Физический факультет МГУ им. М.В. Ломоносова
Москва 2017

# О ВОЗМОЖНОСТИ СОЗДАНИЯ ЭНЕРГОЭФФЕКТИВНЫХ АЛГОРИТМОВ ИНДЕКСИРОВАНИЯ НАГРУЖЕННЫХ СУБД НА СЕРВЕРАХ НА БАЗЕ СПЕЦИАЛИЗИРОВАННЫХ МНОГОЯДЕРНЫХ ПРОЦЕССОРОВ

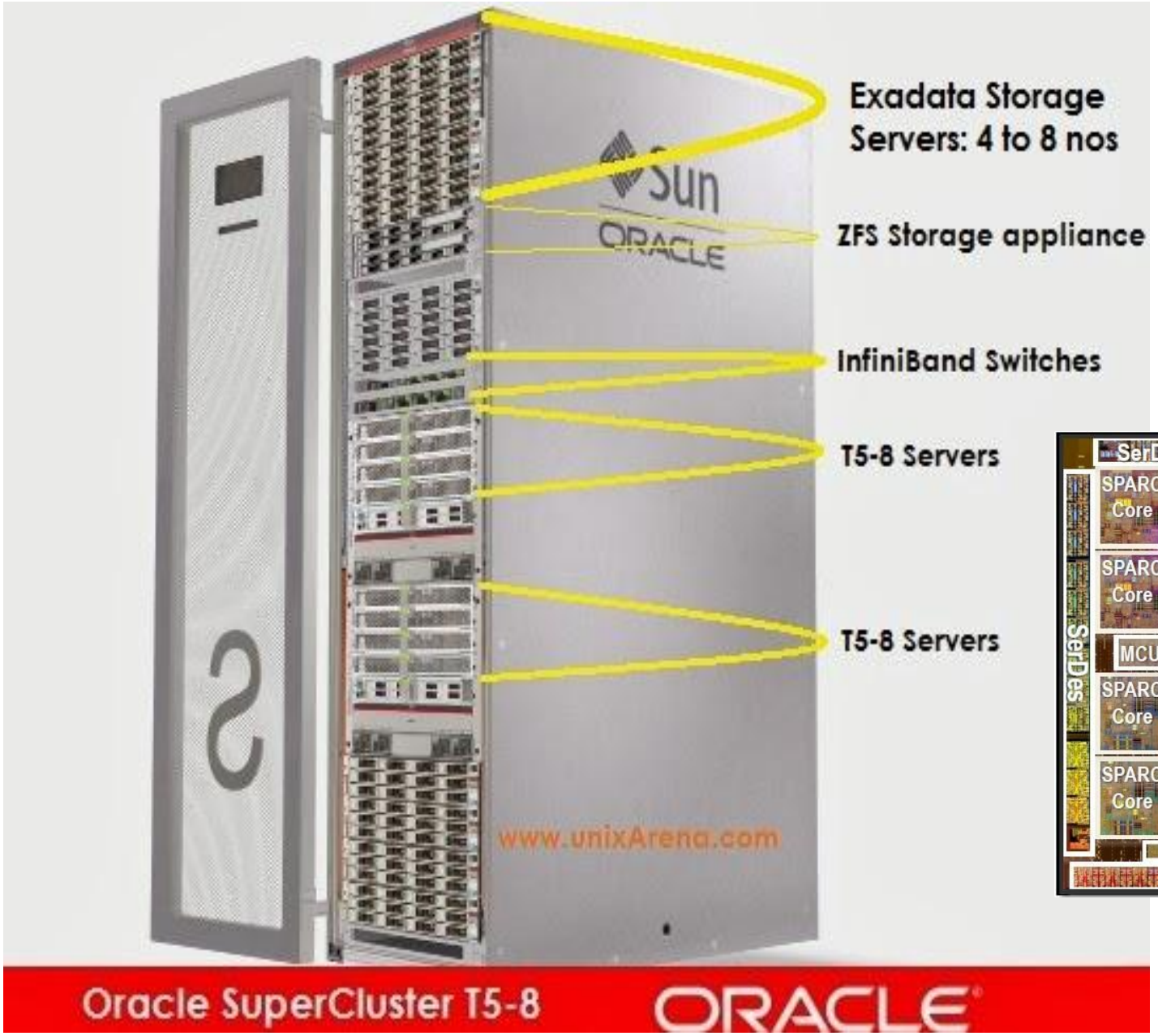
*Сизов Анатолий*
*Елизаров Сергей*
*Щедов Юрий*

# Содержание

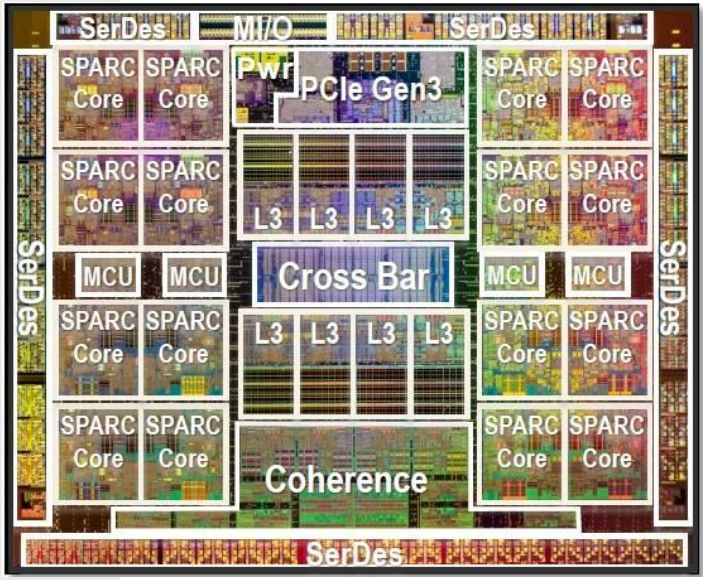
# Некоторые черты современных СУБД

Сегодня подавляющее большинство СУБД построены на универсальных x86 процессорах. Разработчики СУБД и приложений для повышения производительности год за годом адаптируют код и алгоритмы под распространенные аппаратные платформы и разработанное для них системное ПО, преимущественно Linux. Стоимость разработки и поддержки современных СУБД значительно превышает цену “железа”.


# Некоторые черты современных СУБД







# Некоторые черты современных СУБД


Социальные сети сегодня - мощный стимул развития СУБД:

- Гигантский объем данных;
- Графовое представление;
- Ограниченная когерентность;
- Постоянное обновление.

# Развитие кремниевой технологии, специализация и энергетика

- АЛУ уже давно не является ключевым компонентом процессора
- Производительность ограничивается не кол-вом транзисторов, а их энергетикой
- Стоимость переключения транзистора практически не падает с технологическим прогрессом
- Основной подход к повышению производительности - специализация + многоядерность


**Пример**: изготовленный в 2016 Sunway TaihuLight содержит CPU с 260 вычислительными ядрами, против 72 ядер еще не выпущенного Intel Knights Landing. Уже достигнутая энергоэффективность TaihuLight **6 ГФ/Вт.**, лучше чем планируемая у Intel.

# Возражение программиста: хочу абстрагироваться от “железа”

- Увы, уже давно не получается;
- Оптимизация кода под размер кэша;
- Избегание “дорогих” операций записи в общий ресурс.

# Постановка задачи

**Цель работы:**

**Исследовать возможность создания энергоэффективных алгоритмов для СУБД оптимизированных для исполнения на серверах на базе специализированных многоядерных процессоров.**

# Стандарты тестирования производительности СУБД

**Высокоуровневые тесты производительности (уровня SQL-запросов)**

- **Промышленные тесты TPC.org —** проработанные спецификации тестирования и методики выявления узких мест в различного рода системах (OLTP, принятия решений, виртуализации, BigData и энергоэффективности). Производители коммерческих СУБД конкурируют за высокие места в тестах TPC.
- **Открытые высокоуровневые тесты** - предназначены для тестирования определенных СУБД или работу СУБД в специфической среде:
  - **Sysbench OLTP и PGbench —** пакеты тестирования MySQL и PostgreSQL;
  - **Linkbench** — эмуляция нагрузки на систему Facebook TAO (ограниченная версия графа, используемая как быстрый интерфейс конечного пользователя к ресурсам FB), работающей на базе MySQL. Linkbench - имитация одной составляющей большого пула серверов TAO. В MySQL генерируется БД графа на базе вероятностного распределения. Тип запросов: одиночный поиск, короткие диапазонные запросы, вставка и удаление узлов графа).

# Стандарты тестирования производительности СУБД

**Высокоуровневые тесты производительности (уровня SQL-запросов)**

- **Промышленные тесты TPC.org —** проработанные спецификации тестирования и методики выявления узких мест в различного рода системах (OLTP, принятия решений, виртуализации, BigData и энергозатрат). Производители коммерческих СУБД конкурируют за высокие места в тестах TPC.
- **Открытые высокоуровневые тесты** - предназначены для тестирования определенных СУБД или работу СУБД в специфической среде:
  - **BG Bench** — схожий с Linkbench пакет тестирования, предназначенный для тестирования MongoDB, SQL-X и др.
  - Тесты баз данных NoSQL через их командный интерфейс (Redis и др).

**Недостатки: оторванность от реальной жизни, трудность сопоставления результатов, проведенных на различном железе, ограниченная масштабируемость, использование производителями СУБД специальных оптимизаций «под тест».**

# Стандарты тестирования производительности СУБД

**Тесты производительности уровня API -** методики тестирования, работающие как генераторы запросов через библиотеки API. Тесты производительности разрабатываемых «с нуля» алгоритмов индексирования: MassTree, BW-Tree, PalmTree, AtomicHashMap, RB-Tree, KISS-Tree, ARTfull и некоторых других.

**Недостатки: трудность сопоставления результатов на разном железе, отсутствие стандартов де-юре и де-факто, отсутствие чёткой трактовки понятия «операции записи» (как вставки, обновления или удаления), различие в типах хранилищ данных.**

# Стандарты тестирования производительности СУБД

**Низкоуровневые тесты критически важных мест (micro-benchmarks) СУБД.** Инструменты тестирования синхронизации многопоточного доступа к различным типам структур данных — попытка сопоставить работающие в памяти хранилища данных.

- **Synchrobench** — пакет тестов, написанный C/C++ и Java, который позволяет исследовать поведение массивов, бинарных деревьев, хэш-таблиц, очередей, связных списков и др. структур при работе с разными типами блокировочных и безблокировочных алгоритмов, использующих программные и аппаратные решения;

- **ASCY (asyncronized concurrency)** — попытка на основе Synchrobench выделить главные аспекты масштабируемости и переносимости многопоточных алгоритмов работы с хранилищем данных. Анализ аппаратных расширений x86.

**Преимущества: Возможность сравнения различных архитектур и алгоритмов.**

# Стандарты тестирования производительности СУБД

**Высокоуровневые тесты на уровне SQL-запросов (TPC, Sysbench , PGBench):**

- Закрытость коммерческих реализаций СУБД, сложность сравнения систем между собой;
- Влияние модулей верхнего уровня СУБД на результаты;

**Тесты уровня API (BW-Tree, PalmTree, Kiss-Tree):**

- Отсутствие стандартов на использование запросов разного типа (поиск/вставка/удаление);

**Micro-benchmarks (Synchrobench, ASCY):**

- Возможность сравнить разные архитектуры и алгоритмы;

# Подход компании Facebook, горизонтальное масштабирование

**Facebook TAO**

- Интерфейс пользователя к основной БД Facebook;
- **B-Tree**;
- MySQL;
- Одной из задач является быстрое добавление информации, реализующееся за счет репликации и упрощение архитектуры строения данных (таблица объектов, таблица связей, таблица счетчиков, таблицы сопутствующих данных);
- Наиболее используемые виды связей: симметричная по направлению (friend), направление от пользователя к объекту (likes), направление от понравившегося объекта к пользователю (likers);
- Разреженность графа, наличие плотных сообществ, слабо связанных с остальным графом.

# Подход компании Facebook, горизонтальное масштабирование

**Facebook Unicorn**

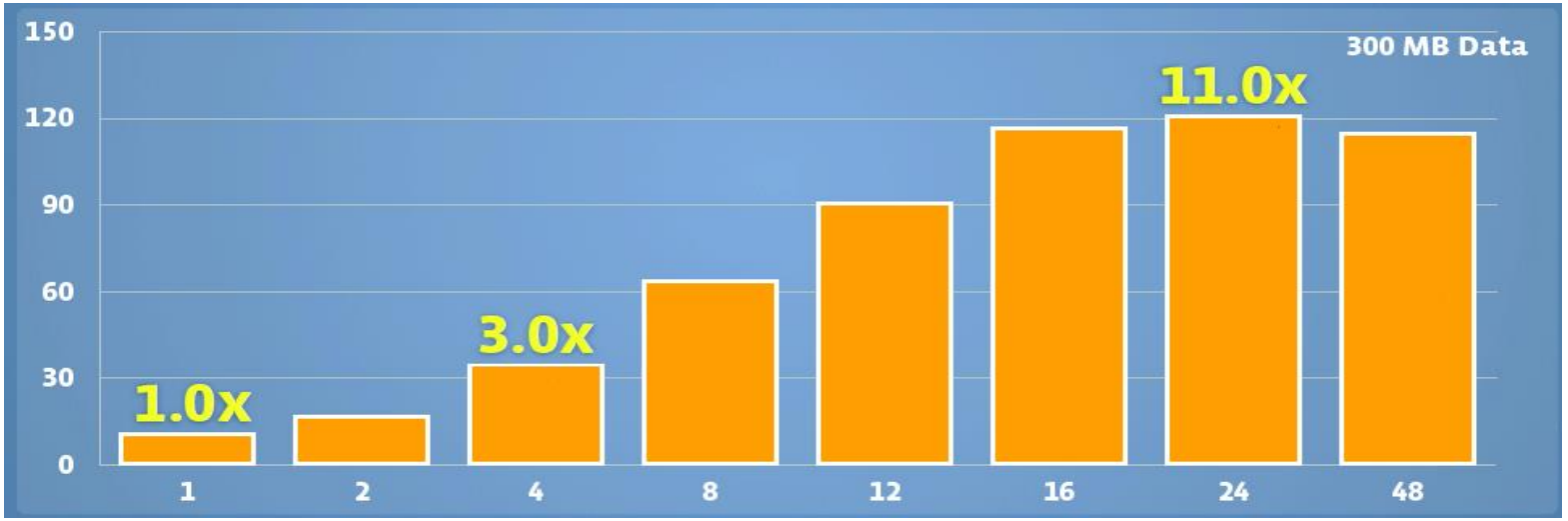
- Хранение изолированных фрагментов общего графа Facebook;
- Используется инвертированный индекс и собственный алгоритм **AtomicHashMap**;
- Множество вертикалей данных;
- Ручное планирование разбиения БД на разделы (shard);
- Ручное планирование архитектуры данных и направлений сортировки, исходя из анализа статистики запросов пользователей;
- Использование промежуточного языка запросов высокого уровня для перепланирования нагрузки;
- Для управления всей распределенной БД используется надстройка-агрегатор.

# Подход компании Facebook, горизонтальное масштабирование

**Facebook Unicorn**

- За счёт **AtomicHashMap** достигнуты хорошие показатели производительности и масштабируемости каждого сервера: рост производительности почти пропорционален числу ядер;
- **Недостатки:** алгоритм узко специализирован в рамках задачи, требуется непрерывное ручное перепланирование архитектуры хранилища.

***Зависимость производительности от количества потоков. Intel Xeon X5650@2.67Ghz***






*https://www.facebook.com/Engineering/posts/374508222598113*

# Выбор алгоритма индексирования, B-tree структуры данных

**Индексирование** — основа обработки данных в большинстве реляционных и нереляционных СУБД. Индексирование позволяет ускорить доступ к данным в десятки раз. Индекс может быть использован как карта указателей на данные, так и быть самим хранилищем данных.

**B-Tree:**

- хранение данных в индексе в отсортированном виде;
- поддержка диапазонных запросов (между x и y);
- поддержка запросов на упорядочение (ORDER);
- поддержка многоколоночного индекса.

**Примеры коммерческих систем:**
Facebook TAO

**Hash-индексирование:**

- Быстрые, по сравнению с B-tree операции поиска;
- Требует меньше памяти.

**Примеры коммерческих систем:**
Facebook Unicorn

# Выбор алгоритма индексирования, B-tree структуры данных

**Индексирование** — основа обработки данных в большинстве реляционных и нереляционных СУБД. Индексирование позволяет ускорить доступ к данным в десятки раз. Индекс может быть использован как карта указателей на данные, так и быть самим хранилищем данных.

**B-Tree:**

- Хранение данных в узлах;
- Быстрая, по сравнению с B+Tree обработка одиночных запросов.

**B+Tree:**

- Узлы хранят только указатели;
- Быстрая, по сравнению с B-Tree, обработка диапазонных запросов;
- Операции вставка и удаление проще, чем в B-Tree.

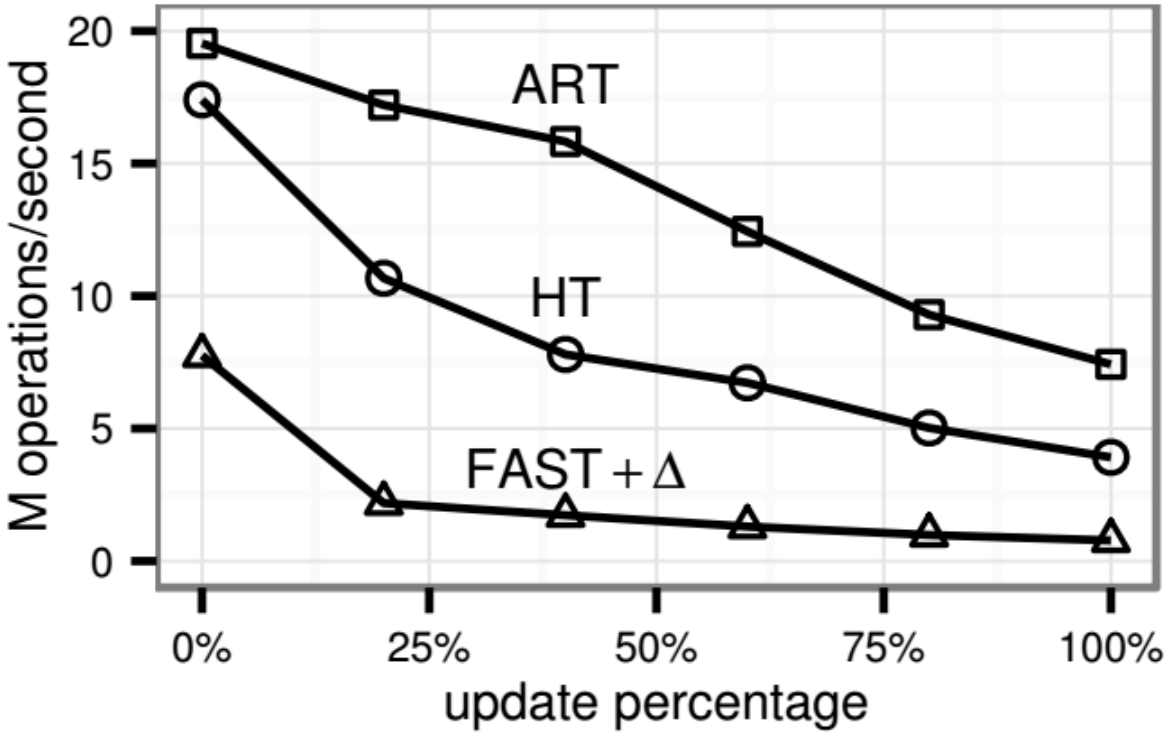
**B+Tree самый актуальный на данный момент алгоритм индексирования, позволяющий добиться высокой производительности при конкурентной нагрузке R/W запросами.**



*Goetz Graefe (2011), "Modern B-Tree Techniques"*

# Общий доступ к памяти и lock-free алгоритмы

Наличие даже незначительного количества запросов на **запись** может существенно ухудшать производительность СУБД.




***Производительность алгоритмов индексирования в зависимости от количества операций update.***
***Intel Core i7 3930K***

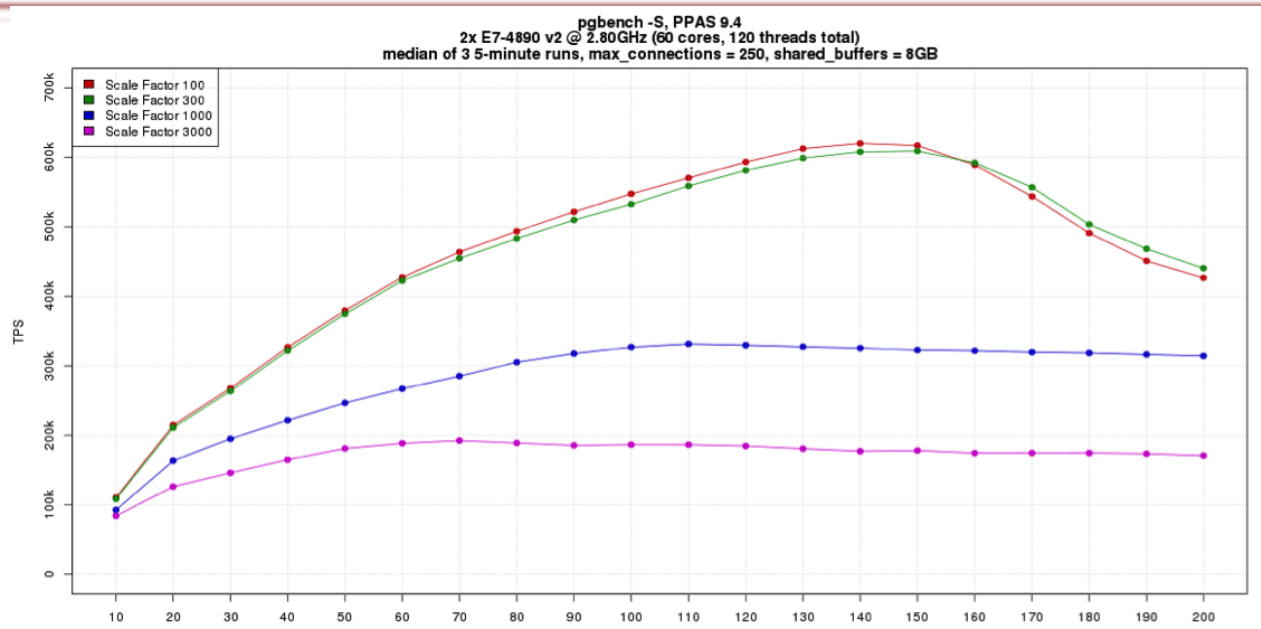
**Типичная распределение типов запросов для OLTP нагрузки:**

**80% чтение/20% запись.**

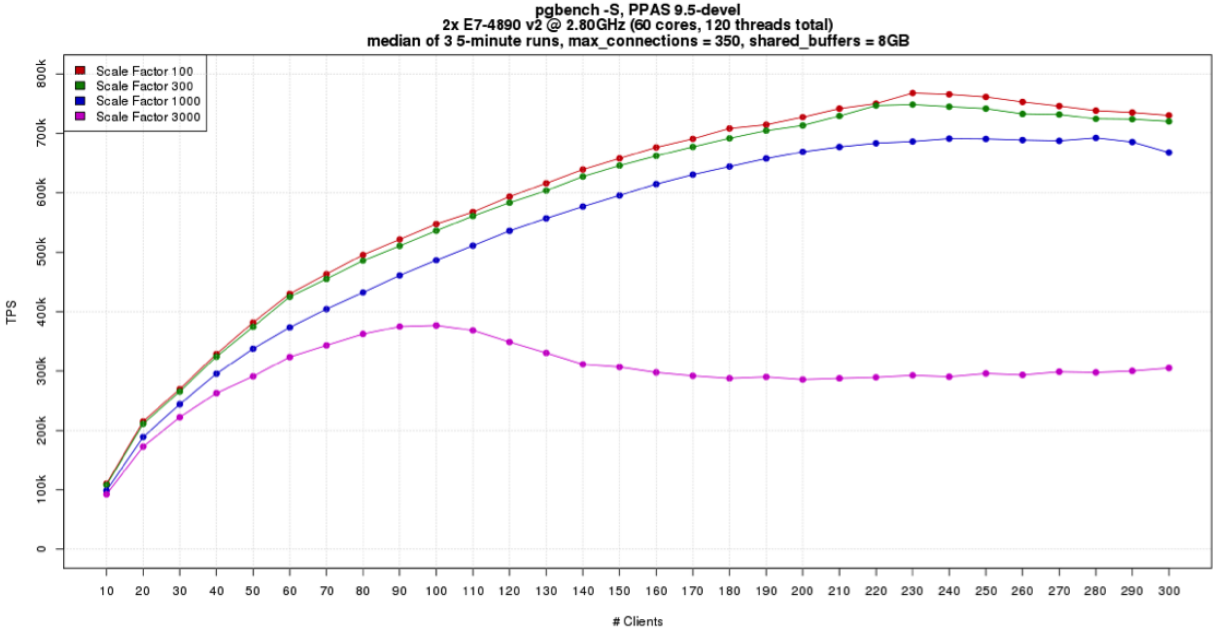


*Leis V. et al. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases, 2014*

# Общий доступ к памяти и lock-free алгоритмы




***PGBench, PostgreSQL, 4x Intel® Xeon® E7-4890 v2, Производительность СУБД в зависимости от количества клиентов.***




***PGBench, PostgreSQL, 4x Intel® Xeon® E7-4890 v2, Производительность улучшенной версии СУБД в зависимости от количества клиентов.***

*https://www.enterprisedb.com/blog/enterprisedb-postgres-plus-and-lenovo-3850-x6-equals-performance-and-scalability*

# Общий доступ к памяти и lock-free алгоритмы

**Сложности разработки Lock-free алгоритмов индексирования.**

• Атомарные операции на современных процессорах x86 дороги и требуют блокировок кэш-линий либо дополнительного кэш-трафика;

• Работа с атомарными переменными и lock-free алгоритмами требует тонких манипуляций с различными моделями памяти (пример, модели в C++);

• Разработка эффективного lock-free алгоритма, как правило, требует большой зависимости от аппаратной платформы и плохо переносится;

• Высокая сложность разработки и тестирования истинно lock-free алгоритмов;

• Недостаточно проработанные механизмы аппаратного HTM (hardware transaction management) на x86

**Виды алгоритмов B+Tree без блокировки:**

• **BW-Tree** (Microsoft Azure);
• **MassTree;**
• **PalmTree.**

# Промежуточные выводы

- Методы тестирования производительности СУБД разнообразны, выбор определенного метода зависит от задач, стоящих перед разработчиком СУБД;
- Большие объемы данных и невозможность обеспечить необходимую для OLTP производительность толкают разработчиков СУБД к повышению производительности за счет разбиения/репликации БД на большое количество серверов;
- B-Tree является основой основной части БД, а алгоритм индексирования на его основе является наиболее актуальным;
- Наличие даже небольшого количества операций записи приводит к значительному понижению производительности СУБД за счет необходимости блокировки по данным на нижнем уровне, и блокировки транзаций на верхнем;
- Разработка lock-free алгоритмов на основе B+Tree является наиболее актуальной задачи повышения производительности СУБД в рамках сервера.

# Анализ решений BW-Tree

**Разработка BW-Tree (реализована в Microsoft Hekakton и Azure DocumentDB)**

- **Эластичные страницы.** Узлы дерева являются логическими, размер не фиксируется, используется кэширующая таблица соответствий физических узлов логическим. Внутри страниц (узлов) используются PIDы - логические указатели на другие узлы, так что при записи физической страницы нет нужды в обновлениях до самого корня дерева.
- **Дельта-обновления.** При обновлениях создается страница с новым содержимым, адрес на которую в кэширующей таблице соответствий в последствие становится окончательным;
- **CAS-инструкции.** Широко используются операции без блокировок на базе CAS-инструкций. Например, деление узлов дерева делается серией CAS-операций;
- **Стабильное хранилище**, структурированное логами, служит постоянным хранилищем и устроено на подобие транзакционного лога. При любых модификациях в него попадают дельта-страницы (т. е. только обновленные узлы).

# Анализ решений BW-Tree

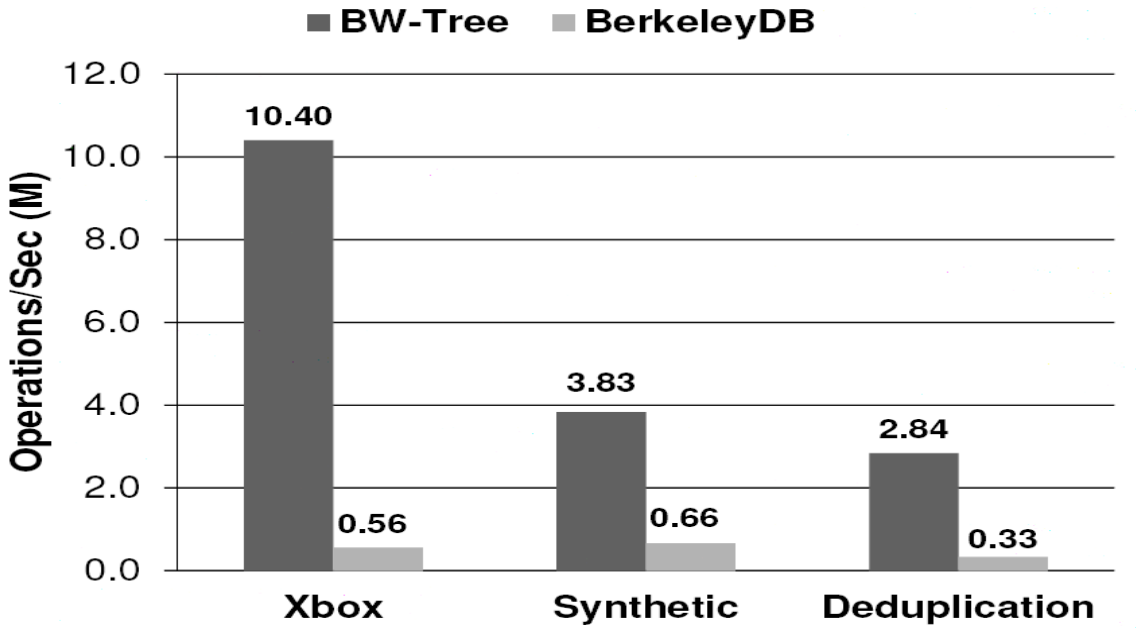
**Xbox LIVE** — хранилище с ключом в 94 байт и значением в 1200 байт. Нагрузка в 27млн. операций r/w (из них 11% write);

**Syntetic** — хранилище с ключом в 8 байт и значением в 8 байт, где нагрузка r/w из них 15% - write;

**Deduplication** — хранилище с ключом в 20 байт SHA-1 и значением в 44 байт;

**BerkleyDB** сконфигурирована на максимальную производительность в конкурентной среде - отключен режим транзакций.

***Сравнение производительности BW-Tree и B-Tree. Intel Xeon W3550 (3.07GHz).***






*J.Levandoski et al. The BW-Tree: A B-tree for New Hardware Platforms. Microsoft Research. 2013*

# Анализ решений MassTree

Открытая база данных ключ-значение **MassTree** имеет в своем составе модуль индексирования на основе **Radix-дерева** (префиксного дерева), подмножеством которого являются поддеревья **B+tree**.

**Radix-дерево** эффективно реализует длинные ключи,

а поддеревья **B+tree** — короткие и средние ключи.

## Особенности MassTree

- Блокировки на уровне узла (на базе spinlock) и оптимистический контроль конкурентной работы (операции get не блокируются и не пишут в общую память);
- Сложность реализации диапазонных запросов (проходы по множеству поддеревьев B+Tree);
- Масштабируемость ограничивается ростом стоимости обращений к памяти

***Производительность MassTree на ядро в зависимости от количества ядер.***

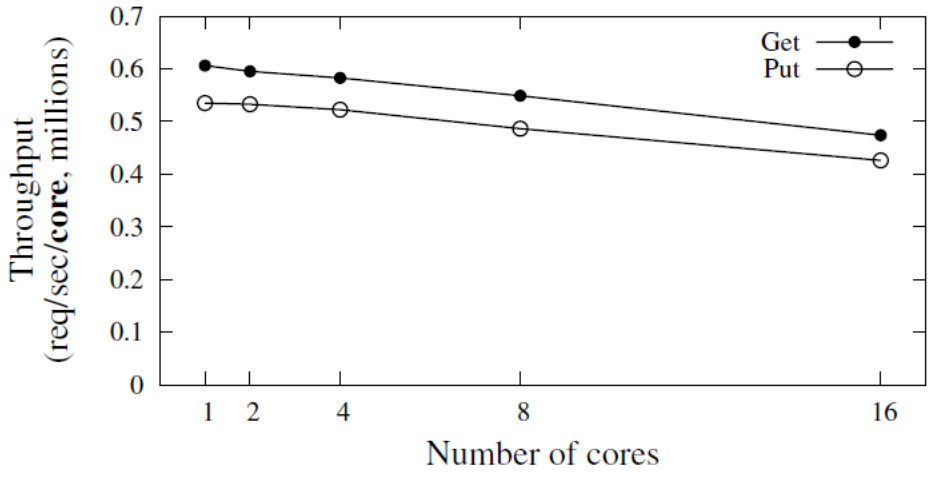



**Figure 10.** Masstree scalability.



*Mao Y., Kohler E., Morris R. T. Cache craftiness for fast multicore key-value storage, 2012*

# Анализ решений MassTree

Сравнительное тестирование (16 ядер, 20млн. пар ключей) показало производительность на чтение, сопоставимую с memcached. В сравнении с Redis преимущество на чтение — 65%, на запись — 51%. Многократное превосходство над VoltDB и MongoDB.

| | Throughput (req/sec, millions, and as % of Masstree) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Workload | Masstree | MongoDB | | VoltDB | | Redis | | Memcached | |
| *Uniform key popularity, 1-to-10-byte decimal keys, one 8-byte column* | | | | | | | | | |
| get | 9.10 | 0.04 | 0.5% | 0.22 | 2.4% | 5.97 | 65.6% | **9.78** | 107.4% |
| put | **5.84** | 0.04 | 0.7% | 0.22 | 3.7% | 2.97 | 50.9% | 1.21 | 20.7% |
| 1-core get | **0.91** | 0.01 | 1.1% | 0.02 | 2.6% | 0.54 | 59.4% | 0.77 | 84.3% |
| 1-core put | **0.60** | 0.04 | 6.8% | 0.02 | 3.6% | 0.28 | 47.2% | 0.11 | 17.7% |
| *Zipfian key popularity, 5-to-24-byte keys, ten 4-byte columns for get, one 4-byte column for update & getrange* | | | | | | | | | |
| MYCSB-A (50% get, 50% put) | **6.05** | 0.05 | 0.9% | 0.20 | 3.4% | 2.13 | 35.2% | N/A | |
| MYCSB-B (95% get, 5% put) | **8.90** | 0.04 | 0.5% | 0.20 | 2.3% | 2.69 | 30.2% | N/A | |
| MYCSB-C (all get) | **9.86** | 0.05 | 0.5% | 0.21 | 2.1% | 2.70 | 27.4% | 5.28 | 53.6% |
| MYCSB-E (95% getrange, 5% put) | **0.91** | 0.00 | 0.1% | 0.00 | 0.1% | N/A | | N/A | |

**Figure 13.** System comparison results. All benchmarks run against a database initialized with 20M key-value pairs and use 16 cores unless otherwise noted. Getrange operations retrieve one column for $n$ adjacent keys, where $n$ is uniformly distributed between 1 and 100.



*Mao Y., Kohler E., Morris R. T. Cache craftiness for fast multicore key-value storage, 2012*

# Анализ решений PalmTree

**PalmTree** — реализация **B+Tree** без блокировок (**lock-free**) с использованием механизма пакетной обработки данных. Рассматриваемый прототип **PalmTree** работает с хранилищем типа ключ-значение.

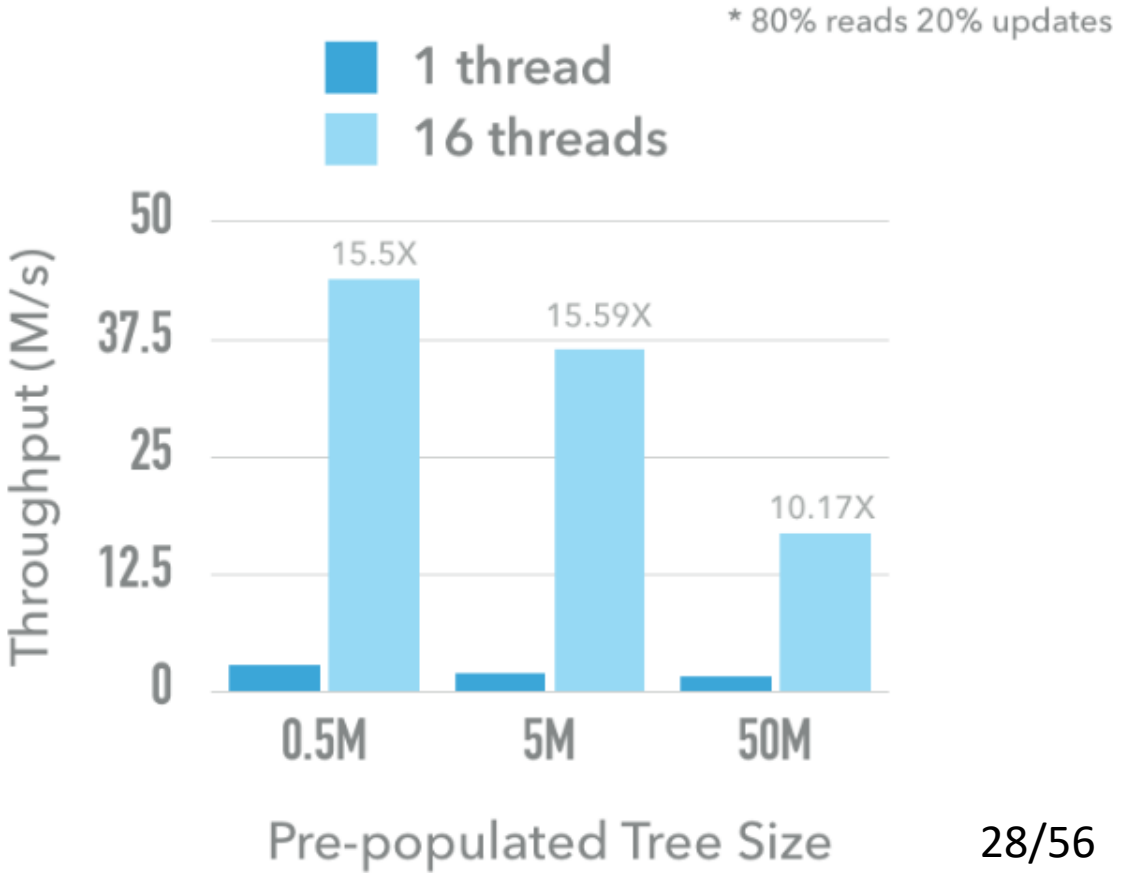
**Особенности PalmTree:**

- **Объединение запросов в пакеты**, которые обрабатываются «совместными усилиями» пула потоков. Таким образом, пакетная обработка, позволяющая выполнять большее количество заданий в единицу времени, должна компенсировать накладные расходы на обмен данными между потоками и расходы на выполнение алгоритмов планирования поточной работы;
- **Разделение задачи на этапы**. При этом между различными этапами должны существовать точки синхронизации, чтобы была возможность отслеживать, что каждый отдельный «исполнитель» закончил работу над своим этапом и готов приступить к выполнению следующего. Пакетная обработка данных позволяет алгоритму не бояться высоких нагрузок на запись.

# Анализ решений PalmTree

Во всех приведенных ниже тестах для PalmTree операции на чтение составляют 80%, операции на запись - 20%, из них 10% - вставка, 10% - удаление.

***Производительность PalmTree в однопоточной и многопоточной реализации в зависимости от размера дерева.***

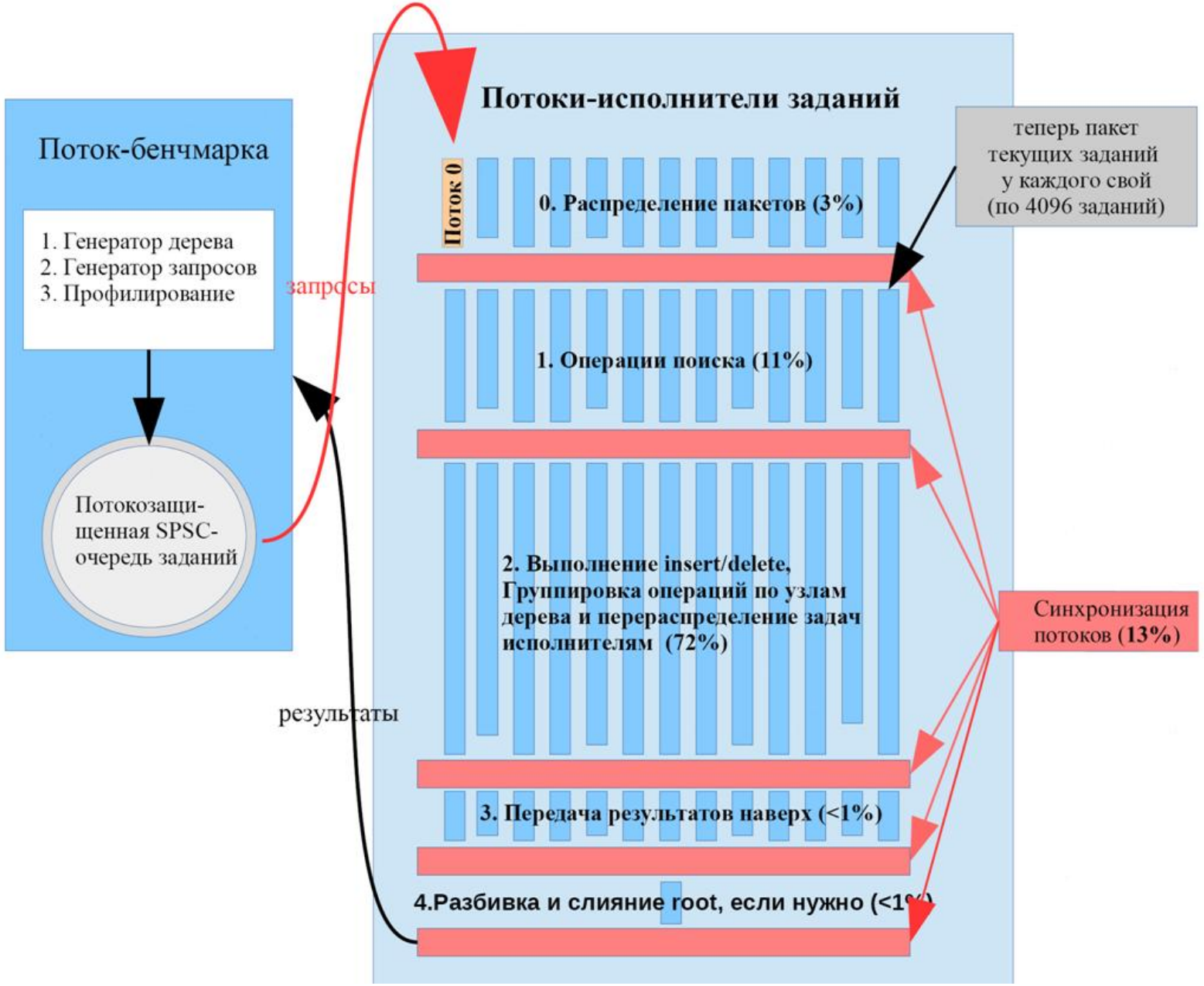



- При небольших размерах дерева удается получить линейный прирост производительности
- При работе с деревом в 10 млн.узлов и выше наблюдается падение производительности вследствие ограничений пропускной способности памяти;

*Runshenzhu. Palm Tree: a Lock free B+Tree. 2012.*

# Анализ решений PalmTree

**Этапы обработки пакета запросов:**



# Анализ решений PalmTree

**Ключевые моменты подготовки пакетов заданий для поддержания целостности:**

• Не предусматривается внесения каких бы то ни было изменений в дерево, пока не закончены все операции чтения;

• Для каждого потока-исполнителя запоминаются те узлы, к которым он получал доступ на нижнем уровне, а затем отбрасываются те узлы, к которым получал доступ исполнитель с меньшим ID;

• Каждый узел дерева, который нужно модифицировать, является «собственностью» одного потока-исполнителя;

**Приняты дополнительные меры для повышения производительности:**

• Перед распределением запросов по потокам - предупреждающая сортировка;

• По получении очередной порции заданий каждый исполнитель в пуле потоков самостоятельно выполняет отбор «удобных» для себя задач;

• Для поиска в узле листьев, которые пока не отсортированы, специально для архитектуры x86 могут использоваться SIMD-операции линейного поиска вместо стандартного бинарного прохода;

# Анализ решений PalmTree

**Эффективная синхронизация потоков (*красная черта*)**

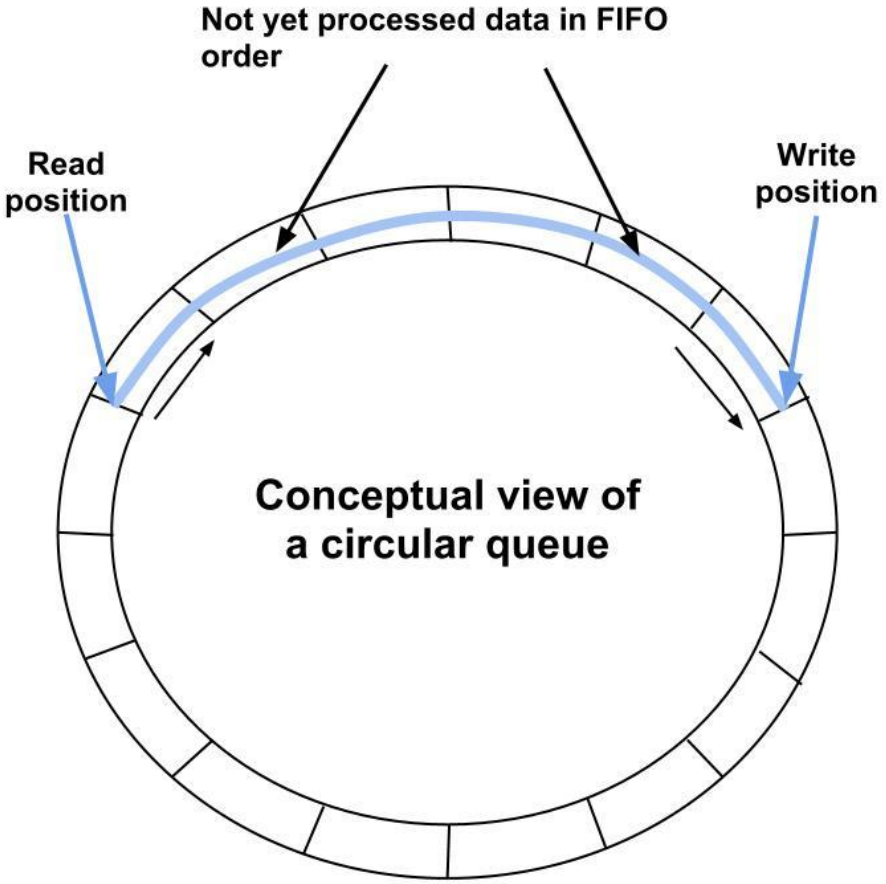
Использование тикет-spinlock в купе с атомарными операциями
эффективнее с точки зрения нагрузки на шину данных, так
как общепринятый spinlock на базе тест + CAS-операции (TAS)
слишком дорог, т. к. каждая операция CAS — это лишний
Трафик на шине данных.

```
std::atomic<int> m_count;
std::atomic<unsigned long> m_generation;
std::atomic<unsigned int> next_ticket;
std::atomic<unsigned int> now_serving;

bool wait() {
  lock();
  auto gen = m_generation.load();
  if (--m_count == 0) {
    m_generation++;
    m_count = P;
    lock.unlock();
    return true;
  }
  unlock();
  while (gen == m_generation);
  return false;
}

void lock() {
    auto my_ticket = next_ticket++;
    while(my_ticket != now_serving) ;
  }
 void unlock() {
    now_serving++;
 }
```

# Анализ решений PalmTree

**Круговая очередь** (SPSC — single produce single Cusomer) используется для целей передачи заданий от клиентов индекса PalmTree потокам исполнителям.

Эту задачу выполняет только два потока. Обеспечивается непрерывность работы обоих потоков.



# Анализ решений PalmTree

**SPSC** — это связный бесконечный список, работающий
по принципу FIFO. Первый вставленный элемент
затирается новым элементом.
Реализация основана на классе Boost::lockfree::spsc_queue, который является lock-free
и wait-free одновременно.
Очередь в PalmTree содержит 5млн. Элементов.

```
#include <boost/lockfree/spsc_queue.hpp>
#include <thread>
#include <iostream>
boost::lockfree::spsc_queue<int> q{100};
int sum = 0;
void produce()
{
 for (int i = 1; i <= 100; ++i)
   q.push(i);
}
void consume()
{
 int i;
 while (q.pop(i))
   sum += i;
}
int main()
{
 std::thread t1{produce};
 std::thread t2{consume};
 t1.join();
 t2.join();
 consume();
 std::cout << sum << '\n';
}
```

# Анализ решений PalmTree

**Быстрый линейный поиск в листьев B+tree**

В процессе обработки листьев во время вставки или удаления элементов в те моменты, когда в узлах нарушен порядок сортировки листьев в место стандартных бинарных операций поиска выгодно использовать операции поиска SIMD

```
const __m256i keys = _mm256_set1_epi32(target);

// Загрузить данные из указанных адресов в вектора
_mm256_loadu_si256 (reinterpret_cast<const __m256i *>(&data[i]));

_mm256_cmpeq_epi32(vec1, keys); // сравнить вектора с ключом в 4-х экземплярах
mm256_movemask_epi8(tmp); // сделать маску из старших битов 8-ми битных элементов
__builtin_ctz(mask) / 2;   // посчитать кол-во 0 слева
```

# Анализ решений PalmTree

**Отличительные черты PalmTree:**

- Предварительная сортировка запросов по ключу;
- За счет пакетной обработки значительно упрощается алгоритм и нет нужды в сложных механизмах блокировок или их обхода (lock elision);
- Формирование отдельного пакета для каждого потока-исполнителя;
- Один узел (страница) B+Tree дерева может быть обработан только одним потоком-исполнителем;
- Хорошая масштабируемость.

**Возможная оптимизация PalmTree:**

Возможна реализация синхронизации точка-точка (когда поток ждёт не всех, а только соседей, т. е. смежные потоки, связанные с текущим общими данными).

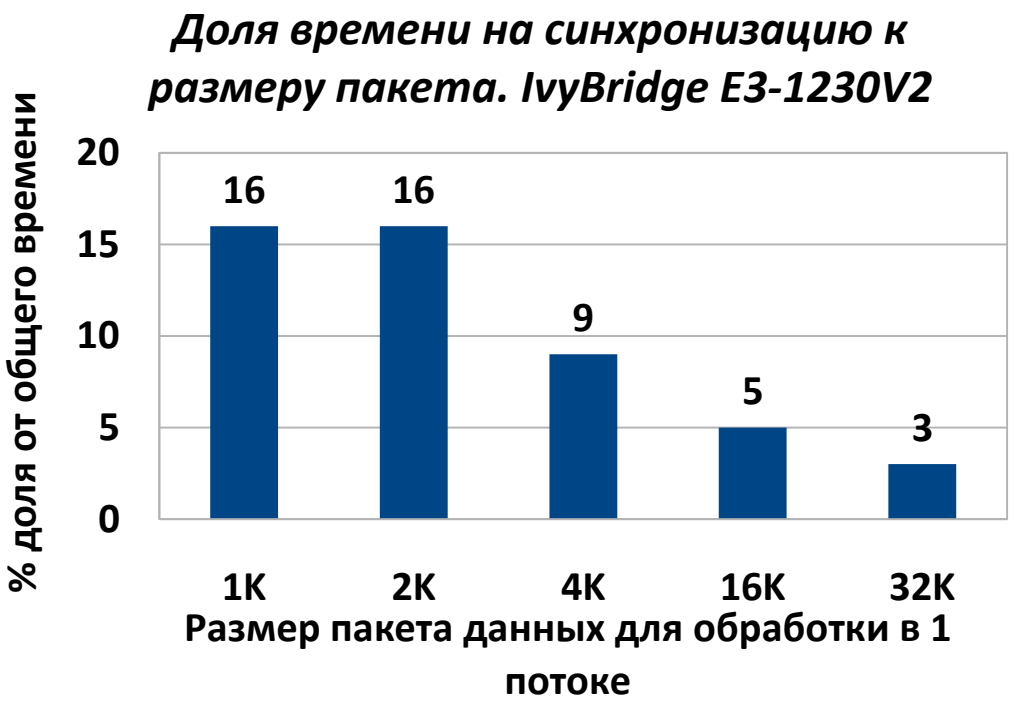
**Недостатки PalmTree:**

- Время отклика на одиночный запрос соответствует времени обработки пакета запросов;

# Анализ решений PalmTree

**Недостаток:** Время отклика на одиночный запрос соответствует времени обработки всего пакета запросов (**4096 запросов — размер пакета для x86**);

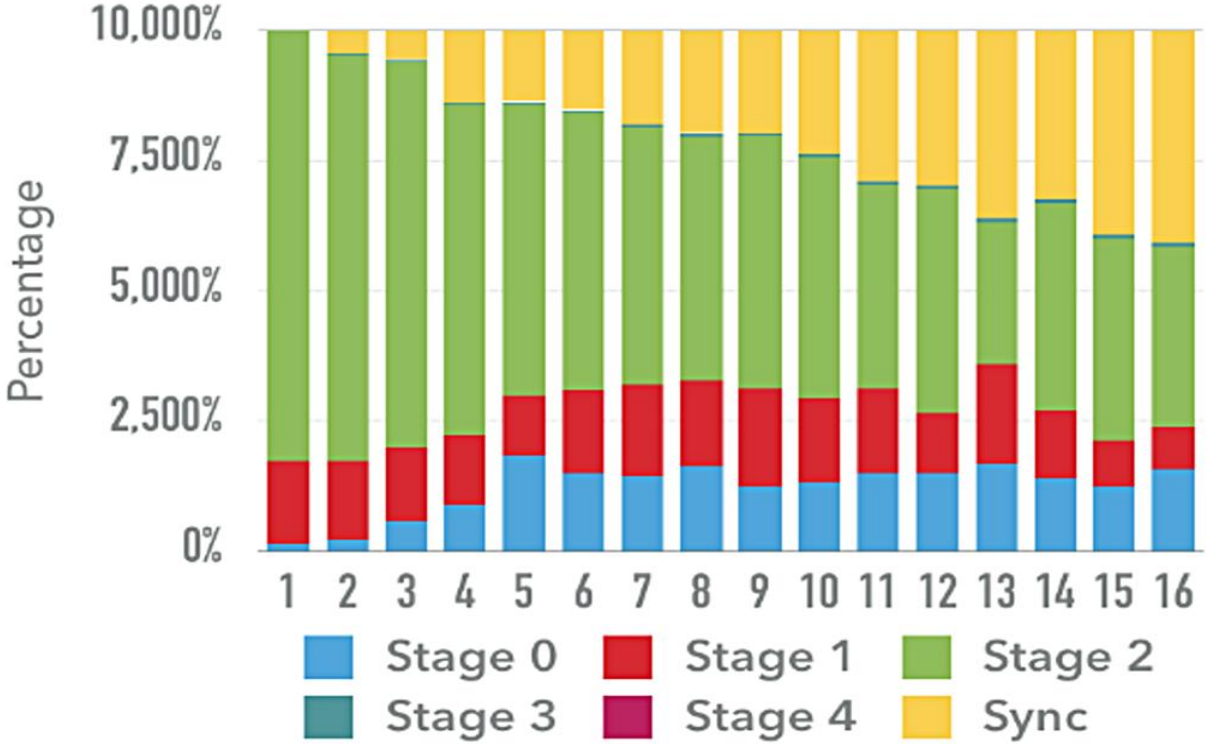
**Возможный путь решения:** на многоядерном специализированном процессоре с большим числом ядер возможно уменьшение размера пакета запросов для одного потока, при этом использование тегированной памяти поможет уменьшить долю времени, затрачиваемого на синхронизацию (см. также следующую страницу).




# Анализ решений PalmTree

При увеличении количества потоков-исполнителей эффективность снижается в том числе за счёт роста доли времени на синхронизацию. На графике при 16 потоках доля времени на синхронизацию достигает 33%, а доля времени на этап 2 уменьшается. Аппаратная реализация памяти на Full/Empty битах может помочь снизить долю синхронизации и улучшить масштабируемость алгоритма за счет замены дорогих атомарных операций платформы x86.

***Распределение времени выполнения программы по этапам для PalmTree в зависимости от количества потоков.***






*Runshenzhu. Palm Tree: a Lock free B+Tree. 2012.*

# Промежуточные выводы

**BW-Tree**:

- Многообещающий алгоритм, позволяющий в полную меру использовать **кэш-память** процессора, а также использующий **дельта-обновления** для избежания блокировок узлов дерева. Активно поддерживается Microsoft.

**MassTree**:

- Алгоритм, позволяющий избежать блокировок за счет **разделения деревья на поддеревья**. Плохо выполняются **диапазонные запросы**, так как необходимо производить поиск в более чем одном B+Tree.

**PalmTree**:

- Решение проблем блокировок осуществляется благодаря **групповой обработке запросов**, позволяющими эффективно разбить задачу на множество **независимых** потоков. При большом количестве потоков значительную роль играют **операции синхронизации**. Использование **дельта-обновлений** может помочь частично избавится **от барьерной синхронизации** и позволит более эффективно использовать многопроцессорную архитектуру.

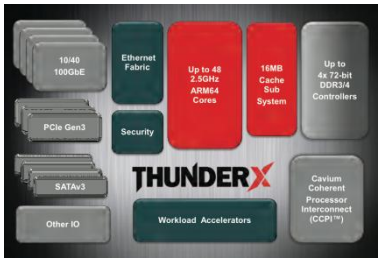
# Мировой опыт

- Мультиядерные процессоры x86, Power, SPARC

**Intel Xeon E5-2650**. 32 нм. 8 ядер. 8 операций двойной точности за такт на ядро. 2 GHz, пиковая производительность 128 GFLOPS. 95 Вт. **1.3 ГФ/Вт**


- Многоядерные процессоры ARM/MIPS

**Calvium ThunderX** 64-bit ARMv8 28 нм. 48 ядер, 2.5 GHz. 80 Вт.

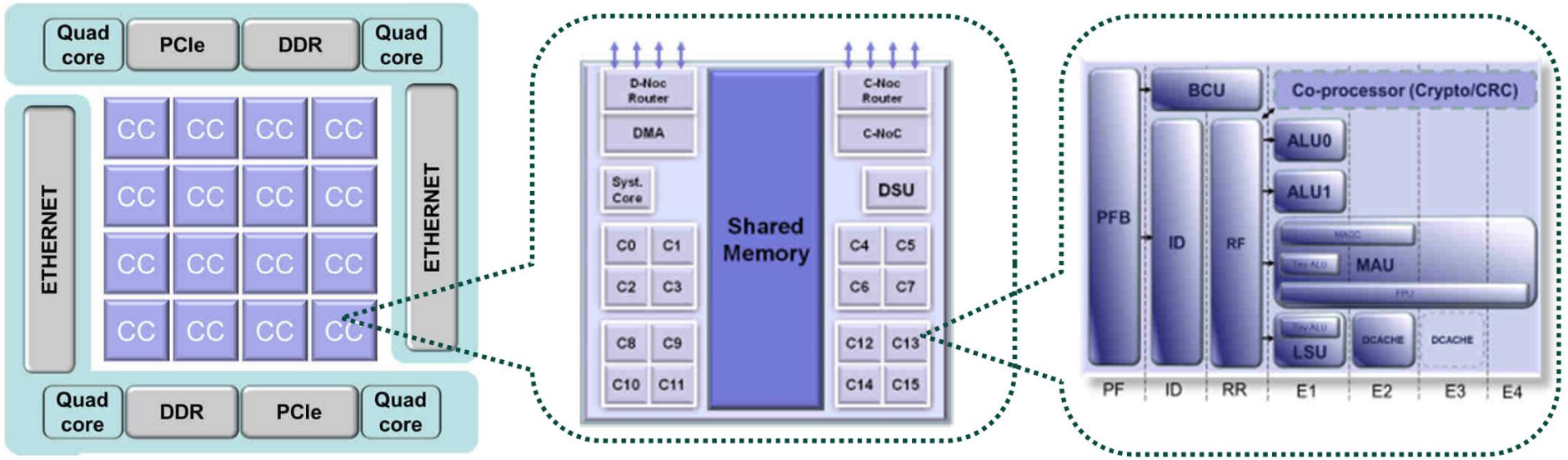



**Kalray2 Bostan** 64-bit. VLIW. 28 мн. 256 ядер. 422 GFLOPS DP. 800Mhz. 10 Вт. **20 ГФ/Вт**

# Мировой опыт

## Kalray2 Bostan

### MPPA®-256 Bostan Processor Architecture




**Manycore Processor**

- 16 compute clusters
- 2 I/O clusters each with quad-core CPUs, DDR3, 4 Ethernet 10G and 8 PCIe Gen3
- Data and control networks-on-chip
- Distributed memory architecture
- 634 GFLOPS SP for 25W @ 600Mhz

**Compute Cluster**

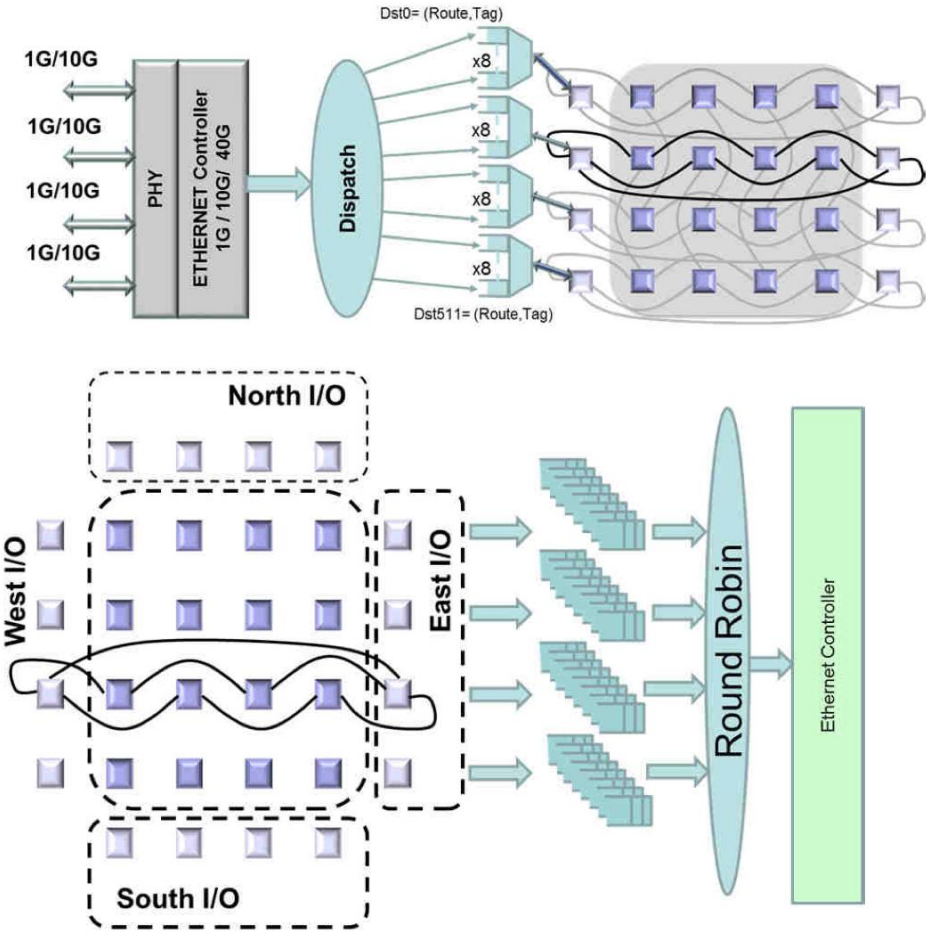
- 16 user cores + 1 system core
- NoC Tx and Rx interfaces
- Debug & Support Unit (DSU)
- 2 MB multi-banked shared memory
- 77GB/s Shared Memory BW
- 16 cores SMP System

**VLIW Core**

- 32-bit or 64-bit addresses
- 5-issue VLIW architecture
- MMU + I&D cache (8KB+8KB)
- 32-bit/64-bit IEEE 754-2008 FMA FPU
- Tightly coupled crpto co-processor
- 2.4 GFLOPS SP per core @600Mhz

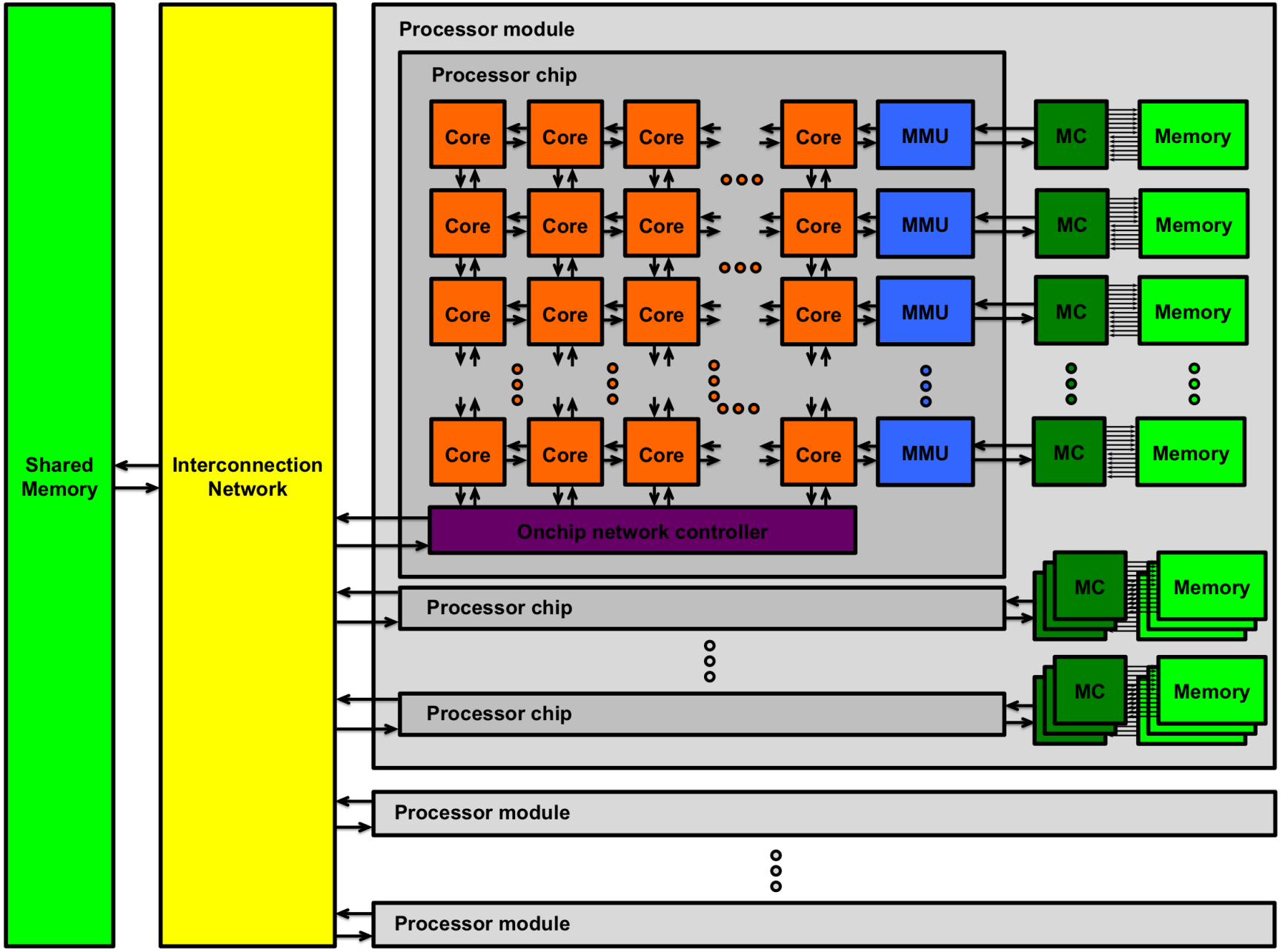
# Мировой опыт

## Kalray2 Bostan

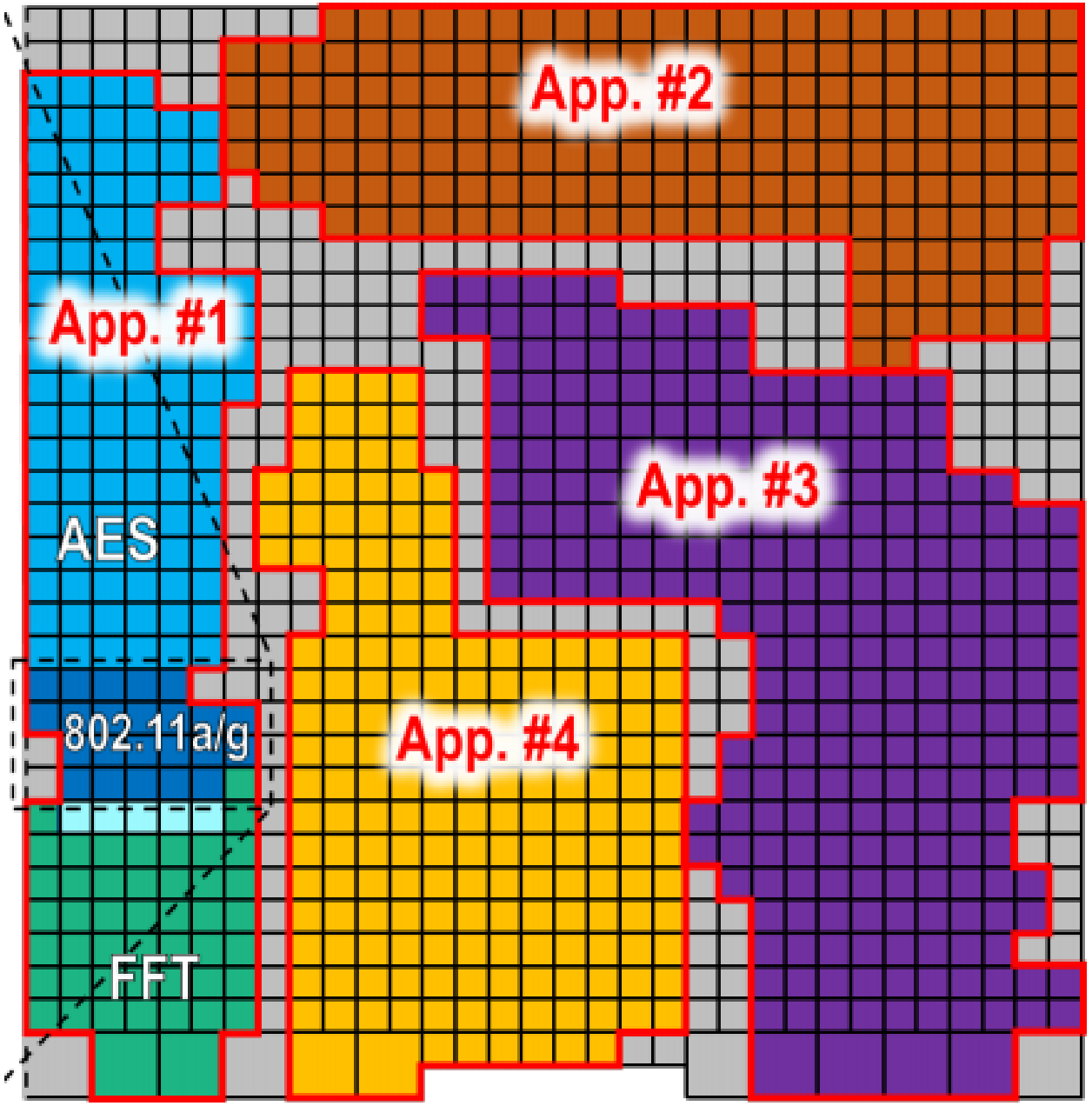



- Ethernet as the main high-performance / low-latency IO
  - Integration of Ethernet Rx and Tx to the D-NoC architecture
- Per Ethernet Rx port
  - 8 classification tables for hardware dispatch
  - Round-robin or classified cluster & core allocation
- Per Ethernet Tx port
  - 64 independent Tx FIFOs
  - Weighted round-robin between Tx FIFOs
  - Flow control between clusters and Tx FIFOs

# Тысячи простых ядер



# Тысячи простых ядер

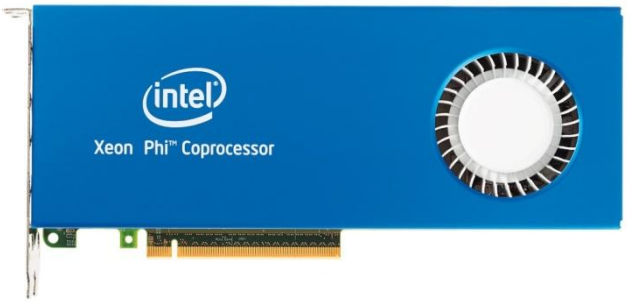


# Мировой опыт

- Графические ускорители NVidia, AMD, Intel

**GeForce GTX 1080**. 16 нм. 6 GB оперативной памяти. 2560 ядер CUDA, 1607 МГц. 4.5 TFLOPS. 180 W. **25 ГФ/Вт.**


**Intel Xeon Phi 3120A**. 22 нм. 57 ядер, 6 GB оперативной памяти, 1100 MHz. 16 операций двойной точности за такт. Пиковая производительность 1 TFLOPS. 300 Вт. **3 ГФ/Вт.**




- Программируемые ускорители Xilinx, Altera


Altera **Stratix GX 2800** 14 nm, 9 TFLOPS (SP) 100 Вт, **50 ГФ/Вт.**

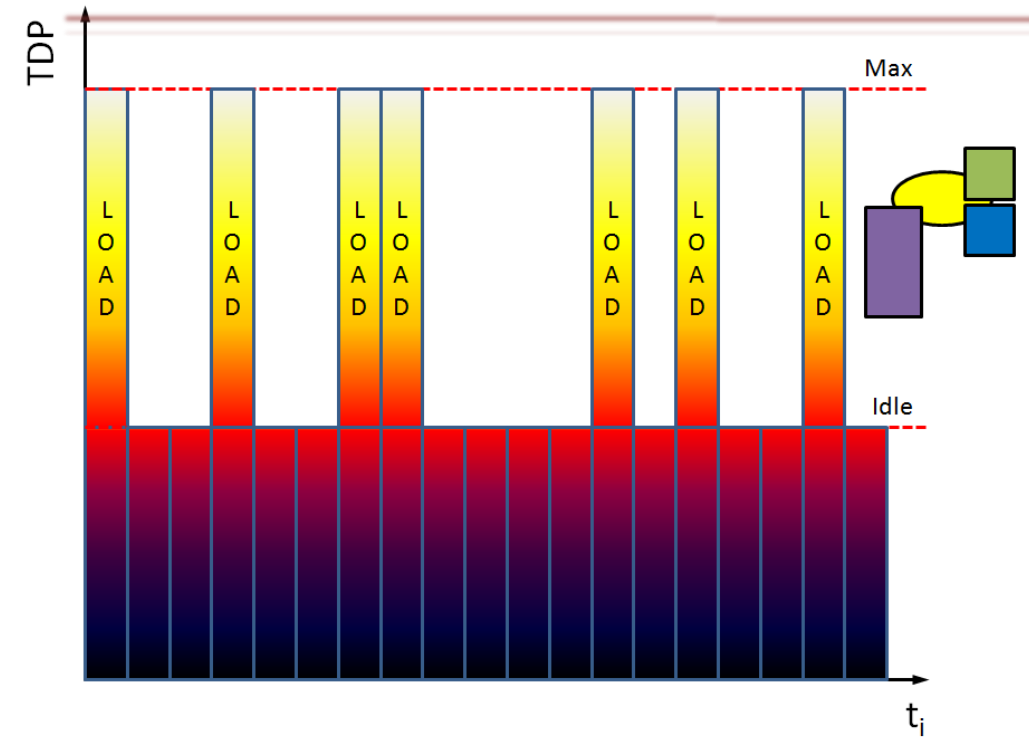
# Лёгкие потоки

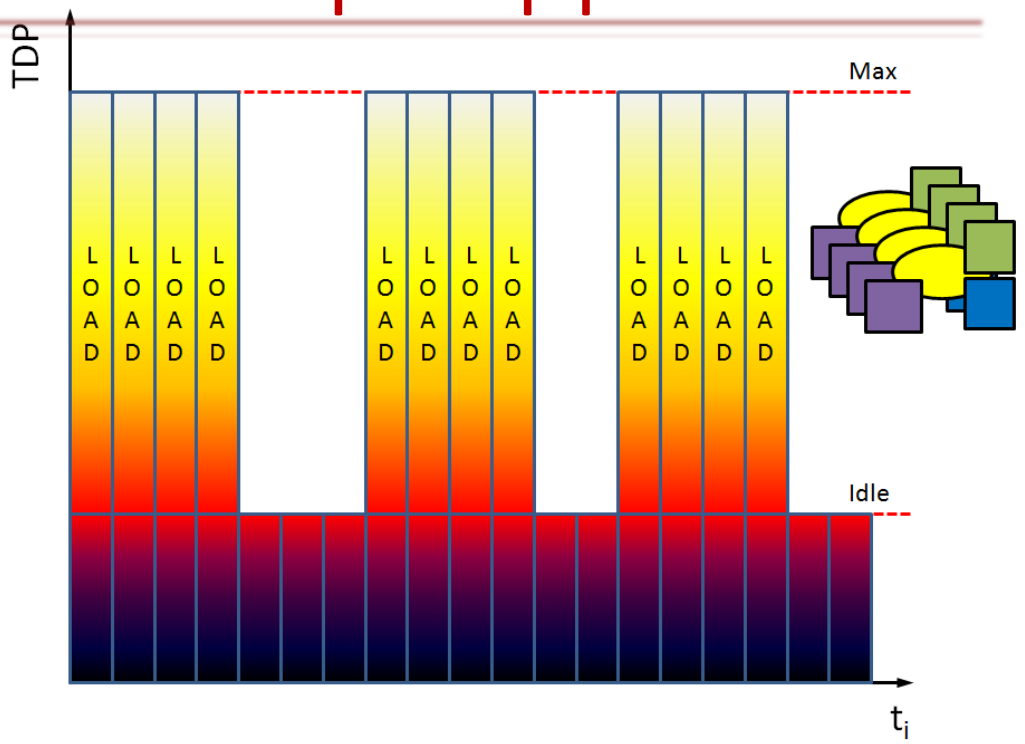
- Принципы и основы реализации “легких” потоков предложены несколькими группами, развивается, например, в проекте Qthread Library от Sandia National Laboratories, патентована Peter'ом Kogge и коллегами, применена в машинах серии Cray XMT.
- Идея - обеспечить механизм порождения, управления и завершения вычислительных потоков не на уровне операционной системы, а на аппаратном уровне со сверхнизкими накладными расходами. В этом случае кроме стандартной операции запуска функции вводится удаленный запуск функции, который производится на одном из свободных ядер в соответствии с заложенными в систему алгоритмами.

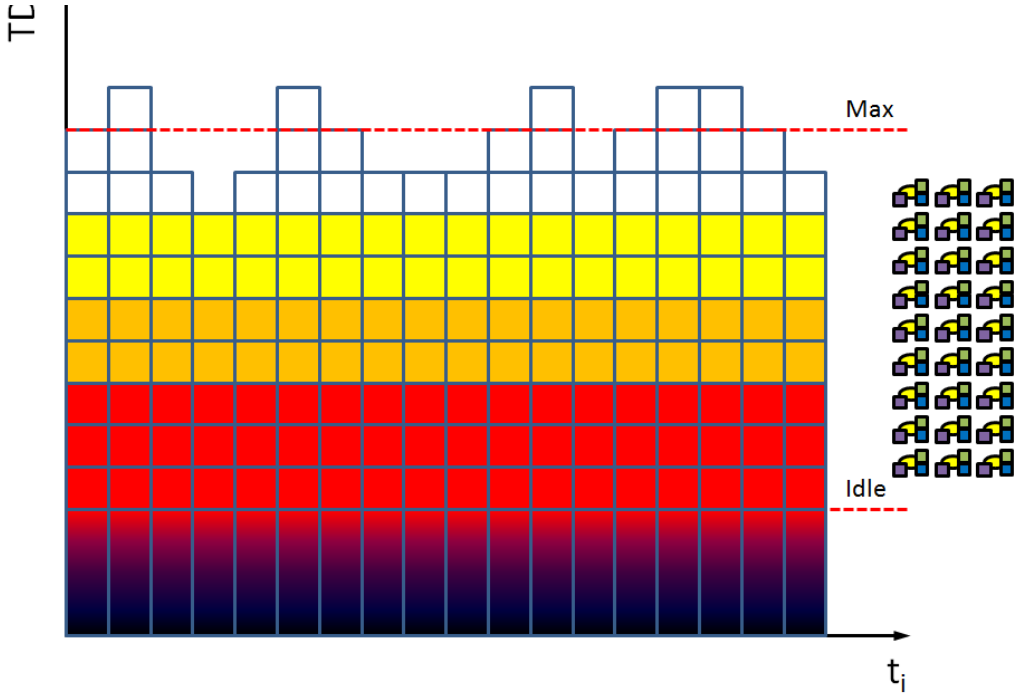
Формализм Qthread:

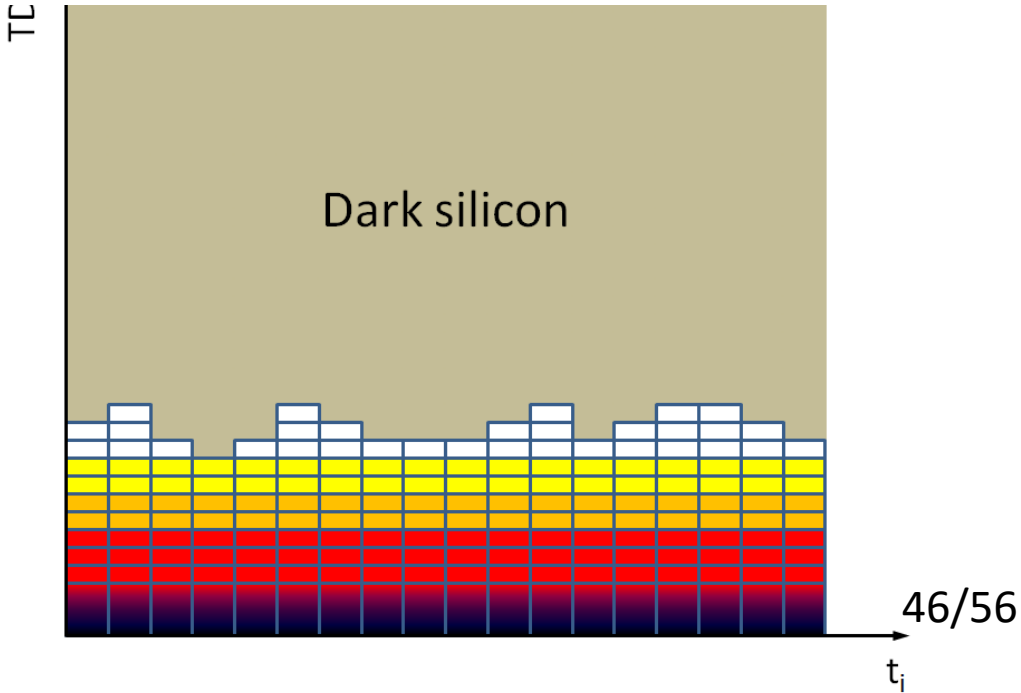
- **int qthread_fork (qthread_f f, const void *arg, aligned_t *ret);**

# Энергоэффективность



# Доступ к общим ресурсам

- Архитектура общей памяти с дополнительными признаками - FE-битами предложена несколькими группами, развивается, например, в проекте Qthread Library от Sandia National Laboratories, патентована Peter'ом Kogge и коллегами, применена в машинах серии Cray XMT.
- Идея - обеспечить механизм работы группы потоков с общем памятью не на уровне операционной системы (семафоры), а на уровне контроллера памяти. В этом случае кроме стандартных операций read и write вводятся readFF и readFE - функции, считывание для которых возможно только из "полной" ячейки, после чтения значение FE-бита ячейки сохраняется или изменяется на "пусто". Такой запрос на чтение, в случае обращения к пустой ячейке блокирует поток, породивший запрос до момента появления данных в указанной ячейке.

Формализм Qthread:

- **int qthread_readFE(aligned_t *dest, const aligned_t *src);**

# FE-биты

- FE-бит, значение

Значение FULL означает, что в ячейке есть данные, и их можно считывать. Значение EMPTY означает, что в ячейке нет данных, и запрос на считывание должен быть отложен до того момента, когда данные в ячейке появятся.

- FE-бит, считывание из ячейки

Простое считывание возможно только при значении FULL, данные считываются, значение бита не изменяется (обычный вариант). Считывание для модификации возможно только при значении FULL, данные считываются, значение бита изменяется на EMPTY. Теперь никакой другой запрос на считывание не будет выполнен, пока какой-то процесс (возможно тот, который изменил бит на EMPTY, а возможно и любой другой) не запишет в ячейку данные.

- FE-бит, запись в ячейку

При любом значении FE-бита запись в ячейку приводит к тому, что бит аппаратно устанавливается в состояние FULL, что означает, что теперь в ячейке есть данные, и их можно считывать. Предыдущее значение бита при этом игнорируется.

Синтаксис: **unsigned int malt_read_modify(volatile void* addr)**

# FE-биты и умная память

Важно, что:

- FE механизмы не порождают глобальных блокировок
- FE механизмы могут использоваться для блокировок любой гранулярности
- FE механизмы не используют механизмы опросов
- FE механизмы не потребляют ресурсы ожидающего потока

Умная память это не только FE, это:

- “настоящие” атомарные операции
- “исполнение инструкций в памяти”, например i++, косвенная адресация
- DMA без DMA
- и многое другое

# Поддержка пользователя

Поддержка привычных пользователю языков и библиотек

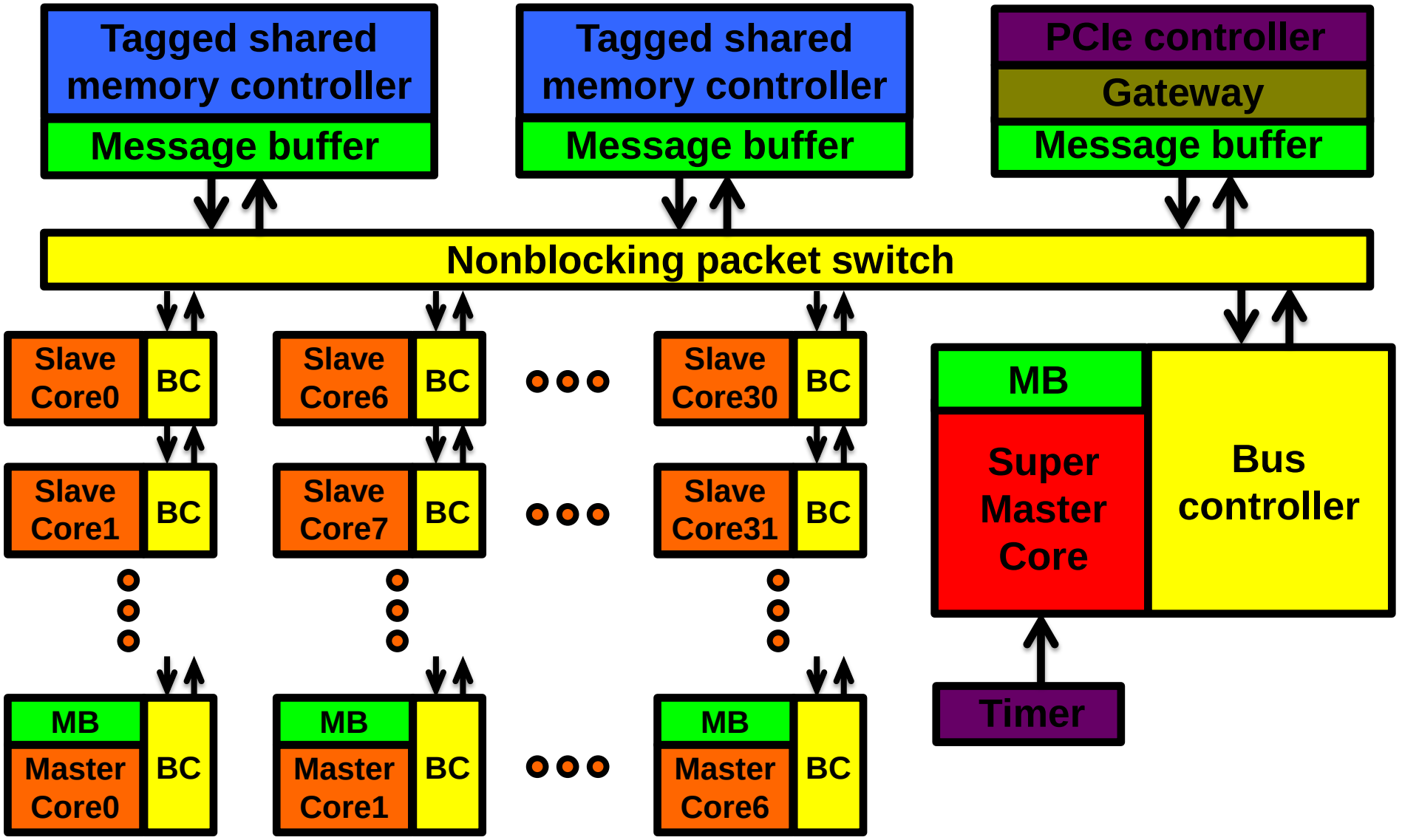
Какой бы ни была аппаратная платформа, программирование должно осуществляться на известном, хорошо изученном и популярном языке. Должна использоваться привычная для программистов среда разработки. Это является необходимым условием для развития. История знает множество интересных и даже гениальных аппаратных разработок, которые не были замечены сообществом программистов или "умерли" не просуществовав и года т.к. не имели привычных средств программирования, опирались на "непопулярные" языки и неудобные среды разработки.

С другой стороны абсолютно необходимым является возможность собирать существующие и разрабатываемые программные пакеты на новой аппаратной платформе, только это позволит без чрезмерных усилий создать достаточный для работы комплект прикладного и системного ПО. Поэтому требуется поддержка по крайней мере на уровне кросс-компиляции наиболее распространенного языка для создания прикладного и системного ПО.

# MALT - Multicore Architecture with Lightweight Threads

MALT system

# MALT - Multicore Architecture with Lightweight Threads



# MALT - Multicore Architecture with Lightweight Threads

- **Компилятор, C/C++**
- **Отладчик GDB**
- **Профайлер**
- **Эмулятор**
- **Прототип на ПЛИС**
- **Прикладные библиотеки**
- **Примеры реализации задач**

# MALT - Multicore Architecture with Lightweight Threads

**Основные черты разработки:**

- **Разработано в России**
- **Не использует закрытых IP**
- **Доступно для тестов**
- **Проверено на эмуляторе**
- **Проверено в ПЛИС**
- **Проверка ключевых блоков в кремнии 2017 году!**

# Выводы

- **Микроэлектроника сделала виток от специализированных решений к универсальным и обратно, сегодня во главу угла ставится энергоэффективность, а не производительность.**
- **Тот те путь проделали компьютеры, они снова перестают быть персональными.**
- **Сегодня ниша специализированных процессоров и соответствующих им СУДБ появилась и еще не занята.**
- **Отечественные разработки в этой области вполне конкурентны, основаны на сильной математической школе и внимании к деталям в области микроэлектроники с высокой степенью специализации.**
- **Указанное выше совместно с проведенными исследованиями говорит о возможности создания энергоэффективных алгоритмов работы с данными нагруженных СУБД на серверах на базе специализированных многоядерных процессоров.**

# Спасибо за внимание!

*Сизов Анатолий Дмитриевич (anatoliy.sizov@gmail.com)*
*Елизаров Сергей Георгиевич (elizarov@physics.msu.ru)*
*Щедов Юрий Геннадьевич (stranger48@yandex.ru)*

*Лаборатория инженерной физики*
*Физический факультет МГУ*