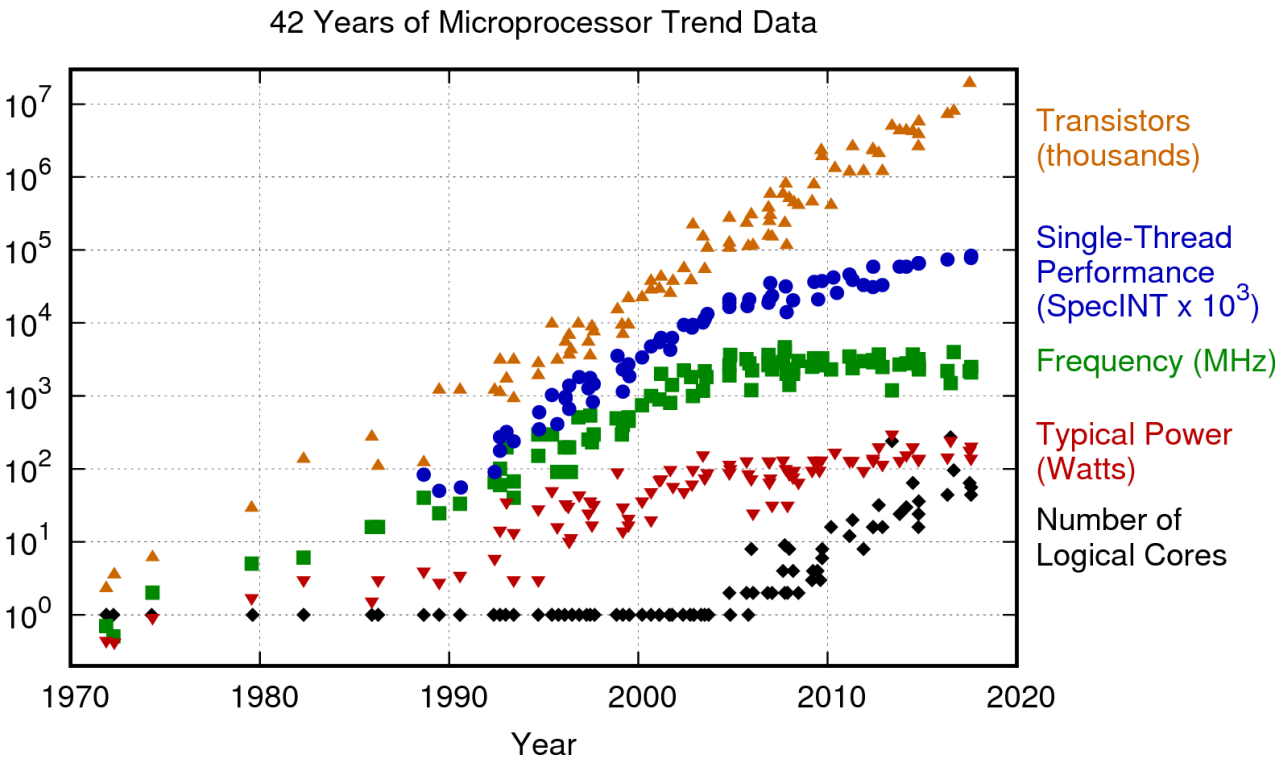
大規模分散OLTP

次世代DB / 分散OLTP

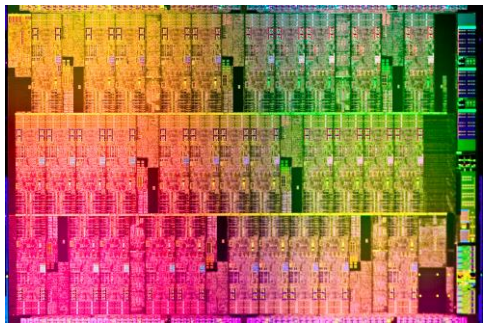
NAUTILUS

背景:ムーアの法則の終焉とメニーコア化の進展

- ムーアの法則の終焉によりプロセス微細化は限界にあたりつつある。
  - クロック周波数の向上は頭打ちになっている
- この結果として半導体業界はCPUのメニーコア化を進めつつある。



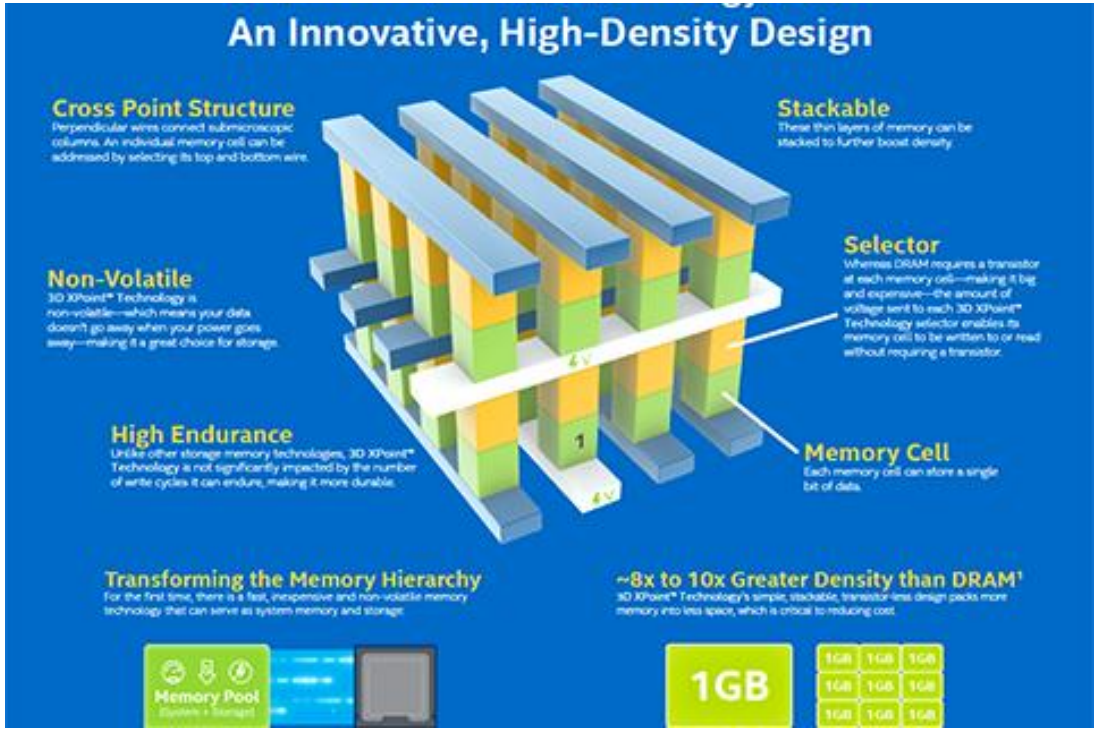
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp



thousand –core machineの登場

背景:メモリーデバイスの進化

- メモリーの高機能化と高密度化
- 不揮発性メモリーの発展が進んでおり、NVM/SCM NVDIMM等の高速の不揮発性メモリーも登場し、市場に投入されつつある。
- 同時にメモリーの高密度化も進みつつあり、サーバあたりのメモリー容量もTByteクラスまで伸張しつつある。



出典:2017/7/28Intel press release

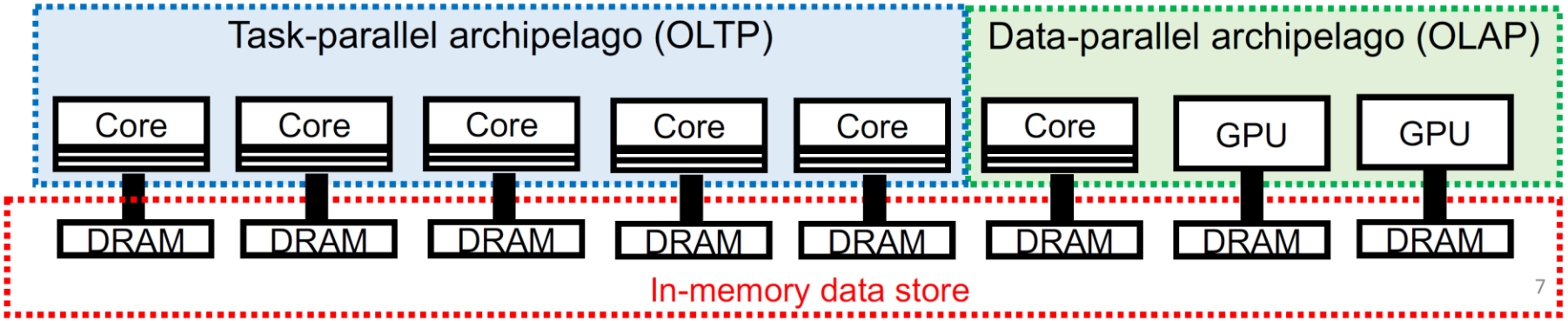
背景：OLAPとOLTPの統合

NAUTILUS

- OLAPとOLTPの統合
  - ビックデータ・IoT・AIの伸長は特にOLAPの需要を大幅に伸ばした
    - OLAPマーケットも大幅に伸長し、プロダクトも成長しつつある
  - 他方、現状のOLAPでは、サニタイズして正規化するために**OLTP系でメンテナンスされているデータとのETL処理**を行う必要がある
    - →コストがかかる
  - 現状ではOLTP系の処理とOLAP系の処理系は分離しており、その統合が急務である
  - これらの統合は、OLTP系またはOLAP系から個別に進化させていくには、その最適化ゆえに根本のアーキテクチャ変更を行う必要があり、現実的ではなく、新たに新しいアーキテクチャの登場が待望されている。

HTAP

- OLTPとOLAPの共存



- データ同期の方法
  - 更新処理とそのプロパゲーション
- 整合性の確保
  - Snapshot Isolationの問題点の顕在化
- タスクの分散並列実行
  - リソース管理の仕組み

出典：The Case for Heterogeneous HTAP
Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki
Data-Intensive Applications and Systems Lab, EPFL

DBの現状課題

- アーキテクチャ的な限界～既存の環境の変化
- DBをめぐる環境の変化

- CPUアーキテクチャの変化と対応
  - メニーコア化の進展
  - 同時に多数のコアを利用することが必要になっている。
    - 従来のDBは少数のコアの利用を前提している。この仕組みをスケールアウトさせることは困難であり、設計思想の最初からメニーコアの仕組みを前提にしたアーキテクチャが必要になっている。

- サーバ・アーキテクチャの変遷
  - 超高速インターコネクトの利用
  - コア・ローカルメモリーとソケットを越えたリモート・メモリーまた、ノードを越えた他ノードのメモリーを管理する必要がある。
  - Non-Uniformなメモリー上でのデータのロケーション管理が必要になり、ソフトウェアレベルでのデータのConsistencyの管理が必要になる。
    - 既存DBはNUMA前提ではない。

DBの現状課題

- 不揮発性メモリー
  - メモリーの大容量化と不揮発性メモリーの利用により、Redo logのみによるDBのリカバリーが可能になった。Undo logのdiskへの書き込みは不要となり、ARIESの実装が事実上、必要なくなった。

- メモリー単価の下落
  - メモリー単価の下落は、メモリーモジュールの大容量化・サーバあたりのメモリーの大容量化に進んでいる。利用できるメモリー量が著しく増加しており、DB上でも様々な仕組みを利用できるようになっている。
    - 従来のようなsingle-versionまたは 2-versionのモデルではなく、multi-versionの仕組みも利用することが可能になり、serializabilityの確保可能性が向上している。
    - replicationを効果的に利用することが可能になり、DBのパフォーマンスや可用性を大幅に向上させることが可能になってきている。

- バスの高速化
  - interconnectの高速化・バンド幅の大幅な向上が行われつつある。コアローカルのメモリー間・ソケット間でのメモリー間転送・ノード間の通信といったデータの転送・参照・コピーをそれぞれのレイテンシー・スループットに対応させ、最適なパフォーマンスを引き出す必要がある。

- これらのメモリー対応・バス対応を行うためには従来のDBのアーキテクチャでは根本のところから作り替える必要がある

現在のDBの性能・機能限界

- RDBMSの性能の問題
  - 従来のRDBMSは、ハードウェアはコア数は少なく、メモリー容量は制限的という前提を敷いており、現在に至るまで基本アーキテクチャの思想は変わっていない。
  - In placeまたは2-versionまでのページモデルを前提とし、リカバリーはARIESをベースにしている。結果として、ハードウェアの性能が高進しても、それを生かし切れずスケールアップに限界がきている。

- NoSQLの機能の問題
  - 圧倒的なハードウェアの豊富さ・容量を生かすスケールアウトを目的としているNoSQL系の実装は、スケールアウト確保のトレードオフとして一貫性の担保を放棄している。すなわちトランザクション機能を実装していない。
  - しかし、ユーザレベルから見ると、一貫性担保は必要な機能であり、結果、ユーザはアプリケーションの下位レイヤーに必要な一貫性担保の仕組みを実装している。
    - RAMPSに見るように、これらの実装は汎用性に欠ける。
    - ACIDRain (http://www.bailis.org/papers/acidrain-sigmod2017.pdf)
  - また、結果としてアプリケーション構築負荷を上昇させている。このような現状ではNoSQL系の実装を、厳格な業務システムに適用することには限界がある。

DBの提供環境の問題

- 商用DBの減少
- 現状の商用DBはベンダーの統合の結果、その数が減少し、企業システムにおいてはほぼOracle社/MS社の寡占状態にあるといってよく、ユーザの選択の幅は極めて狭い。
  - Oracle社は今後のビジネスをクラウドに移行する意図を鮮明しており、メニーコア環境下のオンプレミス用ライセンスは制限的に提供される可能性もある。

- OSS-DBの課題
- 商用DBの代替と目されているOSS系は、近年のOSSの発展の結果として、既存のアーキテクチャでは十分なパフォーマンスを出すに至った。しかし、その一方でコードベースの増大・ステークホルダーの増加を招き、抜本的なアーキテクチャの変更は難しい。実装変更の合意が取りにくく、環境の大幅な変更に追随できなくなっている。
- 大幅なアーキテクチャの変更はスクラッチからの構築になる。特に、今後は複数の手法を試行錯誤する必要があり、その環境を提供できない。

トランザクションの進化：OCC vs MVCC

- Tx処理の二方式の争いが現在地
  - OCC :Optimistic Concurrency Control
  - MVCC :Multi-Version Concurrency Control

- OCC（楽観処理）
  - **Single version**
  - シンプル・イズ・ベスト
    - SILOベース
    - abort前提
    - エンジニアリングがやりやすい

- MVCC
  - **Multi version**
    - MVTOベース
    - Conflictをそもそも減らす＝abortは避ける
    - 理論先行だったがエンジニアリングが追随開始

OCC:これもSILOベースにかわりつつある

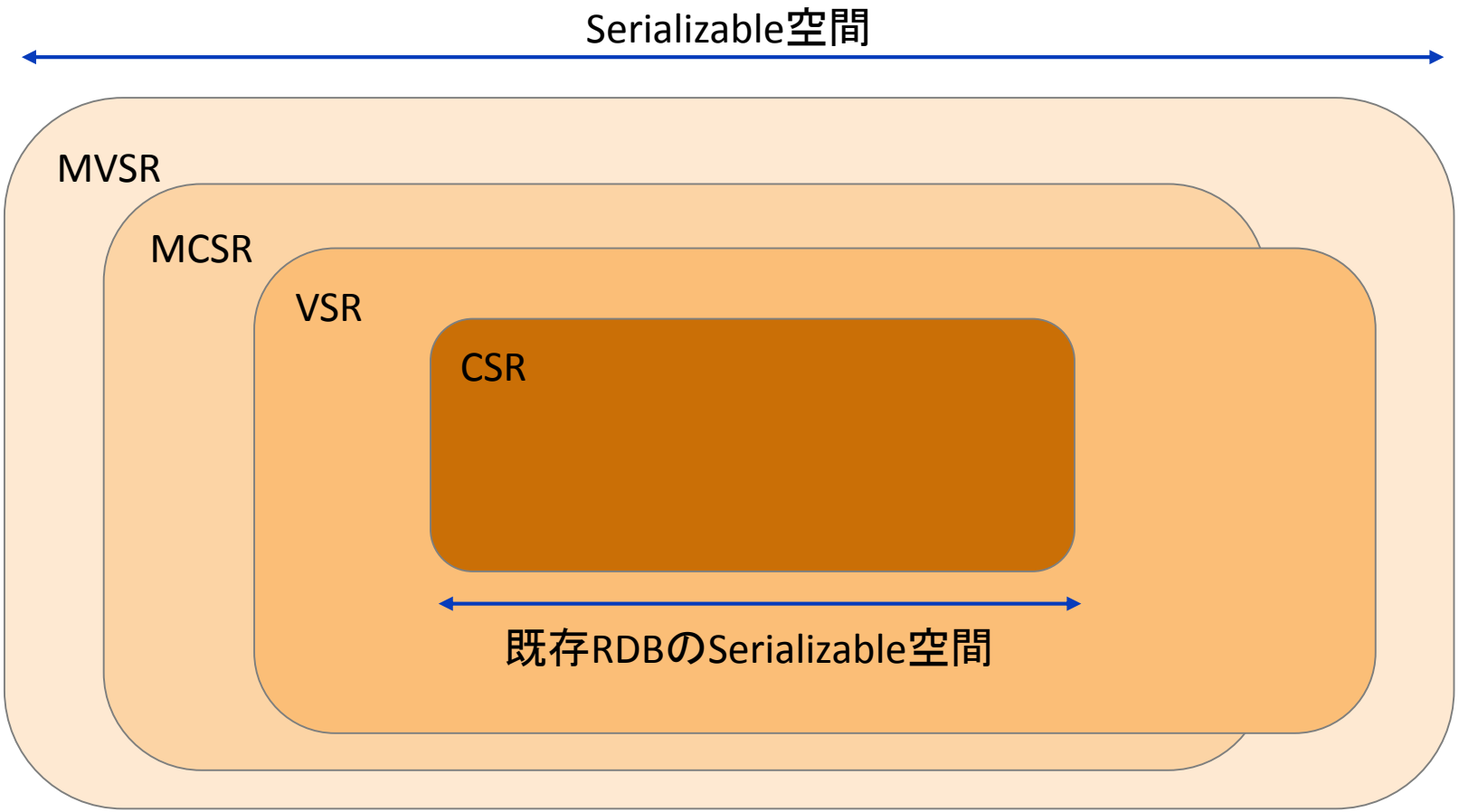
- 基本的な仕組み
  - 3Phaseアプローチ
    - Read phase
      - 共有からローカルに引っ張る（コピーかどうかは微妙）
      - 特にmany-core/in memoryでは、uncommittedではあってもlocalに値がcacheされるので、cache missが減る。
    - Validation phase
      - Consistency check
      - serializableかどうかを確認する。できなければabort
      - 要するにread lockは一切取らず（楽観）にwriteで可能かどうか問い合わせる
        - » よってabortになることがある
    - Write phase
      - コミットを行って書き込む
    - (*)Global phase
      - Snapshotting
  - Single version
    - In-place
    - 1-version-in-placeはオーバーヘッドが少ないので、特に競合がない状態では高いパフォーマンスを出す
    - 極めて軽い。GCが楽
    - Read用にはSnapshotを準備

OCCのserialzation空間

- SILO-2PL
  - 基本CSR空間の内部
    - 2PL方式だが、read-lockはとらない
    - Writeロックをとる
    - 言うまでもないがserializable
      - 注意：今後のDBはisolation levelはすべてserializableがデフォルトの世界
  - Read lock free
    - Optimistic処理
    - とりあえず読ませる
    - コミット時点で更新の有無を確認
      - 確認はTimestampを見る
      - 変わっていれば、誰が更新したので、要はlock失敗と同じ。よってabort
    - Writeは普通にlock処理
  - Validation phaseで多少時間をとるが、基本的にreadはwriteにブロックされない。
    - 当然concurrencyも上がる。

serializable

- Serializableは前提
  - その上でどのレベルのserializableなのか、というserializableの中での戦い



今後について、Serialization空間（理論）

- 「広ければ広いほどよい」
  - Abort率が減る
    - 決定的にこの差は大きい。
    - ただしワークロードによる。

- MVSR
  - Multiversion view serializability
    - 理論的最強
      - FSRはinconsistency readが起きるので、枠としてはMVSR以上の枠は存在しない
      - 一般解はNP完全
- MCSR
  - Multiversion conflict serializability
    - MVSRの次に最強
      - MVSRからverison orderに制約をつけて、解決案を簡単にした
        - » それでも一般解はNP完全
        - » 実装はこれに近づける（MVTO）
- VSR
  - View serializability
    - monoversionでの最強
    - NP完全
- CSR
  - Conflict serializability
    - Monoversionでの通常のserializability
    - 普通のRDBが言ってる空間

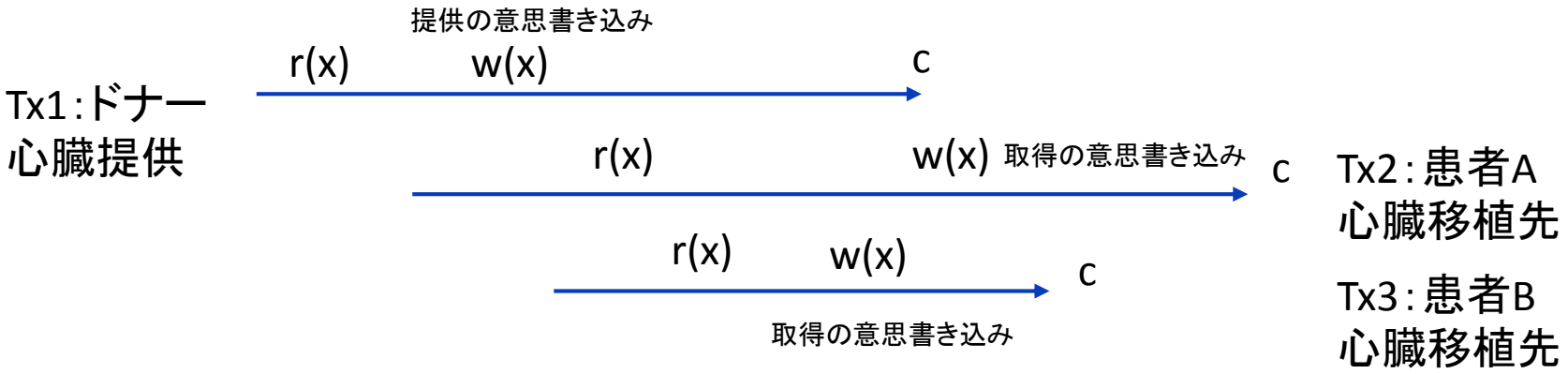
Serialization空間（実際）

NAUTILUS

- 実際は理論上の空間とは別になることが多い
  - 実装上のプロコトルでのSerialization空間
    - MVSRやMCSRがNP完全なので、どうしても実装するにはある程度制約を設けてやらないと事実上使い物にならない
    - 例）MVTO(mulitiversion time ordering)
      - Multiversionでの実装プロトコル（あとで解説）
      - MCSRに準じるがそこまで行っていない
      - CSR<MVTO＜MCSR
  - なにが面倒かというと・・・
    - 大抵の論文・実装は
      - 「serializabilityの証明」をやるだけ
        - » これはこれで面倒
      - **どの程度のserialization空間か？自分で判断する必要がある**
    - 判断が困難
      - しかも実装プロトコルをほぼ完全に理解して、その上での制約を判断して、どのレベルか考慮する必要がある
      - そうでないと実際の利用ケースで「なにがabortされるのかわからない」
      - なので Cicada-MVTO とかHekaton-OCCとかぞれぞれ独自の空間をもつ

そんなにserializationは大事か？

- 問題
  - 例えば、あなたがSEで心臓移植のドナーと患者つなぐシステムのDBAでした。
  - 仮に、以下の取引があったときに誰がその心臓をゲットするか答えよ
    - Isolation levelは当然serializable
    - 間違った時のコストは人の命



- Tx orderはtx2→tx3：先に取引を始めたのはAさん
- Commit orderはtx3→tx2：先に最終意思決定したのはBさん
- リードはtx2→tx3：先に見つけたのはAさん
- ライトはtx3→tx2：先にとりあえず手を上げたのはBさん

OCCの弱点

- Abortが多い
  - 現在のMVCC/OCCだとスループットが1Mtpsぐらいでるので、abortのペナルティが大きい
    - cacheが汚れる
  - 1-versionなので、conflictが加速度的に増える
    - Abort→リトライが輻輳してくるとますます詰まる
    - そもそもオーバーヘッドが少ないのでどんどんリトライできることが裏目
  - 一部 read-only snapshotで軽減する部分もあるが、抜本的な解決にはならない

- Extra read
  - ロックがないので、read時点にガンガンwriteが走る
  - 何にもしてないとパーシャルライトを拾ったりとか、repeat readとかで爆死とかいろいろ面倒になる
  - なので、一回ローカルにコピーする
    - リードセットがデカイとかなりコスト高め

OCCの弱点

- Indexの競合
  - 普通はwrite時点でindexは更新して、コミット時点まで他には見せない
    - キー制約の維持
    - Abort時点でいろいろ面倒になる
  - 一応、いろいろ工夫しているOCCも多い
    - **いずれにしろabortが多いということは不利になる要素を抱え込む**

- 基本的にOCCの弱点は1-versionであること
  - ただし、これはコインの裏表
    - 軽く・素早くがベース
    - そのためabort→retryが詰まり始めると窒息する
  - abort軽減は1-versionではどうひっくりかえっても上限はある

MVCC

- Multiversion～基本的な仕組み
  - 複数のversionを利用してコンフリクトを回避する。
    - s = r1(x)w1(x)r2(x)w2(y)r1(y)w1(z)c1c2 : single-version
    - s = r1(x0)w1(x1)r2(x1)w2(y2)r1(y2)w1(z0)c1c2 : multi-version
  - 仮に更新があったとしても、前のversionを利用したtxが可能。
    - **原理的にw-w競合はおきない**
    - w1(x1)w2(x2):x1,x2は別
    - w1(x)w2(x):single versionでは競合
  - version の管理はtimestamp(以下ts)を利用して行う。
    - tsのアサインはtxの開始時点で行われる。
  - tsを利用してvisibleな利用可能なversionを特定する。
    - w1(x1)w2(x2)c2w3(x3)r4(x?)c3c1c4
  - versionのtsはtxの開始時点のものか、またはcommit時点のものか、どちらかを利用する。
  - 上記はほぼすべてのMVCCで共通

MVCCの弱点

- 基本的にMVのオーバーヘッド。
  - これが重い
  - 歴史的にはMVの理論的優位性は1980年代から証明されていたが、テクノロジーがそれを捌くだけのパフォーマンスが出せなかった
    - 昔の（特に不勉強な）おっさんがMVCCはだめだ、と却下するのはそういう理由
  - この数年でハードが追いついてきた。ちょっと想像を超えるぐらい。

- サーチとストレージのオーバーヘッド
  - 特にIn memoryな高スループットな環境では、version searchとstoreのコストはCPUを食う。
  - storeの空間コストも大きい。
  - 大抵のMVCCではlistや配列の中のversionのsearchに間接参照利用する。これはキャッシュミスやワーキングセットがCPUキャッシュに乗らない場合は特にハイコストになる。
  - 最近のMVCCでは最後のversionでは間接参照を利用せずにin placeでの処理を行う方式もあるが、これは1-VCCと同じくextra readの問題を引き起こす。

MVCCの弱点

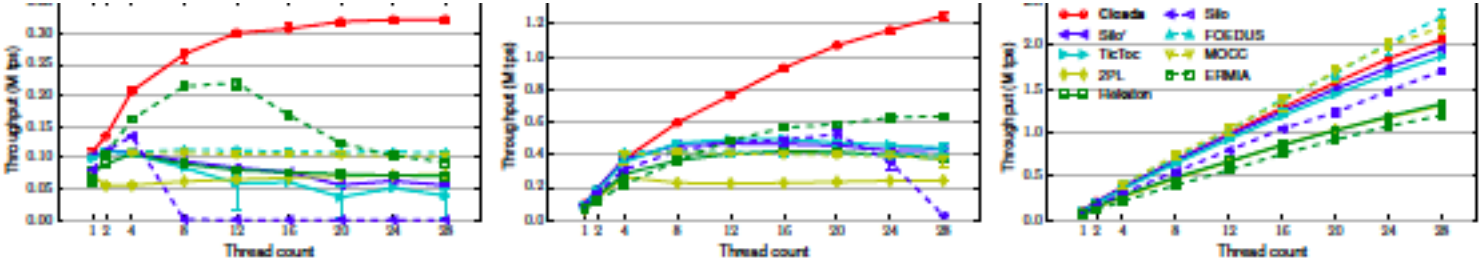
- Footprintがデカくなる
  - multiversionなのでfootprintが大きくなる。
  - ワーキングセットが大きければキャッシュヒット率は下がるし、処理のパフォーマンスも落ちる。
  - 頻繁にGCすることでfootprintを小さくすることもできるが、効率よくやる必要がある。

- Writes to the shared memory
  - 大抵のMVCCでは共有メモリーに書き込むが、これはメニーコア環境では悪手。

- Timestamp allocation のボトルネック
  - tsの発行に、centralizedな方式で、atomicにshared counterを増やす形をとるとワークロードの競合状態に関係なくパフォーマンスに制約を発生させる。
  - 1-VCCよりも桁違いに悪くなる。今後のメニーコア環境ではますます悪化する。

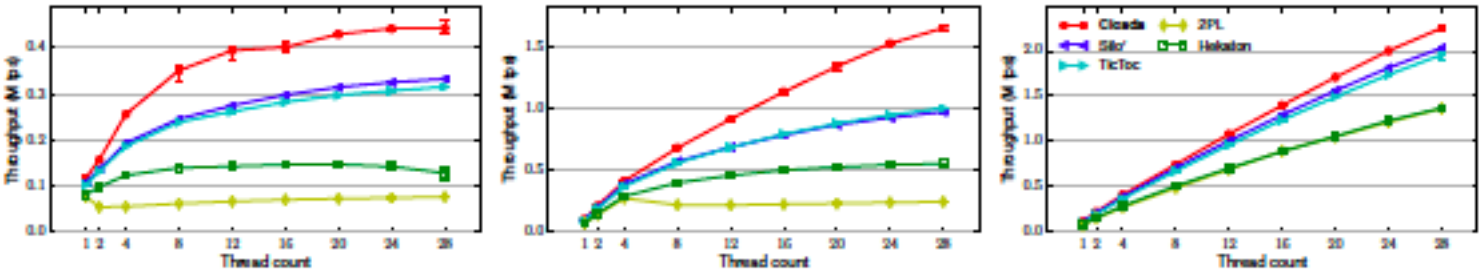
MVCC（Cicada）の圧倒的なパフォーマンス

出典：
Cicada: Dependably Fast Multi-Core In-Memory Transactions
Hyeontaek Lim Carnegie Mellon University hl@cs.cmu.edu
Michael Kaminsky Intel Labs michael.e.kaminsky@intel.com
David G. Andersen Carnegie Mellon University dga@cs.cmu.edu



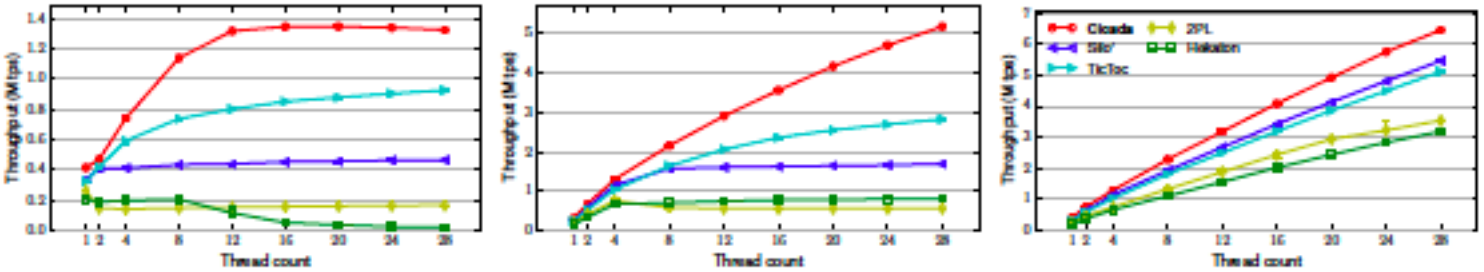
(a) 1 warehouse. (b) 4 warehouses. (c) Warehouse count = thread count.

Figure 3: TPC-C (full mix) throughput; with phantom avoidance.



(a) 1 warehouse. (b) 4 warehouses. (c) Warehouse count = thread count.

Figure 4: TPC-C (full mix) throughput; with deferred index updates and no phantom avoidance. DBx1000-compatible schemes only.

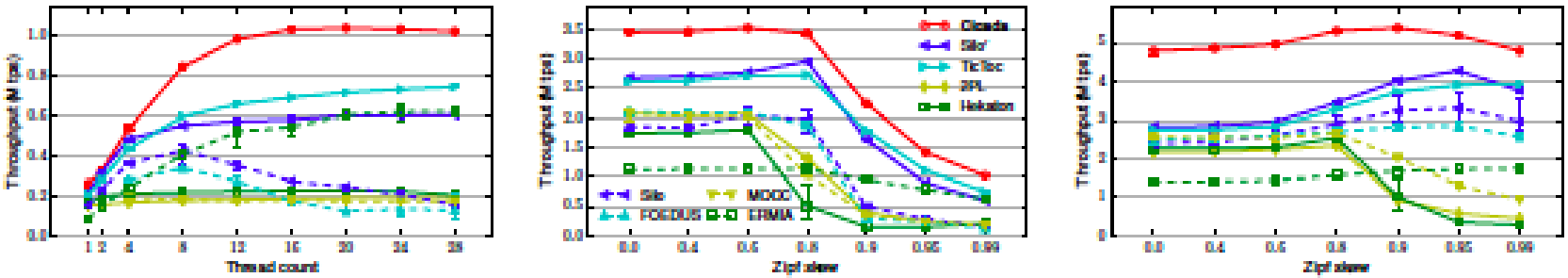


(a) 1 warehouse. (b) 4 warehouses. (c) Warehouse count = thread count.

今時、これだけワンサイドゲームはあんまりない。多分対抗できているのはMOCC/Foedusだけ

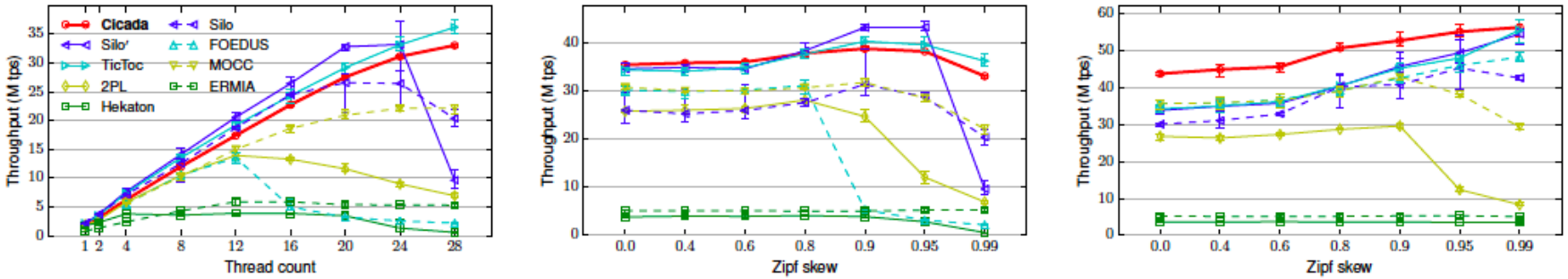
YCSB～やっぱり強い

出典:
Cicada: Dependably Fast Multi-Core In-Memory Transactions
Hyeontaek Lim Carnegie Mellon University hl@cs.cmu.edu
Michael Kaminsky Intel Labs michael.e.kaminsky@intel.com
David G. Andersen Carnegie Mellon University dga@cs.cmu.edu



(a) Write-intensive, skew of 0.99. (b) Write-intensive, 28 threads. (c) Read-intensive, 28 threads.

Figure 6: YCSB throughput using 16 requests per transaction.



(a) Write-intensive, skew of 0.99. (b) Write-intensive, 28 threads. (c) Read-intensive, 28 threads.

Figure 11: YCSB throughput using 1 request per transaction. Included as part of Appendix B.

新世代DB：国内Projectとして立ち上がりつつある

- 提供されるもの
- 課題を解決すべく以下のOSSプロダクト・環境を立ち上げるProjectがローンチされている

- 機能要件：現在のRDBMSのリプレースを目標とする。
  - SQLの実行
    - 既存OSSでサポートされている共通レベルでのSQLのサポート
    - ストアド・プロシージャに代表されるバッチ実行基盤
  - トランザクションの保証
    - Isolationレベルとしてserializableのみをサポート
  - 高速クエリー処理
    - ストアド・プロシージャ等のバッチ処理高速化のサポート
  - pluggableなアーキテクチャの採用
    - フロントエンドとバックエンドをそれぞれ分離する
      - 可能であればheterogeneousな環境をサポートする
    - 複数のデータベースエンジンを採用できるアーキテクチャとする。
  - Customizable Database
    - Pluggableなアーキテクチャを採用することで、各ユーザのアプリケーションワークロードに最適なDBが構成可能になることを目指す

目標Project

- Pluggabilityについて
  - Pluggableにする目的
    - 様々なワークロードに適応させるため
      - DBにおいてはone-doesn't-fit-allは明確
      - 各ワークロードに応じた実行エンジンが選択可能である方が望ましい
    - 研究開発を行いやすくする
      - DBのエンジン開発は日進月歩
      - 製品化時にはすでに次世代が出ている
      - OSSのメリットを生かして新しいテクノロジーを採用しやすくする
    - アカデミアの成果を積極的に取り込めようにする
      - アカデミアのDB研究のプラットフォームとしても提供する
      - アカデミアはDB上で様々モジュール試験的に実装することが容易になる
      - 結果として産学協同の体制が可能になる
    - プロプライエタリの製品に対抗できる手段を提供する
      - プロプライエタリの製品はその性格上「one-fits-all」を目標とする
      - したがって「特定のワークロード」に最適なエンジンを選択的に利用することでプロプライエタリの製品をパフォーマンスで凌駕することが可能になる

- Pluggabilityについて
  - 基本的にビルド時点で実装を選択できる形式を目標とする
  - ただし、将来的にはruntimeでの実装選択が可能であることも目標とする
    - 例）Read-mostly-Tx実行時のread-setのts処理の選択適用等

非機能要件

- パフォーマンス要件
  - メニーコア・大容量メモリー・不揮発性メモリー環境でのハイパオフォーマンスなDBを目標とする。
- 目標TPS
  - メニーコア・大容量メモリー・不揮発性メモリー環境で1ノードスループットとしてTPC-Cで10Mtpsを目標とする。
  - バッチ処理での適切な目標設定を行う
- 可用性担保
  - DBとしての障害対策を提供する。
  - 高速リカバリーを提供する。
  - 現状の商用DB同等のHAを提供する。
  - ハードウェア障害に対して適切な復旧手段を提供する

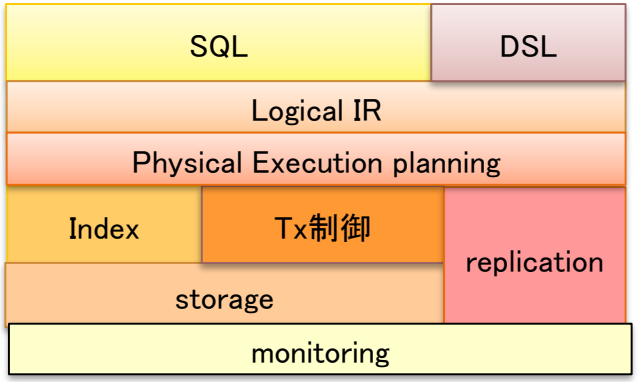
非機能要件

- 運用要件
  - Monitoring
    - モニタリングについては２層でのログ情報を提供する。
      - アプリケーションレイヤーでの通常の運用監視用ログ
      - developer用・チューニング用のログ
        - » アプリケーション用ログについてはサービサーでプラグインできるような要件とする。
    - 統計情報の提供
      - DBの利用状況についての統計情報を提供する
  - Backup
    - オンラインバックアップが可能とする。
  - DR対応が可能なインターフェイスを提供する。
    - replication環境の提供

アーキテクチャ

- 論理構成
- ユーザ・インターフェイス
  - SQL：標準的なSQLをサポート
  - IR（中間表現）の公開：SQLを中間言語にコンパイルする。
  - バッチ処理用言語サポート（DSL）
    - 利用可能言語としてはPython/Javaを想定
    - ストプロ/PLSQL的な実行は、独自のDSLで記述し、IRにコンパイルする。

- 実行計画策定
  - IRから最適化の実行
  - 物理実行計画を策定



- Tx実行制御
  - 物理実行計画を実行
  - commitプロトコルの実装
  - 複数方式の選択実装ができるものとする。
  - OCC/Validation方式　Multiversion/Certifier方式

アーキテクチャ

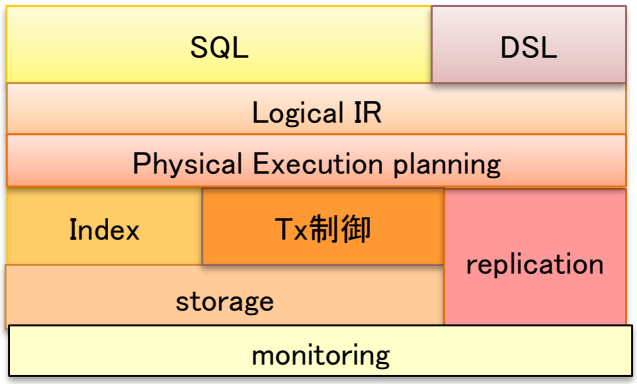
- Pluggable
  - 可能な限り各モジュールは選択実装が可能であるようにする
    - 各モジュール間のAPIを公開し、再構成可能なアーキテクチャとする
    - 実装選択を行うことで、個別のアプリケーションのワークロードにそれぞれ最適な実装を持つDatabaseにすることが可能
  - 選択実装の例
    - 特定ハードウェア依存
      - NVM・NVRAM・SSD
    - ワークロードに合致するTransactionManager
      - MVCC/OCC/MVOCC
    - 特定環境に合致するDSL Complier
    - Replication方式
    - Log方式
      - SILO/WBL
  - まだ必要に応じて、このエンジンを他のOSS-DBへ提供することも可能となる

アーキテクチャ

- ログ制御
  - 構成としてはlog管理/キャッシュ-メモリマネジメントからなる。
  - Logマネージャー/Bufferマネージャー/ロックマネージャー
    - GC
    - Tree構造（メモリーベース）

- Index制御

- ストレージ
  - エンジンの選択可能性の確保
  - Log storage
  - NVM/SSDの利用インターフェイス

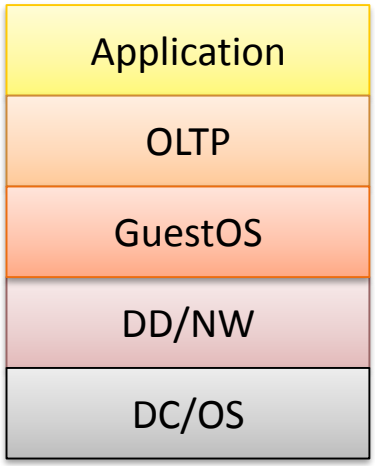


- Replication
  - ノード間通信のサポート
  - LogShipping
  - SnapshotベースでのIsolation（read only）提供インターフェイス

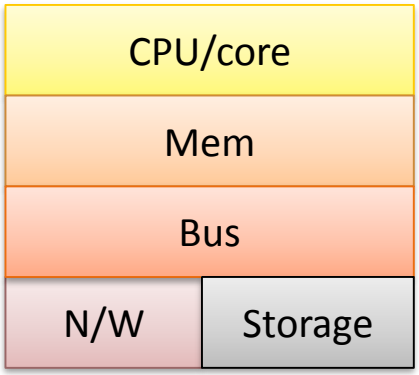
- モニタリング
  - 実行ログの生成・取得

構成スタック

- ソフトウェア物理構成
  - アプリケーションプログラム
  - OLTP:提供対象
  - OS
  - デバイス制御/NW接続
  - Container/VM/ホストOS



- ハードウェア物理構成
  - CPU・コア
    - メニーコアを想定
  - キャッシュ/メモリー
    - NUMA
    - NVRAM
  - 高速Interconnect
    - ストレージ
    - N/W



提供方法

- OSSとして提供される。
- ライセンスについてはdistributorに特許制約がかからないものを選択する。
- Distributionは各distributorから提供される。
- QA等はOSSコニュニティで提供
  - ただし一般的なOSSコミュニティのプロトコルに沿うものとし、そうでないQAは各distributorが有償でサポートするものとする。
- 基準ベンチマークの提供
  - このプロジェクトでは、「実際の」企業利用に供するために適切なベンチマーク使用・データセットを作成・提供することも企図している。
  - この提供されたベンチマークで、既存DBでのスループットの向上を目指す
  - 当該DBの構成をPluggableにすることにより、実装選択が可能となる。実装評価基準として基準ベンチマークを位置づける

プレーヤー（体制）

- OSS実装者
  - 設計/実装/CI等の管理
  - ドキュメンテーション
  - 処理方式の改良検討等
  - ベンチマーキング

- ユーザ
  - PoCの実施
  - 開発に必要なワークロードの原案の提供
  - ベンチマークの基礎資料作成のサポート

- Distributor
  - 独自distributionの作成・販売・サポート

- クラウドサービサー
  - OSSを利用したクラウドサービス化の提供
  - アカデミア等への開発環境の提供

アカデミア連携

- Projectとしては、各大学・専門機関の協力を仰ぐ。
- OSSの実装・環境を提供することで、各研究機関で最適なプロトコル・アーキテクチャを試行錯誤・サンプル実装を行うことを要請する。結果を論文にして発表し、トップエンドの査読付き学会で採択してもらことを目標に置きたい。
- 当該OSSプロジェクトに対する改良等の提案を期待する。
- 特に2020年のVLDBは東京で開催されることが決定しているので、ここを一つのターゲットとする。

概念図：ソフトウェアスタックイメージ