden Pitfall in Implementing Mutual Exclusion, PDCS 2011, ACTA Press.

今度は？



pgbenchの特性(問題？)

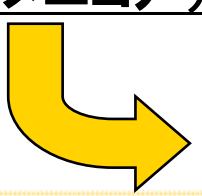
同時接続数が多いと性能が出ない

この測定実験では -s 100

- …特に問題なのは、…pgbench_branches の行数がスケーリングファクタと同じ(つまりデフォルトでは10)しかないため、同時接続数が10を超えるとロック競合が発生して性能が出なくなるということです。
- 現実のシステムではこのような設計は普通は行わないので、実際のシステムでの性能を推し量るという、ベンチマーク本来の目的にはあまりそぐわないことになります。

cf. 石井 達夫, pgbenchの使いこなし,
http://lets.postgresql.jp/documents/technical/contrib/pgbench/

マルチコア（メニコア）の場合は更に問題？

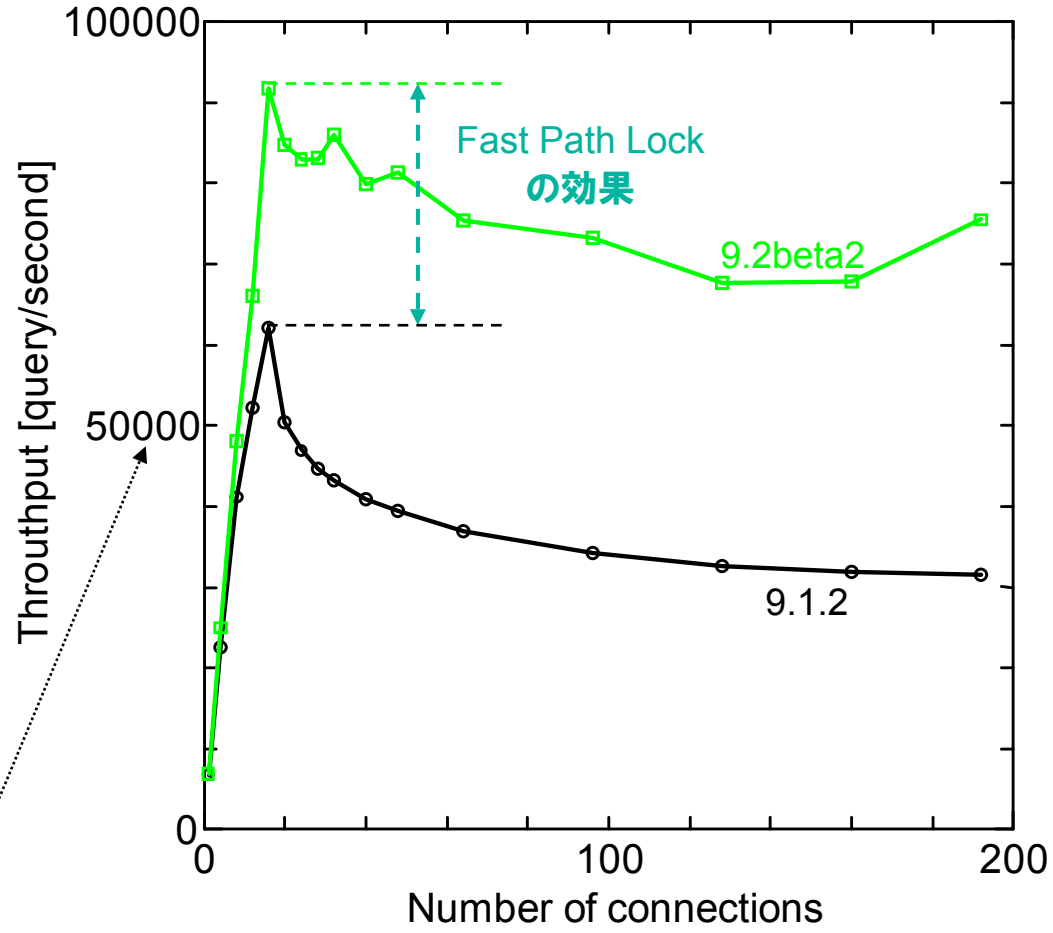


これを検証

DB全体のサイズはほぼ同じ

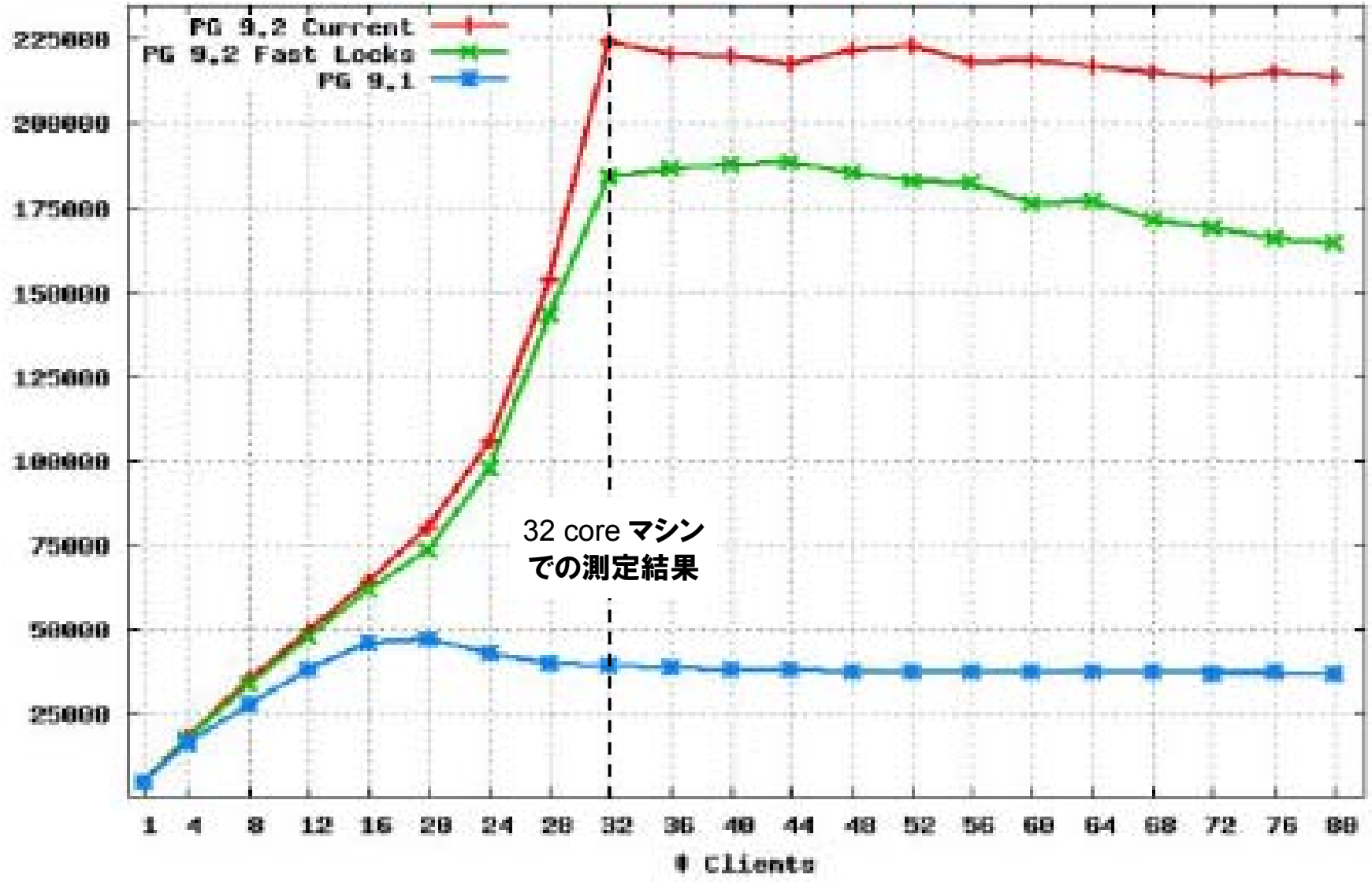
table	touple数(初期値)		x10構成
accounts	scale x 100000	→	scale x 100000
tellers	scale x 10		scale x 100
branches	scale x 1		scale x 10
(history)	0（insertされていく）		

9.2では？　-- select (read) only --



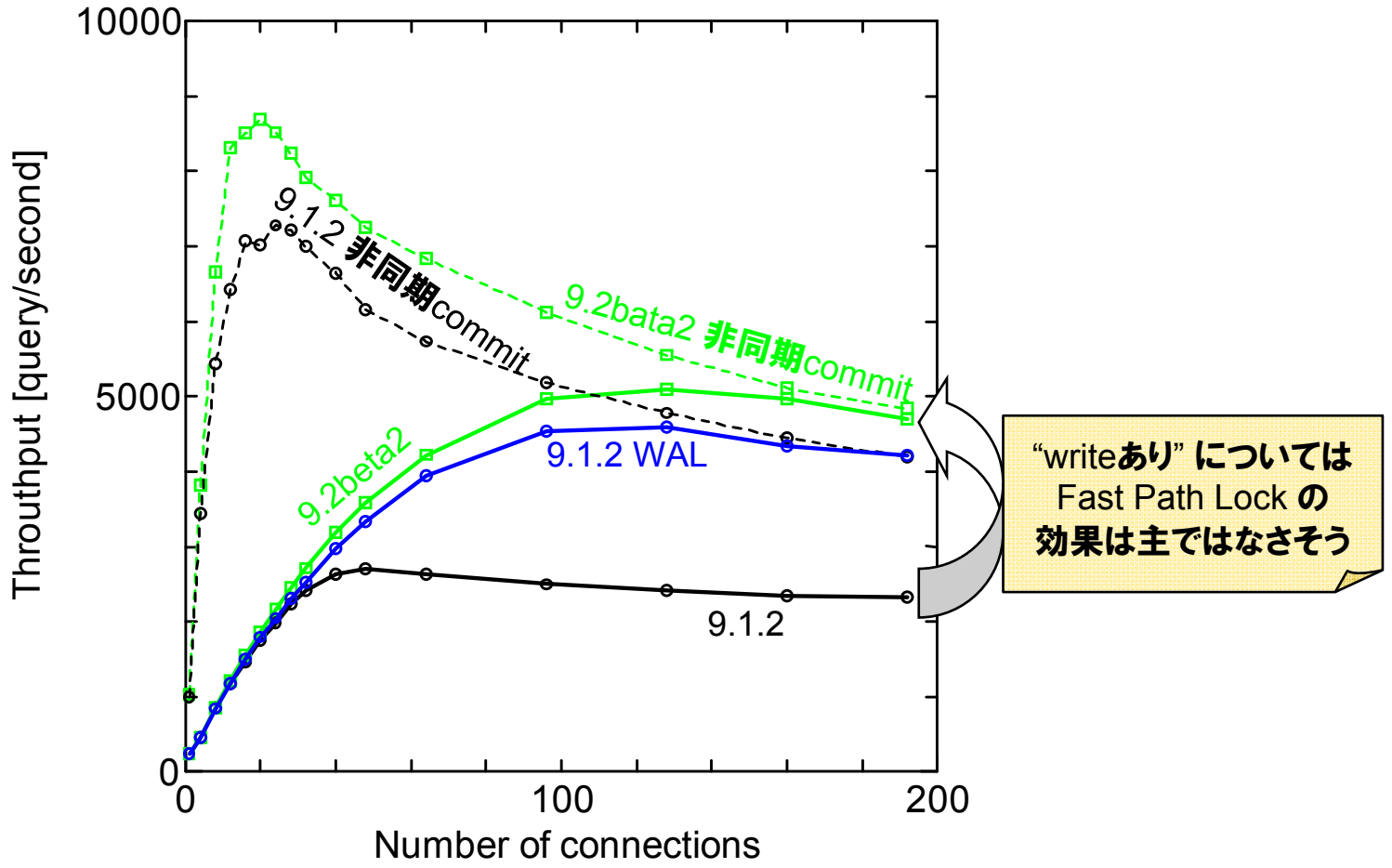
cf. y軸のscaleは他（writeあり）の10倍

（参） PostgreSQL Conf. での発表資料 (select only)



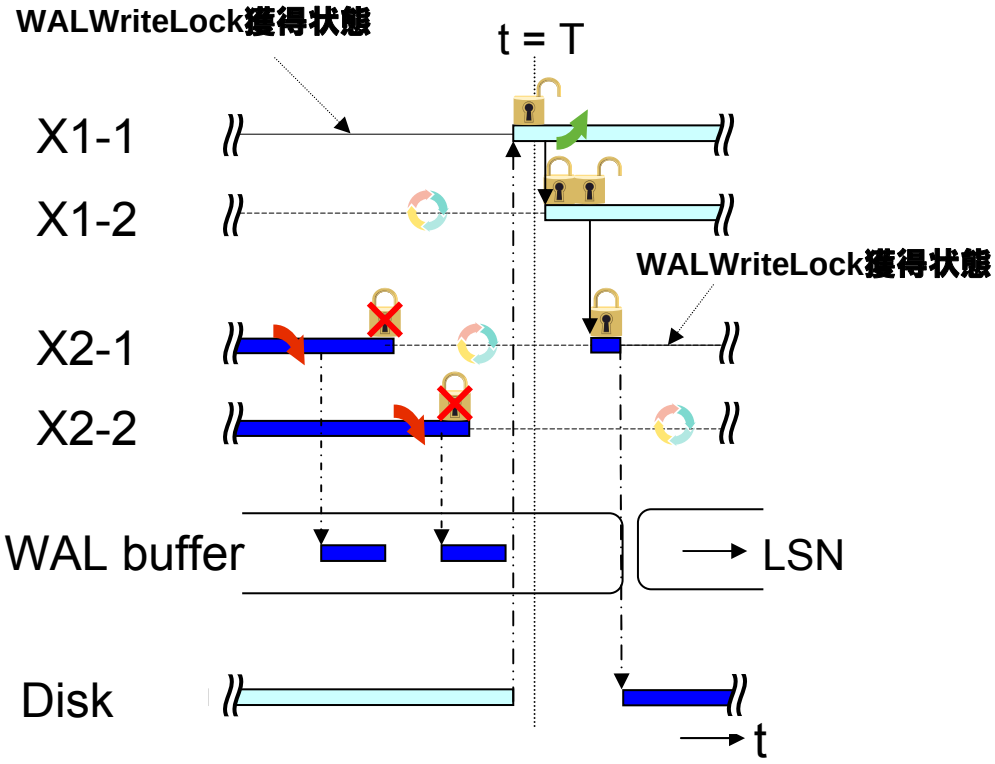
PostgreSQL Conf. 2012 Tokyo, Robert Haas 発表資料 (2012/2/24) by way of H. Takatsuka

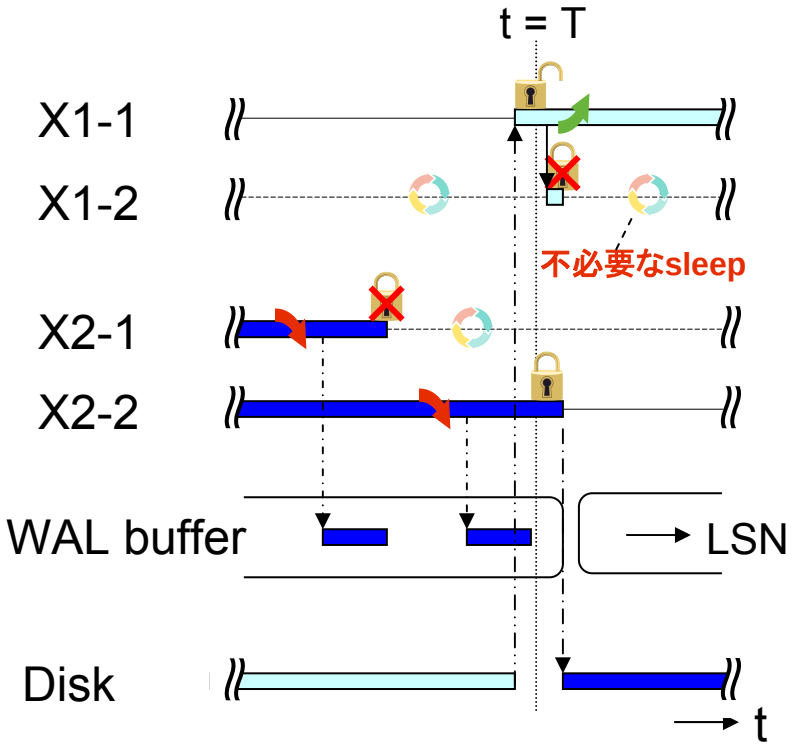
9.2では？　-- writeあり --



⇒ 9.2(bata2) は 9.1.2　WALとほぼ同等

（再掲） 複数のWAL書き込み待ちプロセスをwake up



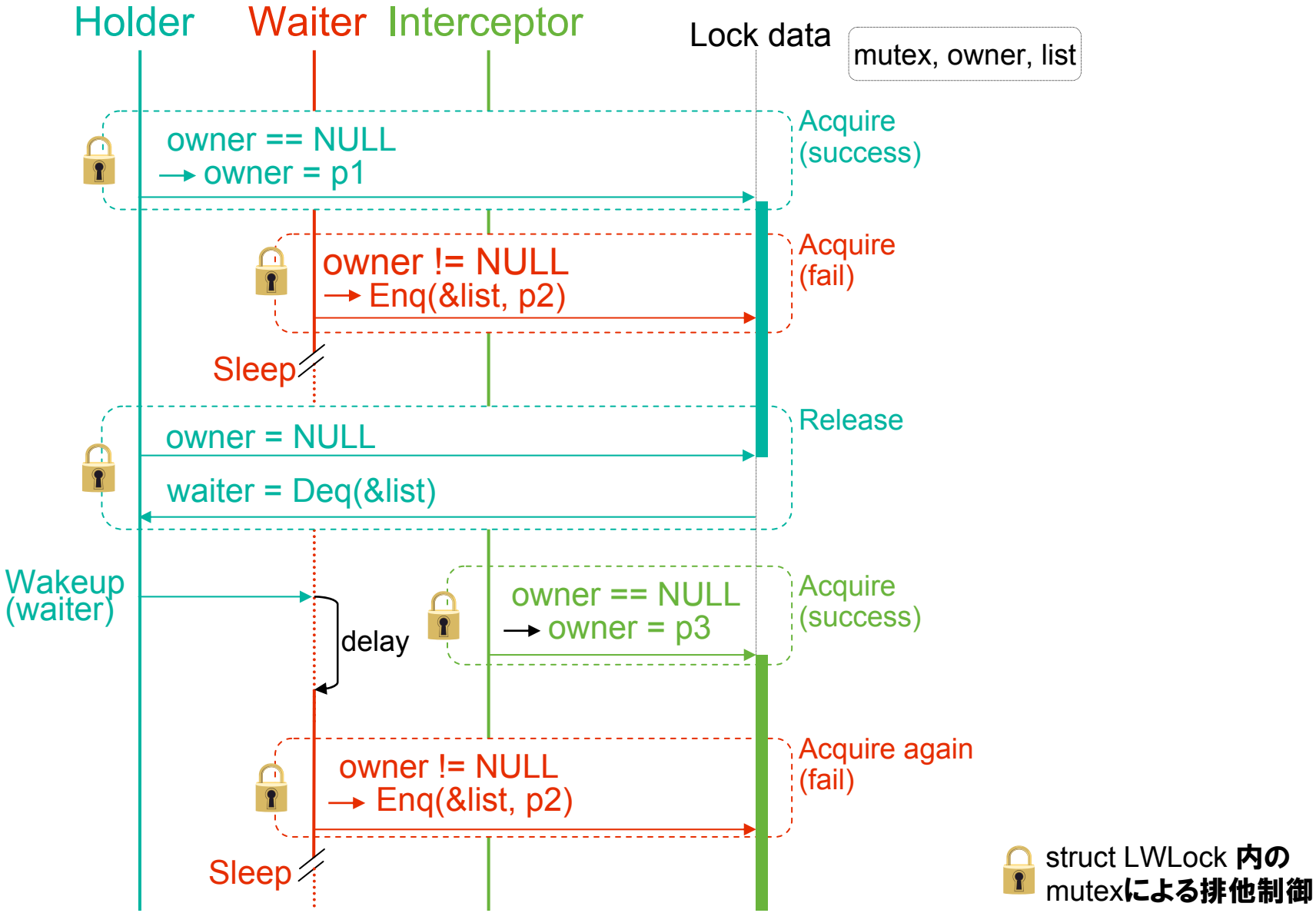


Lock acquire (success)
Lock acquire (fail)
Lock release
Lock wait

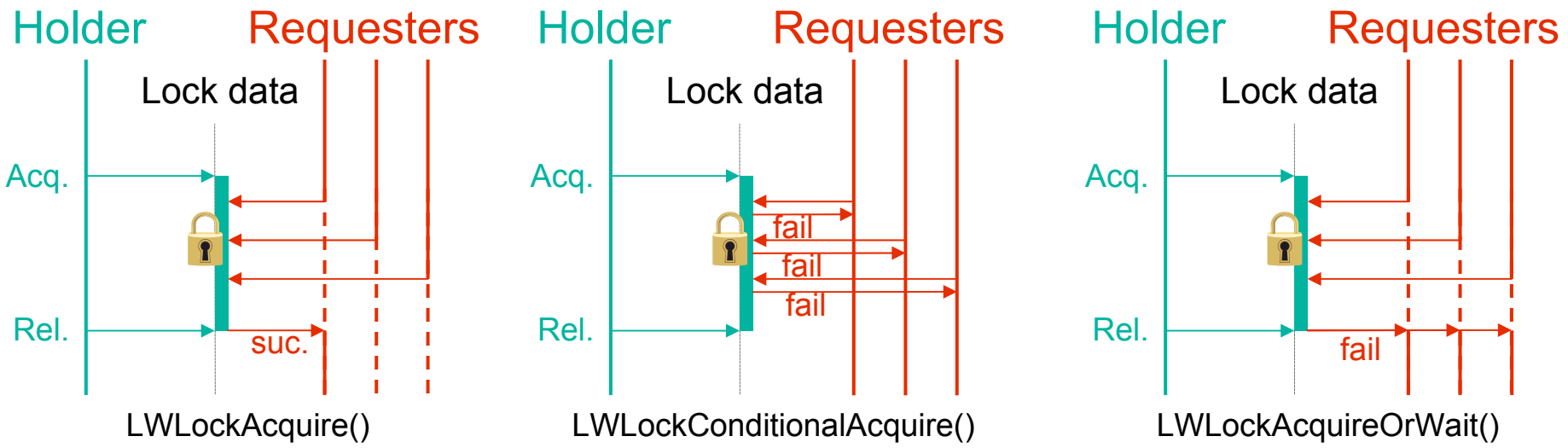
期待どおりの順番でwake up

Wake upの順番が狂うと．．．

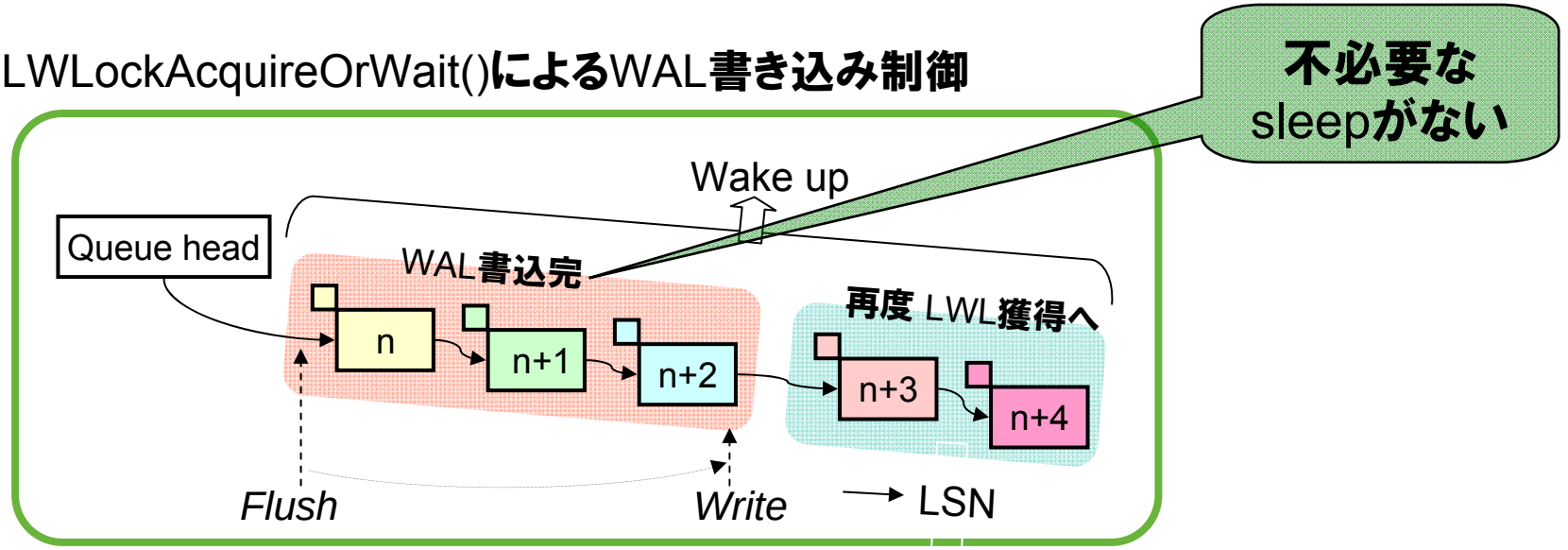
cf. LWLockの横取り



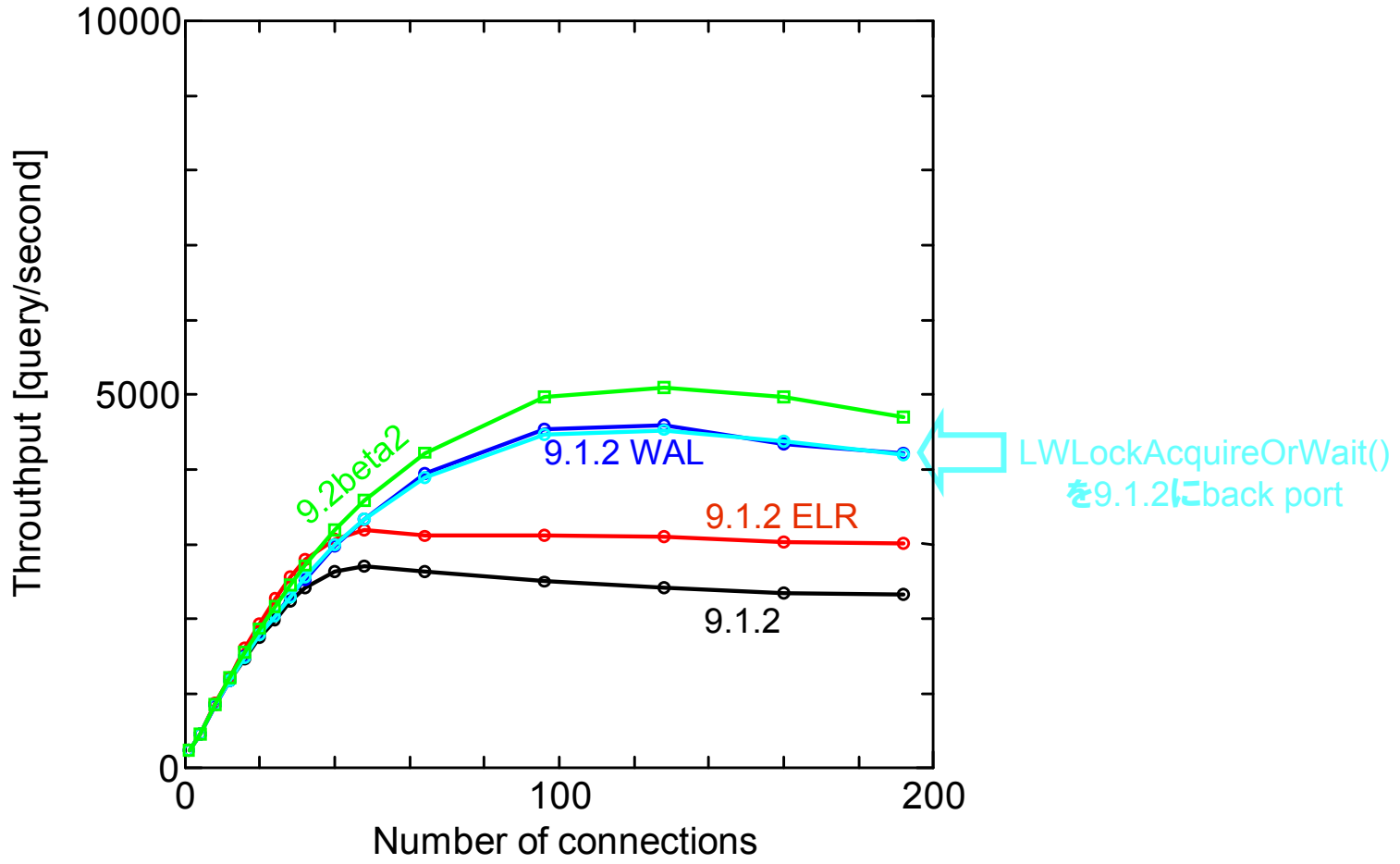
LWLockAcquireOrWait() って・・・



LWLockAcquireOrWait()によるWAL書き込み制御

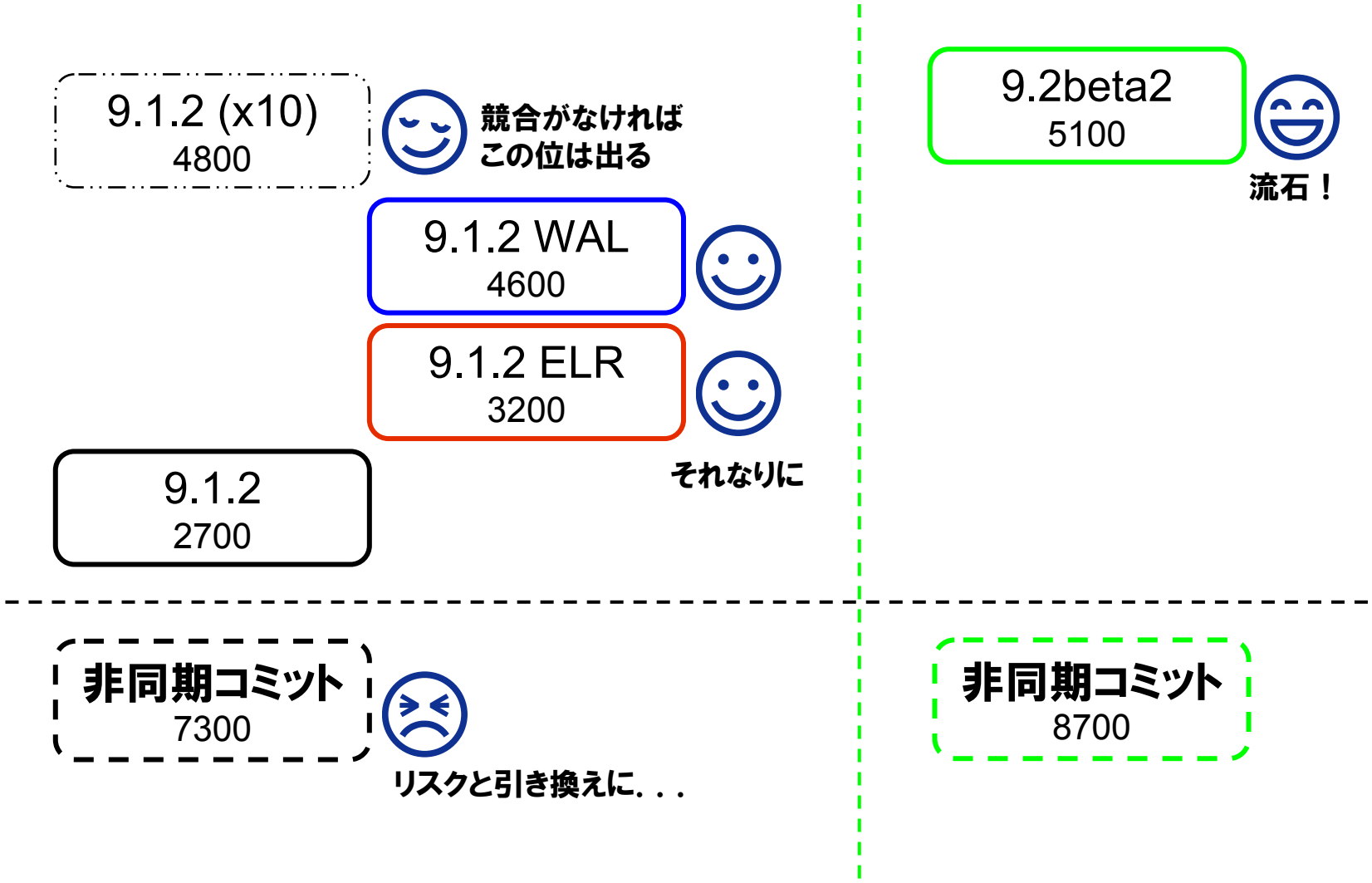


LWLockAcquireOrWait() の効果を検証



⇒ 9.1.2 WALと同等

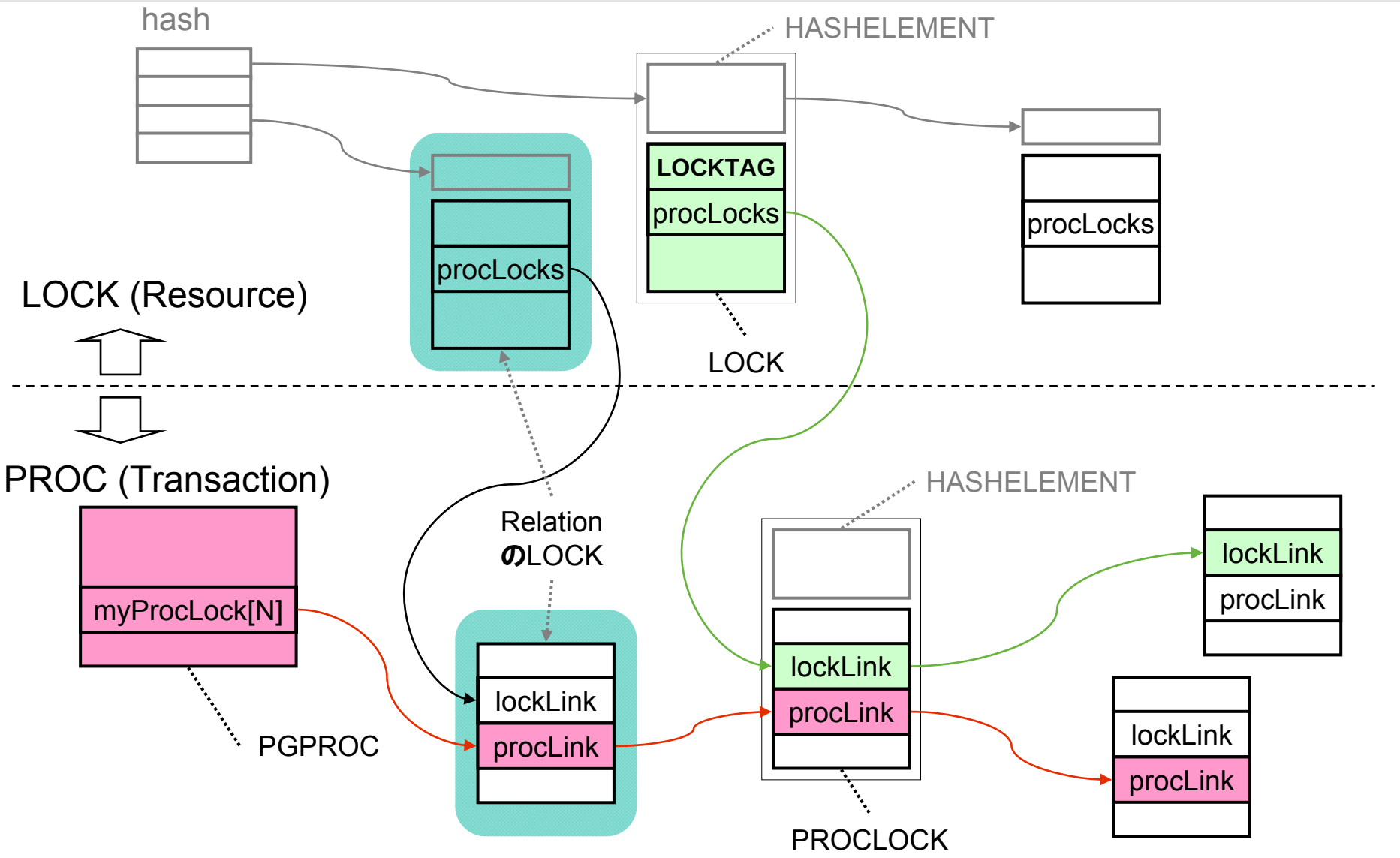
測定結果 俯瞰 -- writeあり --



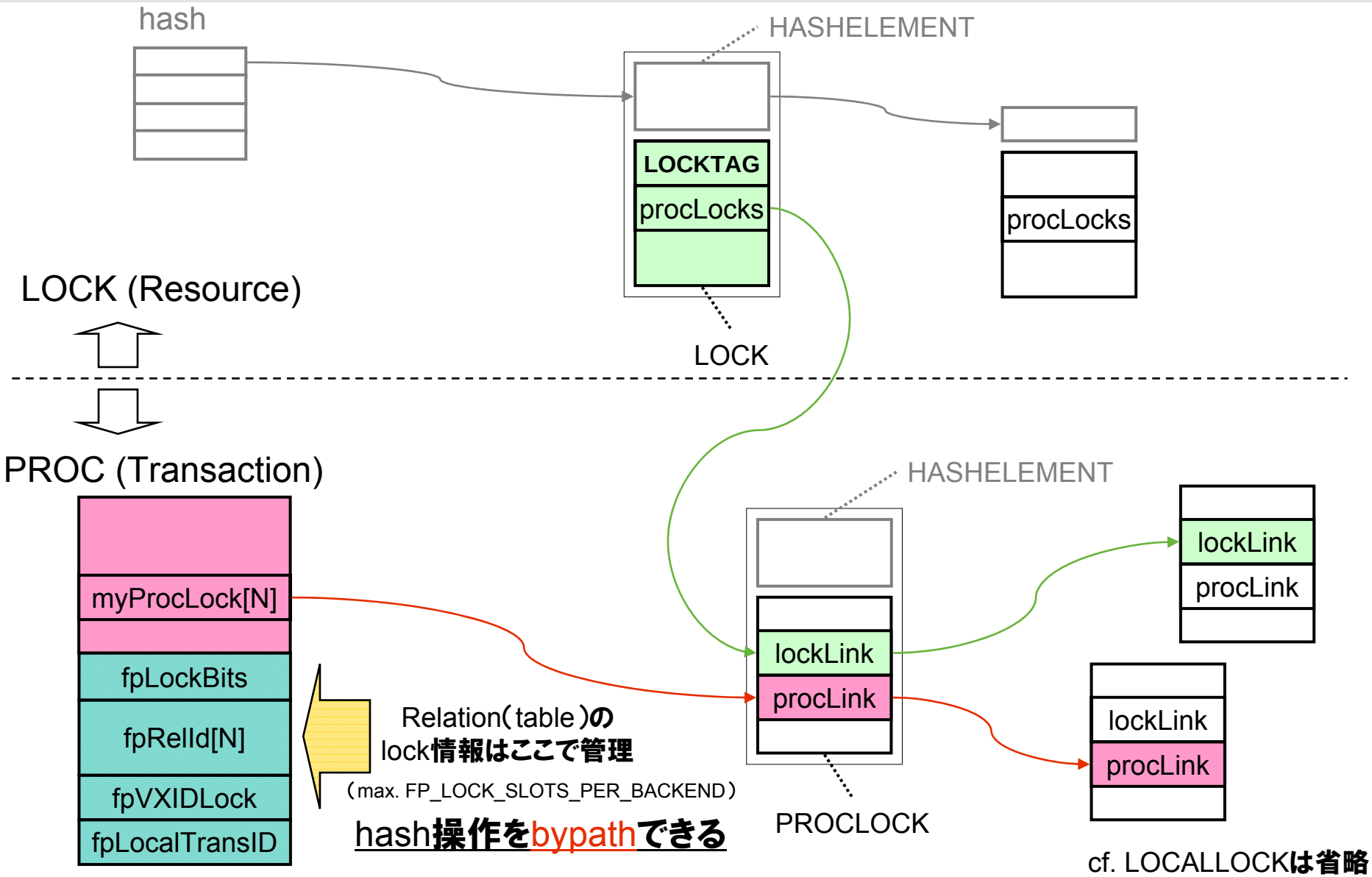
Empowered by Innovation NEC

その他，9.2の（性能）強化点
(from code reading)

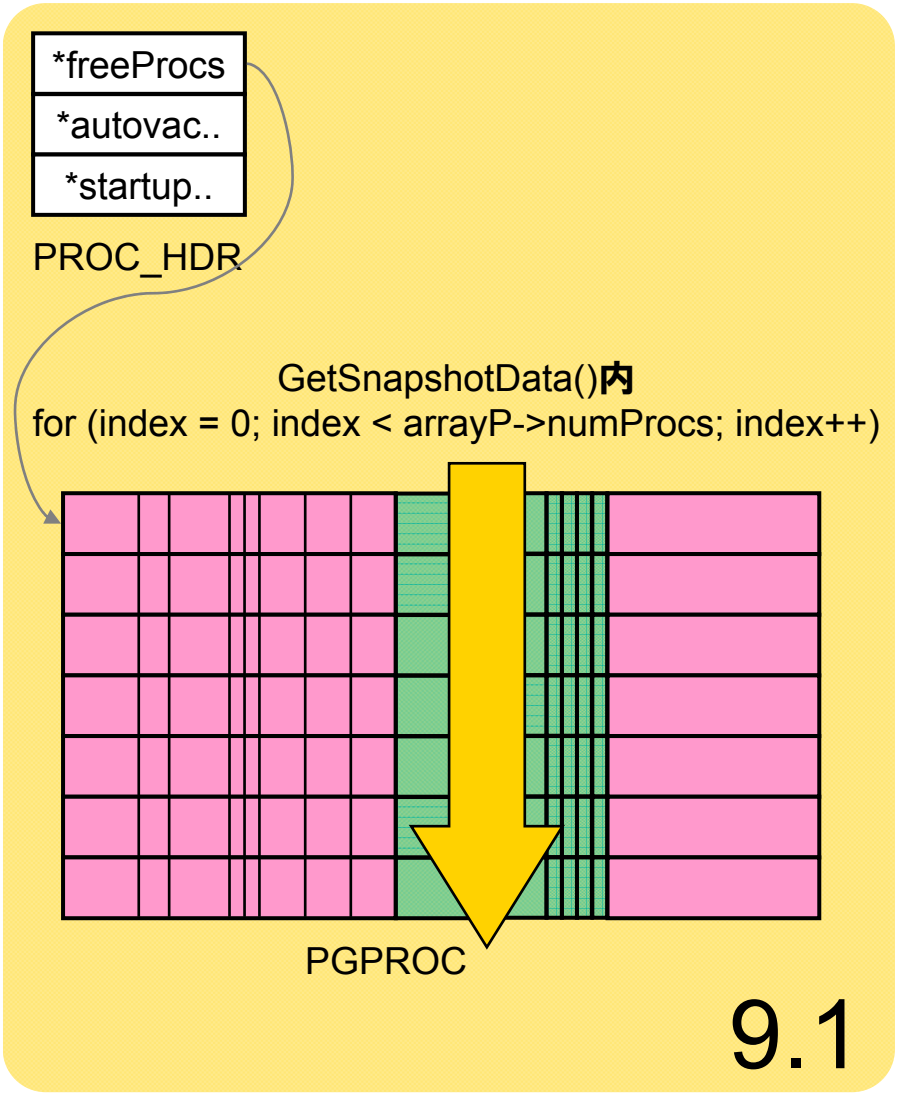
Fast Path Lock (before)

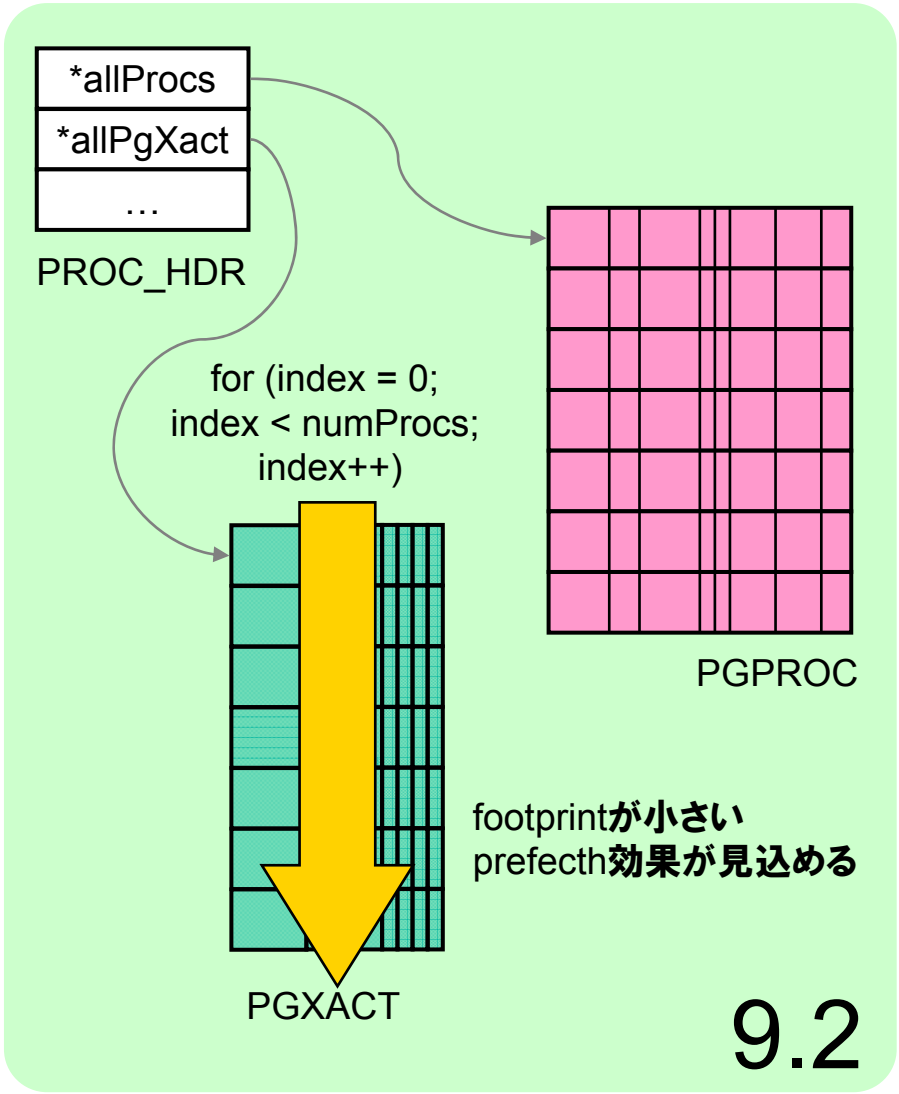


Fast Path Lock (after)



snapshot 取得処理





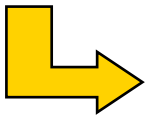
⇒ 考え方はColumn DBに似ている

まとめ

pgbenchだとELRの効果はある

- branch表のレコード更新における競合がボトルネックの源である可能性大
  - branch表のレコード数を10倍 ⇒ ボトルネック緩和（性能向上）
- WAL writeにおける “不必要なsleep” 解消の方が効果は大
  - DLKMで補助する方法
  - LWLock + XLogFlush() の改造 （9.2で導入された方式をback port）

} 同程度の効果



とりあえずは，これで充分そう

9.2（その他の）強化点

- Fast Path Lock（select onlyで大きな効果）
  - 但し，副作用は？ ⇨ lockのactivityを止める際，9.1系とはtrivialな差異がある
    cf. Q&A @ U-stream録画
- Snapshot 取得処理でのキャッシュ・ミス低減
  - 考え方はColumn DBに似ている

Empowered by Innovation

Empowered by Innovation NEC

付録.　ELRの実装方法

1. RecordTransactionCommit()をインライン展開(手作業)

```
static TransactionId
RecordTransactionCommit(void)
{
        TransactionId latestXid = InvalidTransactionId;
. . .

cleanup:
        /* Clean up local data */
        if (rels)
                pfree(rels);

        return latestXid;
}
```

```
static void
CommitTransaction(void)
{
        TransactionId latestXid;
. . .

 latestXid = RecordTransactionCommit();
. . .
}
```

cf. RecordTransactionCommit()を callしているのはここだけ.

展開後

```
static void
CommitTransaction(void)
{
        TransactionId latestXid;
. . .
```

同じ変数名なので，外側のものを使う

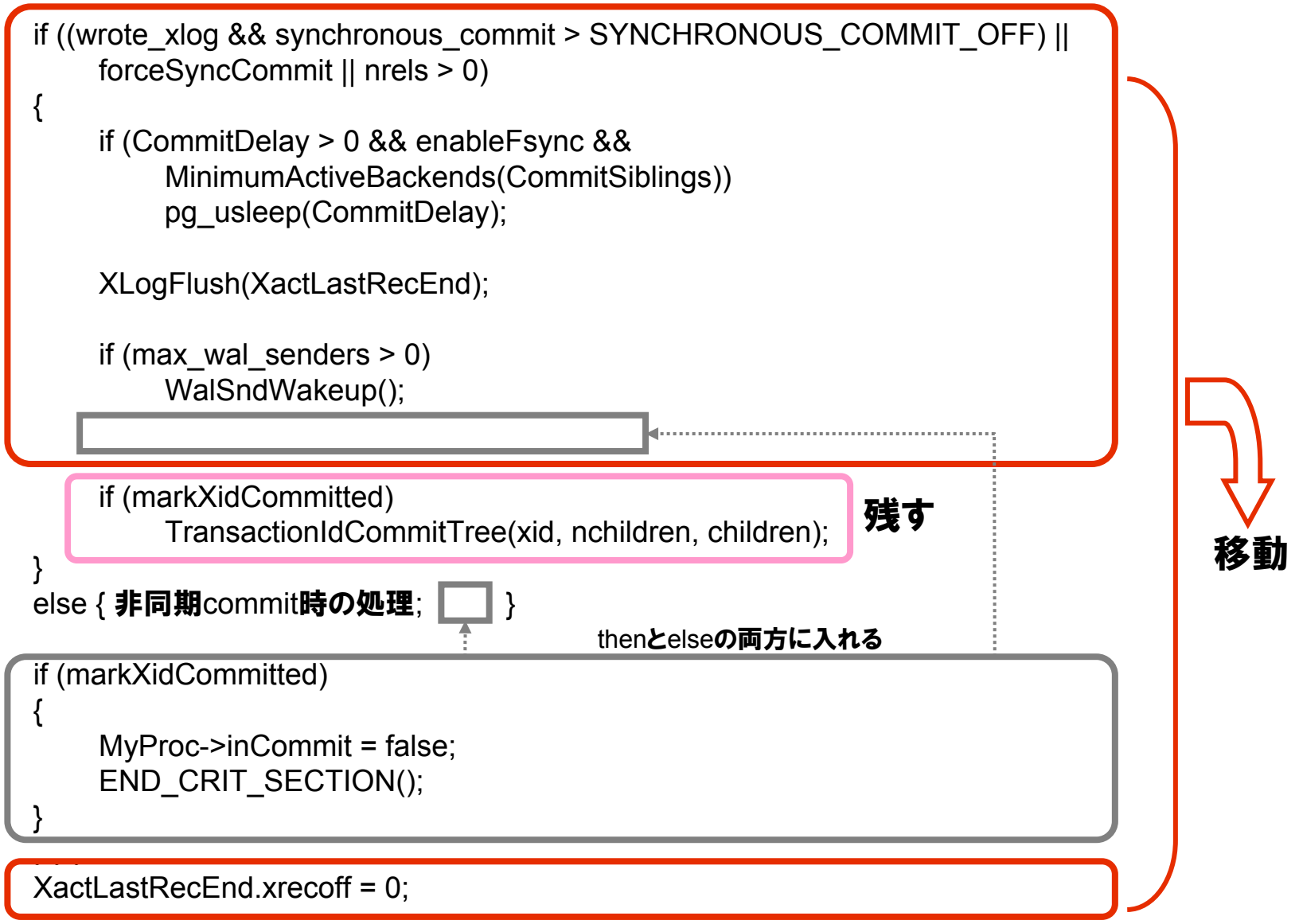
```
{
    TransactionId latestXid = InvalidTransactionId;
. . .

cleanup:
        /* Clean up local data */
        if (rels)
                pfree(rels);

    return latestXid;
}
    latestXid = RecordTransactionCommit();
. . .
}
```

結局，retrun valueは同じ変数名に入る

2. XLogFlush()の呼び出しをCommitTransaction()の最後に移動



(完了) 移動後の状態

```
if ((wrote_xlog && synchronous_commit > SYNCHRONOUS_COMMIT_OFF) ||
    forceSyncCommit || nrels > 0)
{
    if (markXidCommitted)
        TransactionIdCommitTree(xid, nchildren, children);
}
else
{
    XLogSetAsyncXactLSN(XactLastRecEnd);

    if (markXidCommitted)
        TransactionIdAsyncCommitTree(xid, nchildren, children, XactLastRecEnd);

    if (markXidCommitted)
    {
        MyProc->inCommit = false;
        END_CRIT_SECTION();
    }
}
```

残した処理

if の外にあった処理

非同期commit時の処理

(完了) CommitTransaction()の最後 ← XLogFlush() の移動先

```
if ((wrote_xlog && synchronous_commit > SYNCHRONOUS_COMMIT_OFF) ||
        forceSyncCommit || nrels > 0)
{
        if (CommitDelay > 0 && enableFsync &&
                MinimumActiveBackends(CommitSiblings))
                pg_usleep(CommitDelay);

        XLogFlush(XactLastRecEnd);

        if (max_wal_senders > 0)
                WalSndWakeup();

        if (markXidCommitted)
        {
                MyProc->inCommit = false;
                END_CRIT_SECTION();
        }
}

XactLastRecEnd.xrecoff = 0;
```

ifの外にあった処理

移動したXLogFlush()のcall処理

```
RESUME_INTERRUPTS();
}
```

CommtiTransaction()の最後にあった処理