Robust Performance of Main Memory Data Structures by Configuration

Tiemo Bang[*]
TU Darmstadt & SAP SE

Ismail Oukid[†]
Snowflake Inc.

Norman May
SAP SE

Ilia Petrov
Reutlingen University

Carsten Binnig
TU Darmstadt

Abstract

In this paper, we present a new approach for achieving robust performance of data structures making it easier to reuse the same design for different hardware generations but also for different workloads. To achieve robust performance, the main idea is to strictly separate the data structure design from the actual strategies to execute access operations and adjust the actual execution strategies by means of so-called configurations instead of hard-wiring the execution strategy into the data structure. In our evaluation we demonstrate the benefits of this configuration approach for individual data structures as well as complex OLTP workloads.

ACM Reference Format:
Tiemo Bang, Ismail Oukid, Norman May, Ilia Petrov, and Carsten Binnig. 2020. Robust Performance of Main Memory Data Structures by Configuration. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389725

1 Introduction

Motivation: Within the last decade, we have seen different hardware trends that significantly affected the design of single-node database systems: (1) Increases in main-memory capacities made it possible to hold even larger data sets in RAM, thus eliminating the I/O bottleneck of accessing secondary storage (e.g., hard drives). (2) Moore's Law and Dennard Scaling required processor designers to move from single-socket and single-core designs to multi-socket and multi-core designs. As a result of these trends, we have seen a rapid evolution of hardware designs differing in essential characteristics not only memory capacities but also the underlying topology of how cores and memory are connected as well as cache sizes and coherence protocols.

A considerable body of existing work in DBMS research has thus focused on optimising the design of core DBMS data structures such as indexes for specific hardware configurations and workloads. For example, there have been various design alternatives proposed for classical B-trees to adapt them to modern memory hierarchies and make them more cache-conscious for read-heavy workloads [33, 34] or to optimise their behaviour for high-contention scenarios [25] under write-heavy workloads. A significant issue with this manual tailoring of core DBMS data structures is that not only their redesign involves high effort and reintegration into the DBMS but also that a design optimal for one hardware generation and one workload might induce severe performance degradation on another hardware generation when underlying assumptions change.

An alternative to this approach is designing data structures that can provide robust performance [18]. At its core, robust performance means the ability of a data structure to provide acceptable performance for a wide variety of hardware configurations and environmental conditions without adjusting the fundamental data structure design. Achieving robust performance for a data structure, however, is a non-trivial problem because there can be many superimposed causes degrading its performance, not all of which are foreseeable given the speed modern hardware platforms evolve.

Contribution: In this paper, we thus present a new approach for achieving robust performance. Instead of proposing a single design that is robust against different workloads and hardware characteristics, we suggest that data structures can be adapted to a workload and hardware by simple means of a configuration. The main idea to achieve this goal is to strictly separate the design of a data structure from the actual access

[*] tiemo.bang@cs.tu-darmstadt.de
[†] At SAP SE when contributing to this work.

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389725

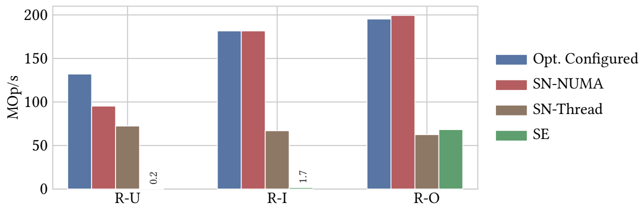

Figure 1: Robust throughput of the *FP-Tree* index on 8 sockets through individually optimal configuration using virtual domains (*Opt. Configured*) for Read-Update 50/50 (R-U), Read-Insert 95/5 (R-I), and Read-Only (R-O) YCSB workloads. Baselines are rigid partitioning strategies: Partition per NUMA region (*SN-NUMA*), partition per thread *SN-Thread*), and a shared-everything strategy (*SE*) without any partitioning.

operations and use a configuration policy for defining the strategy of how to execute access operations on a particular data structure in a declarative manner. This strict separation provides us then with the flexibility to control execution by simple means of a configuration that determines how the access operations are actually executed, making the best use of the underlying hardware.

Clearly, the configuration policy is at the centre of our approach. Thus a key question is: How is it defined and what is its utility? The intuition behind a configuration policy is that it partitions the resources (CPU cores as well as memory) of a given multi-socket machine into so-called *Virtual Domains*. This configuration policy is then used by the runtime system to route tasks submitted by client threads to the responsible virtual domains and send the results of a task back to the client. One could now think that this sounds very much like NUMA-aware processing strategies which modern DBMS engines implement already today to split the resources of a machine and partition the data structures accordingly.

However, NUMA-aware processing strategies solely split the resources based on the hardware topology [21, 29, 31, 36] (i.e., by sockets with their local memory or by single cores). But, they ignore many important aspects of the software stack on top, such as the characteristics of a given data structure and the workload which may (heavily) degrade performance. For example, as we show in Figure 1, when using write-heavy workloads for a modern tree-based index structure design that leverages Hardware Transactional Memory (HTM) of modern CPUs [27] we can see that, however, when more than half of the cores of a socket concurrently access the index structure, the performance degrades heavily due to aborting memory transactions.

In contrast to classical NUMA-aware processing strategies, our approach based on virtual domains allows to split the resources of a given machine in arbitrary granularity (e.g., into virtual domains that span only half a socket) in order to control contention for the data structures in an optimal manner. As shown in Figure 1, our flexible configuration strategy can provide superior performance across different workloads over the rigid partitioning strategies.

Outline: Section 2 discusses the basic intuition of how to provide robust performance for a real system which hosts many different data structures before we give an overview of our approach in Section 3. Sections 4 to 6 then present the details of our main building blocks. Afterwards, in Section 7, we present the evaluation showing the efficiency of our approach for different data structures as well as for executing a typical OLTP workload. To wrap up, Section 8 gives an overview of related work, and Section 9 concludes the paper.

2 The Art of Robust Performance

There exist many different causes for degraded performance of core data structures in main-memory databases on multi-socket hardware. In this paper, we focus on OLTP databases whose workloads are mainly characterised by different mixes of read and write statements ranging from read-heavy to write-heavy mixes where these operations are typically executed over index structures such as modern versions of B-trees or hash-tables. In the following, we first discuss the main causes of performance degradation and how current approaches handle them before we elaborate on our approach to robust performance by (re-)configuration.

2.1 Pitfalls of Rigid Architectures

The sources of performance degradation can be manifold. One primary reason that causes performance degradation of core data structures such as B-trees or hash-tables in main-memory OLTP databases is the overly high-contention that results from concurrent accesses (reads and writes) to the same instance of an index structure [36]. Other reasons for performance degradation include increased latencies as a result of cross-socket memory accesses or high cache coherence traffic resulting from concurrent reads and writes to the same memory [42]. In order to mitigate these effects, different strategies have been devised.

A prevalent strategy to address the aforementioned issues is (as discussed in the introduction already) to use a NUMA-partitioned DBMS design to mitigate the negative side-effects of cache coherence and increased latencies caused by cross-socket traffic [24, 30, 32]. However, this design can still lead to degraded performance since partitioning data structures at the granularity of a socket can also turn out sub-optimal leading to a too high contention for some data structures and workloads [10, 16], as we demonstrate in our experiments.

Hence, another direction that systems like H-Store [21] or Orthrus [36] suggest is to partition the database in an even

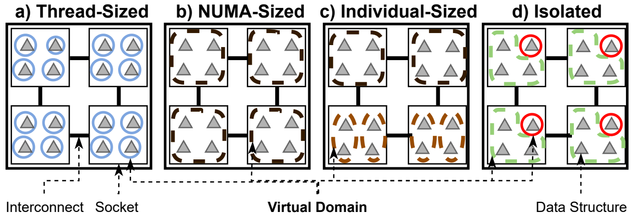

Figure 2: Flexible partitioning via configuration of *virtual domains* for a 4-socket machine: (a) *Thread-sized* with virtual domain per core. (b) *NUMA-sized* with a virtual domain per socket. (c) *Individual-sized* with two sizes of virtual domains. (d) *Isolated* with separate virtual domains for hot data structures.

finer-grained manner per hardware thread. While this design avoids performance degradation due to high contention, it has several other drawbacks such as its sensitivity to skew or the fact that more complex workloads cause an increased coordination overhead between partitions. Consequently, some systems such as Hekaton [11] even suggest avoiding partitioning and use a shared-everything approach instead, to mitigate the negative impacts of partitioning.

2.2 Robust Performance By Configuration

While all the afore-mentioned rigid partitioning strategies have their sweet spot(s), they can also cause severe performance degradation depending on the workload and data structures in use as we show in our experiments. In this paper, we thus propose a different route and suggest an approach enabling a flexible execution strategy that can adapt all these strategies ranging from thread-sized partitions to shared-everything by simple reconfiguration. The basic idea is that based on the mix of data structures and workload present in a concrete instance of a DBMS, we can provide a configuration using so-called virtual domains partitioning hardware resources in an optimal manner.

As shown in Figure 2 c) and d), virtual domains provide many more configuration options beyond what the rigid strategies (shared-everything or NUMA/thread-sized shared-nothing) can provide: First, when splitting the resources of a machine into virtual domains, not all virtual domains need to have identical sizes in terms of CPU or memory, but we can define virtual domains with different sizes to ideally support a mix of different data structures and workloads within a single system. Second, another configuration option provided by virtual domains is the isolation of hot data structures into separate virtual domains using a dedicated set of resources to enable more stable performance.

An important issue is that workloads in DBMS also might change over time and thus require reconfiguration of a hardware platform into larger or smaller virtual domains. At the moment, our approach handles this by offline reconfiguration, i.e., all active operations in the system must complete before a reconfiguration can be applied and the system can then restart with a new configuration. This offline approach can be used for reconfiguration if changes in workloads are known a priory or can be predicted based on reoccurring patterns (e.g., for Black Friday). In the future, we plan to extend our approach further to support online reconfiguration at runtime and thus also support cases where the workload changes are less predictable.

3 System Overview

In the following, we provide an overview of the main building blocks of our approach before discussing how to integrate our approach into a DBMS.

3.1 Asynchronous Tasks & Configurations

The two main building blocks an application needs to provide are asynchronous tasks implementing the access operations on data structures and a configuration that assigns data structures to optimally sized system partitions (virtual domains).

An asynchronous task is a container for an access method defined by the application, e.g., an insert or a lookup operation on a B-Tree. In contrast to operating system threads, tasks in our approach not only are much more lightweight but also are *data-aware*; i.e., a task is only executed inside the virtual domain where the data structure resides. This notion of tasks allows us to fully control contention and locality of access methods by simple means of a configuration.

In addition to tasks, the application can specify a configuration to control contention and locality of access methods for a given set of data structures. A configuration comprises two parts: (1) The first part of a configuration defines which *virtual domains* are being used to execute a given workload. Here the important aspect is the definition of how many domains are used and how resources are allocated to each virtual domain independent of the underlying hardware topology. (2) The second part of a configuration defines how data structure instances are mapped to virtual domains. Notably, an application may split a data structure into several instances and assign them to separate virtual domains to achieve higher throughput. In Section 5, we discuss an ILP-based approach to find an optimal configuration that maximises the overall system throughput given a workload and a set of data structures. According partitioning strategies for data structures are implemented by the application (i.e., the DBMS [1, 28]) on top of our runtime system. However, as we show in our experimental evaluation in Section 7 with our approach, DBMSs become less sensitive to the actual partitioning strategy being used since our approach seamlessly handles severe issues such as locality and contention.

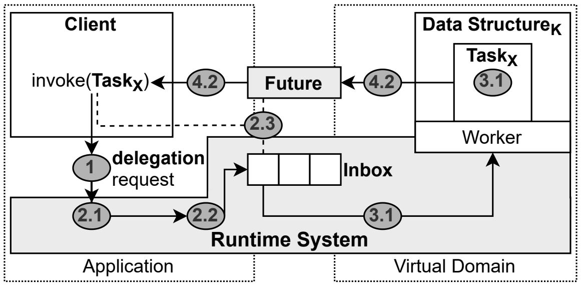

Figure 3: System Overview and Execution Flow: Client threads delegate *asynchronous tasks* to workers in a *virtual domain* which reply using *futures*. A configuration maps clients to virtual domains and workers.

3.2 Runtime System

The main objective of our runtime system is the efficient execution of tasks given a configuration. For efficient task execution, the runtime system provides a simple delegation mechanism based on highly optimised in-memory message passing. Noticeably, the aim of the runtime system is not to provide a full-fledged DBMS but to act as a thin *virtualisation* layer on top of the hardware providing the foundation for robust performance of a DBMS built on top. Below, we discuss the potential direction of how our runtime system can be integrated into a full DBMS.

Figure 3 presents an overview of our runtime system. A client thread submits an *asynchronous task* to be executed (step 1) and obtains an invocation handle, so-called *future*, on the submitted task (step 2.3) to consume the result of the task execution. Internally, the runtime system identifies the *virtual domain* responsible for the referenced data structure upon the invocation of an asynchronous task (step 2.1). It then places the task into the corresponding inbox (step 2.2) returning the future (step 2.3). For efficient message passing between virtual domains their inbox uses a fixed number of slots; details follow in Section 6.

The counterpart to the application's client threads are *worker threads* inside a virtual domain. These workers continuously poll the inbox for new tasks. Once a worker detects a new task (step 3.1), it executes the task (step 3.2) within the virtual domain on behalf of a client thread. Upon its completion, the *task* places its result in the earlier allocated *future* (step 4.1) from which the *client* retrieves the result (step 4.2).

3.3 Discussion of DBMS Integration

As mentioned before, the main contribution of this paper is not to provide a full-fledged DBMS. However, we believe that our delegation-based runtime system can be used for implementing a DBMS. In fact, we show in our experimental evaluation that we are able to execute typical OLTP workloads by implementing a "light-weight" OLTP engine on top of our runtime system. In the following, we discuss the main design choices involved in building an OLTP engine utilising our runtime system, though.

A first design choice for using our runtime system for OLTP is the mapping of transaction logic (i.e., the sequence of reads and writes) to tasks that can be executed by our runtime. A naïve way for this is to map every individual read/write operation of a transaction to a separate task to be submitted to our runtime system by the OLTP engine. Moreover, our programming model also allows more sophisticated implementations where transactions are chopped into sub-transactions and then are mapped to tasks as a whole. Studying the detailed effects of chopping is an interesting route for future work though. As we show in our experiments in Section 7.3, the naïve mapping already enables an efficient execution of OLTP.

A second design choice in addition to mapping transactions to tasks, is how tables of a database (and their indexes) are distributed across virtual domains. For this purpose, we introduce a configuration procedure in Section 5 that takes a set of data structures as input (i.e., the tables and indexes of a database) and compiles a configuration aiming to maximise the overall throughput for a given workload. Before applying this configuration procedure, the DBMS can still apply conventional partitioning strategies on tables as mentioned above and input these table partitions (as well as their indexes) as data structures to our configuration procedure.

In addition to these two main design aspects (i.e., mapping transactions to tasks as well as finding optimal configurations for a set of tables), further DBMS components need to be implemented, such as concurrency control as well as recovery mechanisms. The design of those components, however, is orthogonal to the contributions of this paper since many different schemes can be implemented on top of our runtime system. For instance, our runtime system allows DBMS to implement any concurrency control schemes ranging from pessimistic locking to various optimistic schemes. For our evaluation in Section 7.3, we hence omit these components for our "light-weight" OLTP engine as well as for all baselines (for a fair comparison), i.e., for concurrency control, we rely on latches to avoid data races but do not prevent other anomalies (e.g., lost updates). While this allows no direct comparison with other full-fledged DBMSs incorporating those components, it still allows us to compare the benefits of our execution scheme for OLTP workloads compared to more classical OLTP engine designs where data is partitioned by NUMA regions and transaction managers directly execute operations without delegation.

Finally, an interesting future aspect when designing an OLTP engine on top of our runtime system is that the asynchronous execution model opens up many new opportunities for optimisations. For example, in a classical design of an

OLTP engine, transaction manager threads execute only a single transaction at a time, whereas a design building on our runtime system could rethink this model allowing transaction manager threads to execute operations on behalf of multiple transactions at the same time. That is, when an operation of one transaction is submitted to our runtime, the transaction manager could submit operations on behalf of another transaction instead of blocking until the results of the first transaction are available. However, analysing these optimisations is beyond the scope of this paper and needs a more thorough investigation in our future work.

4 Programming Model

In this paper, we propose a new approach for task-based programming. While asynchronous task-based programming is not new and has also direct support in different programming languages such as Erlang and C# [2, 14, 35, 40, 41] as a lightweight alternative over threads, we propose a novel abstraction called *asynchronous data-aware tasks.*

The essential aspect of a data-aware task is that it only allows accessing a data structure within a single *virtual domain* using precisely the configured resources of that domain. Therefore, a data-aware task must be executed by a worker thread inside a virtual domain. This concept allows us to control not only the degree of contention by simply re-configuring a virtual domain (i.e., by changing the size of the domain and thus the number of worker threads that have concurrent access) but also other transient properties such as cache state and cross NUMA-node traffic which are bound to the virtual domain as well.

Listing 1: API of an *Asynchronous Task.*

```
class Task {
    Task(void* dataStructure, Args... args)
    void operator()(Result& res);
};
```

In order to implement a data-aware task, the outlined API of a task (see Listing 1) only requires the first parameter of the constructor to be the targeted data structure and to implement the function *operator()* to return results of the task using the *Result* object. Additionally, this abstraction must also encapsulate all input parameters for the contained operations. In combination, this simple API enables the runtime system to route the task to the corresponding virtual domain based on the referenced data structure.

To show that this programming model can also be used to implement typical operations of a transaction, the example in Listing 2 implements a task to insert a record into a (partition of a) table. While Listing 2 is a simple and illustrative example, the DBMS could also use tasks to implement more complex operations involving several data structures within the same virtual domain or fusion of several operations.

Listing 2: A task to insert a record into a table.

```
class TaskInsertRecord {
    Table* tab; // Pointer to table
    Record* rec; // Pointer to buffer of record
    TaskInsertRecord(Table* table, Record* record):
        tab(table), rec(record){};
    void operator()(Result &res){
        // Read buffer and insert record
        RowID rowID = tab->insert(*rec);
        delete rec; // Delete buffer
        res.set(rowID); // Return inserted row id
    }
};
```

5 Robustness by Configuration

The main aspect for configuration is the definition of virtual domains partitioning the resources of a given hardware platform. In this section, we define the configuration options of virtual domains before we outline the process of how to find a configuration for a workload and the set of data structures comprising the overlaying application.

5.1 Virtual Domains

A *Virtual Domain* is defined as a set of dedicated logical (SMT) cores, a worker thread placement policy (i.e., if it allows thread migration or requires strict pinning to cores), and a memory allocation policy (e.g., strictly local to individual workers or interleaved across all workers). In this regard, we *virtualise* NUMA-regions which directly represent the hardware topology into flexibly configurable regions. We establish these *virtualised* hardware regions as *domains* to control worst-case contention and data locality, thus the name *virtual domains.*

In particular, only worker threads of a virtual domain are allowed to execute tasks on the data structure instances assigned to that virtual domain, and thus, no side effects can cross its boundaries. Moreover, since all operations on the data structure instance are executed as tasks by dedicated workers of a virtual domain, cache state exclusively resides in the respective CPUs (cache locality) and only these CPUs synchronise for cache-coherence on that cache state [20].

Consequently, virtual domains limit the *(1) worst-case contention* of tasks in a virtual domain to the number of worker threads and *(2) worst-case locality* using the placement policies. Thus, virtual domains provide configurable contention control and locality which can be flexibly specialised for distinct data structure instances expose to diverse conditions through co-existing virtual domains in a single system.

5.2 Configuration Process

Having the means to control contention and locality of distinct data structure instances through virtual domains, the

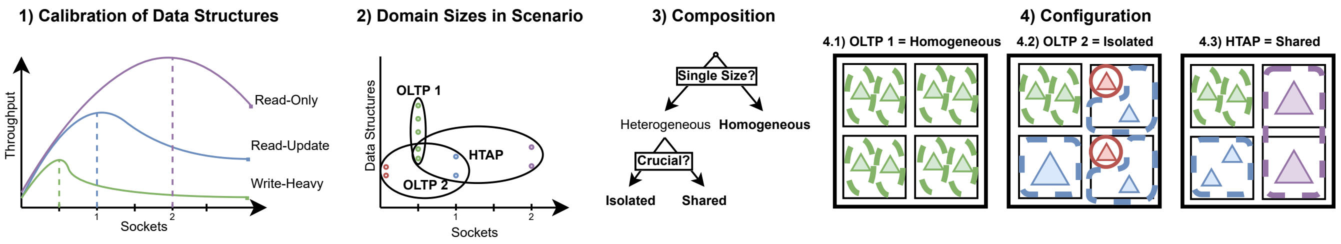

Figure 4: Configuration Process: 1. Calibration of domain sizes for the best trade-off between contention and locality. 2. Optimal domain sizes for different scenarios (OLTP1, OLTP2, HTAP) based on calibration. 3. Composition of virtual domain as *homogeneous* or *heterogeneous* configurations. 4. Resulting configurations for exemplary scenarios.

configuration process is about finding the individually optimal domain sizes and their composition into a single configuration. The overall process of finding a configuration that defines which virtual domains should be used and how data structures are mapped into the domains is shown in Figure 4.

The first step of the process (step 1, Fig. 4) is a calibration phase that gathers performance metrics and quantifies the performance behaviour of the different data structures involved in a workload. The goal is to find the optimal domain size for each data structure instance involved in that workload individually (step 2, Fig. 4). Subsequently, we start the composition process (step 3, Fig. 4) which, based on the calibration information, divides the system resources and maps data structure instances into virtual domains to produce a configuration (step 4, Fig. 4).

Notably, the configuration does not partition the data structures themselves. Instead, we expect the application to partition the data structures utilising application-specific knowledge (e.g., partitioned indexes in a DBMS) while the goal of the configuration process is to find an optimal assignment of those partitions to virtual domains. For finding an optimal configuration for these partitioned data structures, the application can define constraints which data structure instances should be mapped into the same virtual domain to realise co-location of data structure partitions (e.g., an OLTP DBMS could co-locate data of several tables in one virtual domain avoiding transactions across virtual domains).

Calibration of Domain Sizes: The calibration phase executes a given workload under growing domain sizes (i.e., with an increasing number of threads) for each data structure instance individually. This calibration typically results in a common throughput pattern as sketched in step 1 of Figure 4. The reason is that with increasing domain size the contention increases and locality is getting worse if domains span multiple NUMA nodes. As a result of the calibration, we derive the domain size maximising the overall performance up to the point after which the slope of the throughput becomes negative. As sketched in step 1 of Figure 4, the domain size providing maximum performance is typically larger for read-heavy workloads than for write-heavy workloads: that is $\frac{1}{2}$ socket for a write-heavy workload and 2 sockets for a read-only workload in our example.

Since the calibration phase determines the domain sizes for individual data structure instances, it ignores side-effects that might occur when several data structure instances share a virtual domain. Since sharing a virtual domain means sharing its worker threads, contention on individual data structure instances may only decrease, hence does not violate contention control. In contrast to contention, locality gets worse when multiple data structures share the same domain since CPU caches are also shared. Our composition process (discussed next) thus aims to balance the load equally across all virtual domains such that the negative effect of decreased locality is equally distributed across domains.

Composition of Domains: We now discuss the second step of the configuration process deciding the composition of virtual domains. In this step, we partition the hardware resources and assign data structure instances given from the application to individual virtual domains. In the following, we use three typical workloads as examples to explain the composition approach: (1) *OLTP 1* as a typical OLTP scenario where indexes are accessed with a write-heavy workload; (2) *OLTP 2* as a mixed OLTP scenario where indexes are accessed with a mix of write-heavy and read-update statements; (3) *HTAP* as an HTAP scenario where indexes are accessed with write-heavy, read-update, and read-only statements.

As shown in step 3 of Figure 4, we distinguish two high-level cases for the composition: (1) *homogeneous* and (2) *heterogeneous* composition.

The *homogeneous* composition applies, when the calibration indicates a single optimal domain size for all data structures instances (as for OLTP1). Hence, a configuration may coincide with state of the art, e.g., Shared Nothing partitioning schemes, but it may also yield better-performing configurations, e.g., half a socket instead of a full socket as shown with configuration 4.1 in Figure 4 for *OLTP1.*

The second case (*heterogeneous* composition) applies if the calibration shows the data structure instances require different domain sizes for a particular workload; e.g., in HTAP workloads some data structures are used in a read-heavy manner and can use larger domains while others are write-heavy and thus need smaller domains. In this case, we differentiate the *isolated* and *shared* heterogeneous composition: (1) *Isolated* is used for crucial data structure instances necessitating predictable performance (e.g., a lock table where latency matters). The idea behind isolation is that these data structure instances do not share a virtual domain with other data structure instances. Configuration 4.2 in Figure 4 demonstrates *isolation* with thread-sized domains for two crucial indexes (red) in the OLTP2 workload. (2) For all other data structure instances, we apply the *shared* heterogeneous case composing domains of different sizes which can be shared by multiple data structure instances as shown in configuration 4.3 (Figure 4) for the exemplary HTAP workload.

For the *shared* heterogeneous composition, we formulate the problem as a variation of a General Assignment Problem with Minimal Quantities (GAP-MQ) [23] in form of an Integer Linear Program (ILP). Intuitively, the ILP should fulfil the following goals: (1) Most importantly, data structure instances should reside in domains of at most the calibrated optimal domain size. (2) The number of domains should be minimised because a higher number of domains increases the sensitivity to skew. (3) The load between all domains should be balanced.

For the input of our ILP, we introduce the data structure instances of an application as n data structure instances $i \in I = \{1, ..., n\}$ with calibrated optimal domain sizes $s_i \in S \subseteq \mathbb{N}^+$. Further, we specify the number of available worker threads in the system as $w \in \mathbb{N}^+$. Then we define the multiset B as all possible domain sizes comprising any potential configuration within the limits of the given workers w where each domain size $s \in S$ appears $\lfloor w/s \rfloor$ times (multiplicity of s). For example, assuming 192 workers as a system size ($w = 192$) for our *OLTP2* scenario in Figure 4 and optimal domain sizes of $S = \{24, 48\}$ that we identified by calibration, the multiset is $B = \{24_1, 24_2, ..., 24_8, 48_9, 48_{10}, 48_{11}, 48_{12}\}_b$. Based on B, the domains to choose for a configuration are $d \in D = \{1, ..., |B|\}$ with domain size $b_d \in B$, where binary variables y_d indicate the choice of d. In our *OLTP2* scenario, this could be the choice $y_1, y_2, y_9, y_{10}, y_{11} = 1$, i.e., 2 domains of size 24 and 3 domains of size 48. Subsequently, the binary variables $x_{i,d}$ denote the assignment of a data structure instance i to a domain d in the resulting configuration.

For load balancing, we assign an abstract expected load of an instance as $l_i \in \mathbb{R}^+$ as well as a minimum and maximum load of a domain as q_d and $r_d \in \mathbb{R}^+$, where the minimum load avoids domains without any load while the maximum load avoids overloading domains. Finally, we incentivise choosing larger domains by assigning the profit in proportion to the domain size as $p_d = P^{b_d}$ with a large P, s.t. $p_1 \ll ... \ll p_{|D|}$.

$$
\begin{aligned}
\max \quad & \textstyle\sum_{d \in D} p_d y_d & & (1)\\
\text{s.t.} \quad & n y_d - \textstyle\sum_{i \in I} x_{i,d} \le n - 1, & \forall d \in D \quad & (2)\\
& \textstyle\sum_{d \in D} x_{i,d} = 1, & \forall i \in I \quad & (3)\\
& b_d x_{i,d} \le s_i, & \forall i \in I, \forall d \in D \quad & (4)\\
& \textstyle\sum_{d \in D} b_d y_d \le w & & (5)\\
& q_d y_d \le \textstyle\sum_{i \in I} l_i x_{i,d} \le r_d, & \forall d \in D \quad & (6)\\
& x_{i,d}, y_d \in \{0, 1\}, & \forall i \in I, \forall d \in D \quad & (7)
\end{aligned}
$$

Equations 1-7 formulate the ILP for our configuration problem based on the GAP-MQ problem. The objective function formalises an optimal configuration as a choice y_d of domains d maximising the profit through large domain sizes and consequently a minimal number of domains, where the constraint in Equation 2 connects that choice of a domain to the assignment of data structures $x_{i,d}$. The constraint in Equation 3 requires the assignment of each instance to precisely one domain. Equation 4 constrains the assignment of an instance to domains of at most the calibrated optimal domain size to satisfy the calibrated worst-case contention and locality while Equation 5 restricts the choice of domains to the available workers. Finally, in Equation 6, we constrain the assignment of instances to domains, such that the sum of the load of a domain is within the required bounds if the domain is chosen. Solving this ILP determines y_d and $x_{i,d}$ establishing a configuration of domains with assigned instances for our runtime system. Additionally, our ILP can simply reflect application-specific requirements on the configuration by additional constraints. For example, further constraints can incorporate co-location of specific data structure instances to place secondary indexes into the same domain.

6 Runtime System

Given a configuration, our runtime system realises efficient execution of *data-aware asynchronous tasks* on generic data structures in freely configurable *virtual domains* via *delegation* and *futures*.

Efficient and Flexible Delegation: In order to achieve robust performance with an optimal configuration via delegation, the communication between clients and workers must be as efficient as possible, especially it should not cause contention which we seek to reduce through optimal configuration.

Therefore, we implement the delegation as efficient in-memory message passing based on *fast, fly-weight delegation* (FFWD) [37]. At the core of FFWD is a message passing scheme minimising cache coherence traffic for synchronous communication between multiple clients and a single worker. It enables highly efficient communication outperforming common concurrent data structures with shared

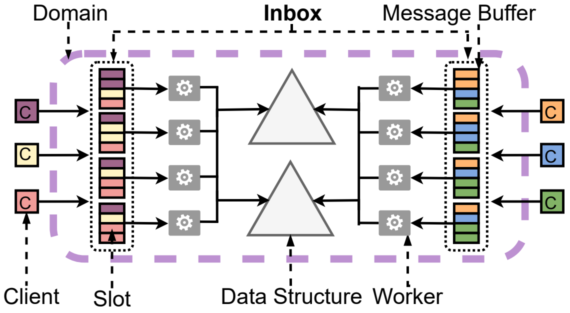

Figure 5: Flexible delegation through an *inbox* constructed from message buffers of workers in which clients obtain ownership of slots (colour coded).

memory synchronisation primitives and latch-free designs, e.g., queues with NUMA-aware *MCS* latches and latch-free queues. In detail, FFWD allocates a contiguous message buffer for the worker in which each client has a dedicated slot for a message at the position of the client id. Then, FFWD minimises cache coherence traffic through efficient detection of new messages via embedded toggle bits and batching of responses for up to 15 clients. Furthermore, their design simply includes common optimisations, e.g., NUMA-aware memory allocation as well as memory alignment to 128 bytes to prevent false sharing of adjacent cache lines [20] and incorporates optimisations not generally possible for concurrent data structures, i.e., complete absence of atomic instructions and memory ordering fences.

Beyond the original FFWD, we extend the messaging scheme to reach the necessary flexibility for our approach of optimal configuration. Specifically, we break the strong relation between a client and a worker in FFWD while maintaining the same optimisations. Figure 5 outlines how we establish an *inbox* for a virtual domain from which clients obtain ownership of slots to delegate to and physically construct this inbox out of the message buffers of the configured workers. Consequently, clients are only loosely coupled with (workers in) virtual domains enabling any number of clients to be transparently serviced by the independently configured number of workers within a virtual domain (limited by the total number of slots of the inbox). Additionally, we enable asynchronous delegation of several tasks to virtual domains via futures by handling responses to delegated tasks and returning ownership of a slot after returning the results.

As optimisation for virtual domains spanning multiple NUMA nodes (e.g., two sockets), the runtime system assigns ownership of a slot in the inbox, such that the backing worker has minimal NUMA distance to the requesting client. For example in Figure 5, the purple client on the left gets assigned ownership of the purple slots from message buffers on the left from the inbox of a virtual domain spanning two sockets and vice versa for the orange client on the right. Thereby both clients communicate locally with workers instead of communicating through an interconnect.

Notably, the implementation of delegation across virtual domains puts little requirements on the underlying hardware platform. For example, delegation can be implemented in NUMA systems with access to shared memory (which is our main focus in this paper) but can also be used in distributed systems with RDMA or future systems like Gen-Z [17]), which we aim to study in future work.

Optimised Delegation Mode(s): On top of the efficient communication scheme, we introduce enhanced delegation that allows clients to asynchronously delegate numerous tasks and only eventually request their results which we utilise to optimise task delegation for bursting behaviour.

We enable the client to announce bursting delegation for a specific data structure instance to the runtime system. Then, the runtime system pre-allocates futures and slots from the inbox of the according virtual domain for a maximum number of outstanding tasks (burst size) specified by the client, thereby we clear the critical path from resource allocation and minimise the overhead for delegation in bursts. Moreover, we provide a delegation mode to maximise throughput based on these bursts. The runtime system can manage a burst for the client in a way that the client can continuously delegate independent tasks and only needs to process the result of the oldest tasks when the burst is completely filled. This delegation mode maximises overlap of pending tasks in addition to minimising the overhead and consequently maximises delegation throughput for the client to the specified data structure instance. Additionally, further extended delegation modes are possible to cover other application-specific delegation patterns. With *bulk bursting*, for example, multiple tasks are delegated under a single synchronisation phase. This mode optimises delegation for a fixed number (i.e., bulk) of parallel operations within a transaction requiring a common synchronisation point.

7 Experimental Evaluation

In the following experiments, we evaluate the efficiency of our approach and illustrate its robust performance by reconfiguration for a range of data structures and workloads.

Baselines and Setup: As baselines, we consider a wide range of fixed partitioning strategies from naïve shared everything to extreme shared nothing: (1) *SE* and *SE-NUMA* represent shared everything strategies, where all threads access all data structure instances. The former is a naïve setting which solely relies on the OS for data placement of its partitioned data structures into NUMA regions. Whereas the latter (*SE-NUMA*) setting is NUMA-aware, but only for memory allocations of the individual partitions. However, all threads are still allowed to operate on all the partitions, i.e., execution is not NUMA-aware. (2) *SN-NUMA* and *SN-Thread* correspond to state of the art shared nothing strategies [31], which we apply to the configuration of data structures in our framework. *SN-NUMA* represents NUMA-aware system partitioning that explicitly dedicates data structures to specific NUMA nodes. *SN-Thread* is an extreme shared nothing strategy, with thread-granular partitioning, where a single thread is exclusively accessing a partition of a data structure.

We compare all these baselines against our approach (*Opt. Configured*), where we use an optimal partitioning strategy that results from applying our configuration process in Section 5. In all our experiments, we use the bursted execution mode with a burst size of 14 for our approach. While the burst size is a configuration parameter, this size has shown on average the best performance across all experiments with only a minimal increase in latency. For both *shared everything* strategies bursting cannot be applied since clients directly access data structure instances.

Hardware: We conduct our experiments on an *HPE MC990 X* system with two hardware partitions each containing four *Intel Xeon E7-8890 v4* CPU (24 cores, 60MB L3), i.e., 192 physical cores and 384 logical (SMT) cores (with HyperThreading). A *NUMAlink* controller combines these hardware partitions to a single, cache-coherent NUMA system [15]. The resulting system has four levels of NUMA, for which we measure memory latencies of 114, 217, 265, and 487ns. In order to assess the robustness for different hardware architectures, we use this system to simulate different architectures by restricting the number of sockets ranging from small-scale NUMA systems connected via one hop to large-scale NUMA systems that need to cross the NUMAlink.

7.1 Exp. 1: Efficiency for Various Data Structures and Workloads

In the first experiment we show the ability of our approach to enable robust performance across a wide range of data structures and workloads.

Workloads and Metrics: To assess the performance of different data structures, we use YCSB [8]. We use workloads A (Read-Update 50/50), C (Read-Only), and D (Read-Insert 95/5) of YCSB. These workloads allow us to investigate the performance of a data structure with increasing contention due to the varying amount of modifications (inserts or updates). We changed the distribution of workload D from *Latest* to *Zipfian* to keep the distribution of records and operations identical across all three workloads for direct comparison. Moreover, we use records of 64-bit integer keys and values, which potentially allow more caching but also may cause higher contention. This allows us to evaluate complex effects of locality and contention in both software and hardware.

Table 1: Data structures employed in experiments with specifics about their synchronisation scheme.

Data structure	Synchronisation scheme
STX B-Tree [3]	none by default, modified: atomic load/store + global lock for inserts
FP-Tree [27]	HTM + global lock for fallback
Open BW-Tree [42]	Copy-On-Write + atomic CAS
Hash Map [39]	Fine-grained locking + spin lock

Table 2: Optimal size (no. of workers) of virtual domains for data structures and workloads.

Workload	Read-Only	Read-Update	Read-Insert
B-Tree	48	24	24
FP-Tree	48	24	24
BW-Tree	48	48	48
Hash Map	1	1	1

We define the number of records as ten times the cumulative last level cache size of all sockets in the hardware mentioned before resulting in 314 M records. We generate the complete workload (records and operations) through the official Java implementation [4, 8, 19], which we then simply replay using our C++ based prototypes. For each experiment, we execute 2M key/value-operations per client thread.

Experimental measurements are presented as the median out of seven executions. We assess the reliability of our measurements in terms of Coefficient of Variation (CV) (ratio of standard deviation to mean) and consider a CV $\leq 5\%$ as reliable. Since all our measurements fit this reliability requirement, we do not present error bars in our plots.

Data Structures: In all experiments, we use a set of data structures commonly used for indexing in main-memory DBMS with different synchronisation schemes, listed in Table 1[1].

In the following, we evaluate the performance using an optimal configuration for each of the before-mentioned data structures and workload and compare it to rigid approaches ranging from Shared Everything to fine-granular Shared Nothing configurations. The overview of data structures and workloads we used in this experiment is shown Table 2.

7.1.1 Exp. 1a - Performance for Various Data Structures

We begin our evaluation by applying the optimal configuration as described in Section 5 to the largest system size (i.e., a machine with 8 sockets) and investigate the throughput of the index data structures under all workloads. Figure 6 shows,

[1] During our experiments, we discovered skew in the hash of the *Hash Map* and extended it with an additional XOR of the upper half of the key with the lower half resulting in a more even occupation of hash buckets, e.g., standard deviation of bucket size 1.2 instead of 4.7.

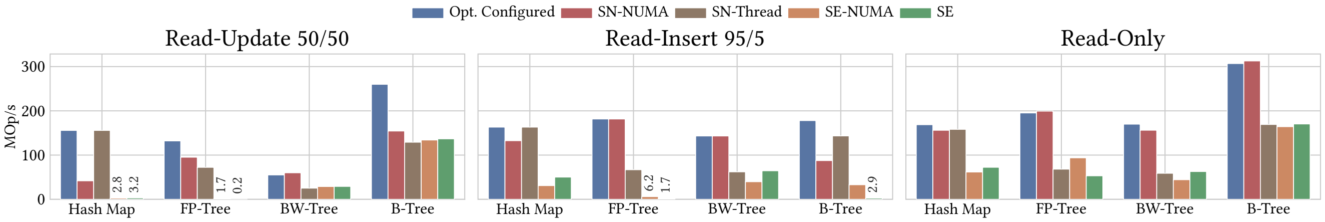

Figure 6: Performance of our approach across a range of YCSB workloads and data structures on largest system size (8 sockets) through individually optimal configuration. Baselines are rigid partitioning schemes.

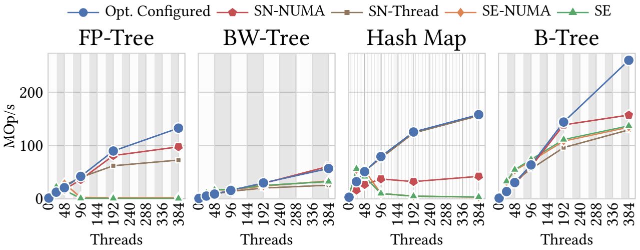

Figure 7: Throughput of read-update workload for various system sizes from 1 - 8 sockets (each 48 threads).

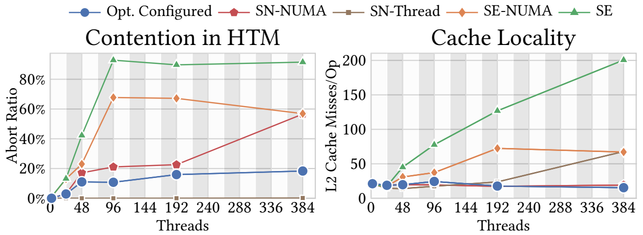

Figure 8: Hardware metrics indicating contention and cache locality of FP-Tree on read-upadte workload.

that *Opt. Configured* reaches the best throughput for all data structures under various workloads ranging from read- to write-heavy. Moreover, we see that there is no single rigid approach that dominates all other rigid approaches under all workloads and data structures, e.g., *SN-Thread* performs well with *Hash Map* but significantly worse with *FP-Tree* and *BW-Tree* whereas for *SN-NUMA* the opposite is the case.

Insight: Configuration of individually optimal domain sizes yields robust (best or close to best) throughput for any of the evaluated index data structures and workloads.

7.1.2 Exp. 1b - Robust Performance for Various System Sizes

In the following experiment, we examine the performance of our approach on different system sizes; i.e., by varying the system size from 1 up to 8 sockets of the machine outlined in the beginning of this section. We illustrate the resulting number of virtual domains used across different system sizes with alternating white and grey shadings in the plots, e.g., the first shading represents the first virtual domain, the second shading represents the second virtual domains.

Read-Update Workloads: For this experiment, we first asses the effect of configuration on different system sizes with an equal mix of reads and updates. The updates are in-place modifications to index records, which do not cause any maintenance, such as node splits in a tree. Thus, these are of high locality, providing an opportunity for low contented, efficient synchronisation. Still, the high update rate puts pressure on synchronisation and causes physical contention.

Figure 7 depicts the throughput of the read-update workload for the same 4 index data structures as in the previous experiment. Our *Opt. Configured* provides robust scalability on the read-update workload for all the data structures, i.e., best or close to best throughput at each scale. The *Shared Everything* settings scale only with the *B-Tree* and *BW-Tree*, whereas *Shared Nothing* settings at most perform as good as our *Opt. Configured*.

While providing robust performance across all data structures, with *FP-Tree Opt. Configured* even improves performance by 560x over *SE*, 1.8x over *SN-NUMA*, and 1.4x over *SN-Thread* at 384 threads. Specifically, both *Shared Everything* settings stagnate after 24 threads and significantly drop in performance for larger system sizes, i.e., performance collapses by over 90% between 1 and 2 sockets. Instead, our *Opt. Configured* setting scales best because it retains the best scale-up performance of 24 threads in virtual domains and efficiently scales these to the largest system size. The other settings either insufficiently limit contention for HTM to perform well or incur much overhead, as detailed below.

For better understanding of the root causes for the performance degradation and the lack of scalability, we analyse the abort rate of HTM transactions and cache locality presented in Figure 8. The performance of *Shared Everything (SE)* settings and *SN-NUMA* setting is tied to the sensitivity of HTM to high conflict ratios (i.e., workload with 50% updates) and length of HTM transactions, amplified by longer NUMA distances, as investigated in [5]. The consequence is high abort ratios of HTM transactions inflicting the substantial performance degradation for these three settings. In contrast, *SN-Thread* does not cause any aborts, but increased L2 (and

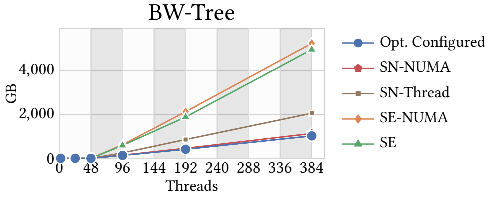

Figure 9: Communication volume on interconnects between sockets for BW-Tree on read-update workload.

L3) cache misses instead, thus indicating overhead of its extreme partitioning which inflates competition between the data structure and the delegation procedure for the private L2 cache of the responsible CPU core. Finally, *Opt. Configured* keeps the abort ratio and cache misses low, such that it performs well for the contended read-update workload. This confirms the benefits of our apt configuration with adequate contention management yet minimal overhead.

In the contrary to FP-Tree, *BW-Tree* manages to scale with the *SE* settings due to its conflict resistant Copy-On-Write (COW) synchronisation scheme but the performance of *Opt. Configured* is superior at larger scales, i.e., up to 1.9x. Figure 9 shows that the COW synchronisation scheme induces high communication overhead on the interconnects of up to 5 TB. Here, our the efficient in-memory messaging pays off with about 5x lower communication volume for *Opt. Configured* and *SN-NUMA* as well as 2.5x less for *SN-Thread*.

The *Hash Map* exhibits a behaviour similar to the *FP-Tree* under *SE*. For the smallest deployments up to a single socket, SE provides high performance. However, for larger deployments the performance of *SE* collapses similar to the FP-Tree. Our profiling analysis of the *Hash Map* indicates that the bottleneck is highly contended synchronisation. This high contention also explains the mediocre performance of SN-NUMA, whose partitioning per NUMA region insufficiently controls contention. Therefore, SN-Thread and the Opt. Configured provide robust performance for the *Hash Map* when scaling to larger deployments. This highlights the benefit of our configurable approach, which allows us to partition data structures into optimally sized domains: for the *Hash Map*, our approach uses many small domains similar to SN-Thread, while for the other data structures before (FP-Tree, BW-Tree) we use a configuration that is closer to the SN-NUMA.

Finally, for *B-Tree Opt. Configured* performs as good as the NUMA-partitioned strategy. However, we use synchronisation with just atomic operations on the record level since the B-Tree itself does not include any synchronisation. This synchronisation is unfair as it does not protect modifications of the structure of the *B-Tree*. Hence, this mainly serves as an upper bound for possible performance with the simplest synchronisation that we could achieve with all strategies.

Insight: Optimal configuration establishes robust scalability for indexes under high contention. In contrast, both

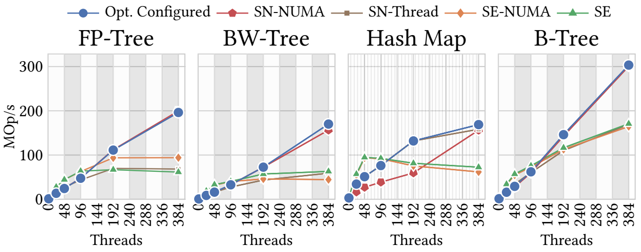

Figure 10: Throughput of read-only workload for various system sizes from 1 - 8 sockets (each 48 threads).

Shared Everything settings suffer steep performance cliffs beyond one socket already and the *Shared Nothing* approaches scale well just for some indexes at larger scale. Only our configurable approach (*Opt. Configured*) handles contention effectively for all data structures providing locality at different system scales.

Read-Only Workloads: Next, we show the effect of configuration on a read-only workload (YCSB C) for the same set of data structures and system sizes. This workload is favourable for a *Shared Everything* strategy, as there is little to no contention and maximum opportunity for high cache utilisation. Therefore, we expect the benefits of our approach over Shared Everything to be limited to the better locality in case of memory accesses or remaining synchronisation, e.g., reader side of latches.

Figure 10 presents the experimental results. For *FP-Tree* the *SE* as well as *SN-Thread* settings scale only up to 96 threads (2 sockets), after which their throughput stagnates and *SE-NUMA* follows stagnating only after 4 socket. In contrast, *Opt. Configured* and *SN-NUMA* manage to scale linearly up to 8 sockets with a maximal throughput improvement of 3.2x over *SE*. Only our approach (and SN-NUMA) provides efficient execution of access methods even for large deployments. Moreover, *BW-Tree* and *B-Tree* present similar behaviour to *FP-Tree*, only that *BW-Tree* is slightly slower and *B-Tree* is faster due to their differing synchronisation.

The data structure *Hash Map* performs well only on a single socket with *SE*, whereas *Opt. Configured* enables robust performance of *Hash Map* reaching 2.3x higher throughput than *SE* at 8 sockets. This improvement results from a bottleneck in the general-purpose implementation of the *Hash Map* rooted in the reader coordination of the reader-writer mutex for synchronisation on the hash buckets. The locality within our virtual domains optimises the execution of the atomic increment to register readers on the mutex.

Insight: For read-only workloads where contention is not as prevalent as for the read-update workload before, our approach (Opt. Configured) shows competitive performance as well as low overhead for all index structures. Most importantly, Opt. Configured again is the only approach that

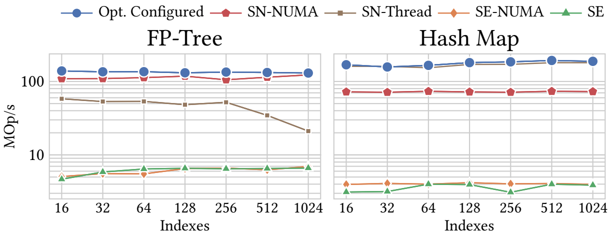

Figure 11: Agg. throughput for increasing no. of indexes (i.e., application size) for read-update workload.

can provide robust performance for all data structures when scaling out: while Opt. Configured performs on par with SN-NUMA offering the best performance for the 3 tree-based data structures, Opt. Configured employs smaller domain sizes for the Hash Map, and thus behaves more like SN-Thread, which is best performing in this case.

7.1.3 Exp. 1c - Robustness for Different Application Sizes In the following experiment, we extend the perspective of robustness by application size using an increasing number of index instances. As the system is under full load even with a single index instance already, there cannot be major improvement of throughput when increasing the number of index instances. On the contrary, we expect this experiment to expose bottlenecks, as it may amplify overheads or impact important performance factors such as cache locality or contention inherent in our framework.

We setup this experiment as previously but increase the number of indexes by separating the prior indexes into smaller indexes (16 - 1024) with the identical total data volume. Moreover, we configure our framework with the same number of optimally sized virtual domains, i.e., 16 domains of size 24 threads, but additionally now instances share domains, e.g., for a total of 1024 indexes 64 instances share one domain.

Figure 11 presents stable throughput under increasing number of indexes for most settings with both index types. Exceptions are on FP-Tree a minor positive trend for both shared everything settings (SE: 1.4x, SE-NUMA: 1.3x) and degrading of SN-Thread by up to 50% beyond 256 indexes. Importantly, the individual configuration of indexes within our framework (Opt. Configured) is stable and provides the best throughput for all numbers of indexes.

Insight: The benefits of configuration within our framework persists for large numbers of indexes.

7.2 Exp. 2: Cost-Benefit Breakdown of Configurability

Having demonstrated the potential of configurability in Section 7.1, we now detail its associated overhead. Indeed, our runtime system, which delegates tasks to virtual domains, causes computational overhead in addition to baselines which directly execute the access methods. However, as we show in the following, this overhead is negligible even for small system sizes and provides significant benefits for robust performance, especially when scaling to larger deployments.

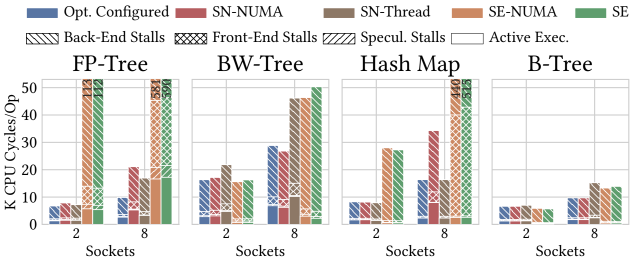

Figure 12: Execution cost breakdown into active execution cycles vs. stall cycles per operation for system sizes 2 vs. 8 sockets with read-update workload.

Workload and Baselines: In this experiment, we use the *YCSB Workload A* (Read-Update) from the previous experiment to show the overhead and benefits of workloads with a mix of operations that is common for OLTP workloads. Moreover, we run this experiment on a small system size (2 sockets) and the largest system size (8 sockets) to show the cost breakdown for small versus larger systems as mentioned before. For each system, we compare the same data structures and baselines as in Section 7.1.

Performance Results: Figure 12 shows the average cost for one operation (read/update) in *CPU cycles per operation* as stacked bars for the different data structures and system sizes. The cost is broken down into the main categories of the *Top-down Microarchitecture Analysis Method* (TMAM) [20] commonly used to guide performance optimisation [38] (e.g., by Intel VTune). Based on TMAM, we distinguish active execution cycles (solid part of bars) and wasted execution time (striped parts of bars), i.e., stalls in the *Back-End* of the CPU (mainly memory accesses), in the *Front-End* (e.g., instruction decoding), and stalls due to bad speculation (e.g., branch misprediction). Notably, in this representation, *lower cost means better throughput per thread.*

Again, in this experiment, we can observe robust performance (i.e., lowest cost or close to lowest cost) for our approach (Opt. Configured) compared to the baselines. That is, as expected for the small system size (2 sockets), our runtime system has comparable performance to the shared-nothing and shared-everything baselines, while we achieve significant benefits for larger system sizes, especially for the FP-tree. Moreover, as observed before, Opt. Configured achieves its robustness by configurability: it resembles the execution cost of SN-NUMA or SN-Thread (which are the best performing) while its distinct configuration of half a socket (i.e., in between the partition sizes of those other approaches) even achieves better cost for *FP-Tree*.

Finally, when observing cost in detail, we can confirm our runtime system adds negligible active execution overhead in comparison to the SE-baselines, which can be mainly attributed to the additional instructions for delegation on top of the bare data structure operations. However, comparing the amount of stall cycles, our runtime efficiently executes these additional instructions (comparable *Front-End* and *Speculation Stalls*) and effectively decreases overall memory (*Back-End*) stalls below the stalls of the bare data structure operations (i.e., improves locality and contention), which benefits overall cost per operation.

Insight: Our cost-benefit analysis shows that our approach (Opt. Configured) has highest active cycles stemming from the additional overhead of our runtime system. However, importantly, this allows our approach to reduce stalls efficiently. As a result, our approach has the overall lowest execution cost (active cycles + stalls), thus can provide the highest performance across different data structures and system sizes.

7.3 Exp. 3: Efficiency for OLTP Workloads

In the last experiment we show that our approach also provides performance benefits beyond single data structures and supports the efficient execution of more complex OLTP workloads. For this purpose, we implement a light-weight OLTP engine on top of our runtime system and observe the performance of transactions of TPC-C – a typical OLTP benchmark.

Light-weight OLTP Engine and Baseline: The light-weight OLTP engine provides basic functionality for partitioning tables and executing transactions as tasks: (1) For partitioning, our light-weight OLTP engine supports a typical hash-partitioning scheme. In particular, it partitions tables including their primary and secondary indexes that are then configured as composite data structure by our configuration procedure from Section 5 to place tables and their indexes jointly in the same virtual domains. As data structures for tables and indexes, we have chosen the *FP-Tree* and the *BW-Tree* from the previous experiments since both are specifically designed for supporting OLTP workloads in main memory DBMS, notably they use very different synchronisation mechanisms (cf. Table 1). (2) For executing transactions as task, our OLTP engine uses a scheme where each individual statement of a transaction is mapped to a task (i.e., we do not do any chopping which could further improve the performance). Finally, we omit higher order components for recovery and concurrency control. As already discussed in Section 3, all these components are orthogonal and different schemes can be implemented on top of our runtime system. Analysing the effects of these different schemes (e.g., for concurrency), however, is beyond the scope of this paper.

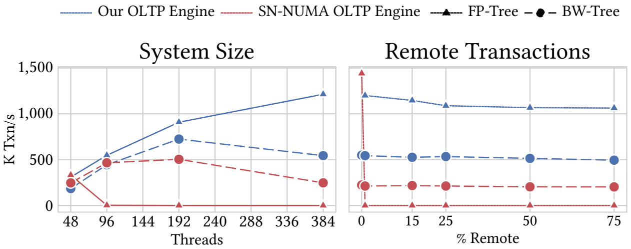

Figure 13: Throughput of TPC-C New-Order and Payment with 8 warehouses for increasing system size with 1% remote transactions (left) and for increasing remote transactions at largest system size (right).

As baseline, we compare our light-weight OLTP engine that is based on our runtime system with an OLTP engine based on the design of [31] that uses a NUMA-aware *shared-nothing* design where transactions are directly executed by transaction managers (i.e., not by delegating tasks to our runtime system). To allow a fair comparison, we also omit concurrency control and recovery in the baseline.

OLTP Workload: In order to compare our light-weight OLTP engine with the shared-nothing baseline, we use the TPC-C benchmark. For this experiment, we implemented the *New-Order* and *Payment* TPC-C transactions as tasks, which represent 88% of the workload. As data, we generate a TPC-C database with 8 warehouses (i.e., one for each NUMA-region considered to be favourable for the shared-nothing baseline used in this experiment). We partition the database across different system sizes by warehouse IDs ranging from 1 to 8 NUMA regions (i.e., system sizes from 48 to 384 threads). Moreover, we vary the fraction of *New-Order* and *Payment* transactions that need to access remote warehouses from 0% to 75% to simulate workloads that range from perfect locality to almost no locality. Finally, we configure tables into virtual domains with the procedure outlined in Section 5.

Performance Results: The results of this experiment are shown in Figure 13. Observing the TPC-C throughput for increasing system size on the left of Figure 13 reveals that using the FP-Tree results in brittle performance in the NUMA-aware system, i.e., degrading from the best throughput at the smallest system size (48 threads) to the worst throughput for larger system sizes (≥ 96 threads), which is in line with our results in Section 7.1. Whereas, the BW-Tree is more robust across different system sizes in the NUMA-aware system design. Nevertheless, the OLTP-engine based on our runtime system increases the overall performance for both indexes (the FP- and BW-tree) due to our effective contention management and efficient communication as observed earlier. Even more interestingly, our approach in combination with the FP-Tree establishes robust performance scaling the throughput of TPC-C transactions linearly with the system size.

Now, we present the TPC-C throughput at the largest system size under an increasing proportion of remote transactions on the right of Figure 13. The results indicate the sensitivity of the two different OLTP engines w.r.t. partitionability and locality of OLTP workloads; i.e., commonly OLTP workloads (especially TPC-C) are partitioned into exclusive partitions to achieve high throughput on large system sizes rendering them sensitive to remote transactions. The performance of the NUMA-aware system with FP-Tree perfectly demonstrates this sensitivity to remote transactions dropping from 1.5M txn/s at 0% remote transactions to barely any throughput at 1%. In contrast, our OLTP engine provides high throughput regardless the remote transactions, i.e., 1.2 - 1.1M txn/s. Again, BW-Tree improves the robustness of the NUMA-aware system. Still, in this case, our approach also enables better throughput for BW-Tree compared to the NUMA-partitioned baseline since our approach can further increase locality and reduce contention.

Insight: As we have shown in this experiment, using our runtime that relies on the configuration of virtual domains can also provide significant benefits for executing OLTP workloads. An interesting insight is that opposed to classical OLTP engines where optimal partitioning is crucial to maximise locality, partitioning does not play a significant role anymore when using our runtime system since delegation can provide high locality (and thus high throughput) even for non-partitionable workloads.

8 Related Work

In this paper, we make the case for configuration to achieve robust performance for a wide range of workloads and a variety of hardware platforms.

As hardware development advances, a huge body of research proposes solutions for new challenges and reiterates optimisations on all levels of system design ranging from synchronisation primitives to data structure designs and all the way to entirely new DBMS architectures. In response to increasing core counts and main memory capacity, systems like H-Store [21] and DORA [29] propose to partition the database in a fine-grained manner per hardware thread to avoid contention on data structures, where the latter actually applies delegation of transactions between worker threads.

Yu et al. [43] give a projection on the arising challenges when hardware with more than 1000 cores becomes commonplace for DBMS. They identify challenges of prior in-memory DBMS which they say will only be amplified as the number of cores increases. Since then, many proposals take different directions on how to adapt DBMS architectures to hardware with so many cores spread over many sockets. For example, ERIS [22] takes the DORA approach from multi-core OLTP to multi-socket OLAP and adds load balancing between system partitions to address skew in the workload. Instead, Hekaton [11] suggests to reject partitioning and to use a shared-everything approach to avoid the negative impacts of partitioning, e.g., skew. Porobic et al. [31] analyse the effect of hardware topologies with different core and socket counts on partitioning strategies and conclude that DBMS must be aware of the underlying hardware.

We propose to decouple the implementation of data structures (and larger components) from the system architecture, such that the configuration can adapt to new hardware and the existing implementations are reused.

Our general approach to adapt to hardware development follows other systems research. Node Replication [7] devises an automatic approach to transform any sequential data structure into a concurrent data structure for large scale hardware through a combination of shared memory and distributed computing approaches. They replicate data structures in a distributed manner across NUMA nodes and apply flat-combining delegation to share and synchronise a distinct data structure between workers on a NUMA node. [6] evaluate approaches to combine message passing known from distributed systems and common shared memory programming for NUMA-aware systems. They expect benefits from message passing in combination with delegation when communication of data and cache coherence between NUMA node becomes too expensive, which we show is already the case for index operations on the BW-Tree from two sockets and more (cf. Figure 9). But they point out that efficient message passing is crucial for delegation. To this end, FFWD [37] devises a delegation scheme with minimal cache coherence transactions between participating CPU cores, outperforming prior delegation schemes including flat-combining [12, 26], shared memory synchronisation primitives [9, 13], and latch-free algorithms. We extend FFWD with broader flexibility to enable efficient configuration and robust performance.

9 Conclusion

In this paper, we proposed a new approach for achieving robust performance of fundamental data structures. The main idea is to strictly separate the data structure design from the actual strategies how access operations are executed and to adjust the execution strategies by means of a configuration. In our evaluation, we demonstrated that reconfiguration establishes robust performance across diverse workloads and hardware sizes. While we believe that our abstractions (tasks and configurations) allow an efficient adaption of existing DBMS to make use of our approach, showing this would be beyond the scope of this paper though and is an interesting avenue of future work.

References

[1] Manos Athanassoulis, Kenneth S. Bøgh, and Stratos Idreos. 2019. Optimal Column Layout for Hybrid Workloads. *Proceedings of the VLDB Endowment* 12, 13 (2019), 2393–2407. https://doi.org/10.14778/3358701.3358707

[2] Nick Benton, Luca Cardelli, and Cédric Fournet. 2004. Modern Concurrency Abstractions for C#. *ACM Trans. Program. Lang. Syst.* 26, 5 (2004), 769–804. https://doi.org/10.1145/1018203.1018205

[3] Timo Bingmann. 2013. STX B+ Tree C++ Template Classes. http://panthema.net/2007/stx-btree/, accessed 2019-05-23.

[4] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the 2018 International Conference on Management of Data - SIGMOD '18*. ACM, 521–534. https://doi.org/10.1145/3183713.3196896

[5] Trevor Brown, Alex Kogan, Yossi Lev, and Victor Luchangco. 2016. Investigating the Performance of Hardware Transactions on a Multi-Socket Machine. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 121–132. https://doi.org/10.1145/2935764.2935796

[6] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra Marathe, and Mark Moir. 2013. Message Passing or Shared Memory: Evaluating the Delegation Abstraction for Multicores. In *Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304 (OPODIS 2013)*. Springer-Verlag, 83–97. https://doi.org/10.1007/978-3-319-03850-6_7

[7] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2018. How to implement any concurrent data structure. *Commun. ACM* 61, 12 (2018), 97–105. https://doi.org/10.1145/3282506

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[9] Travis Craig. 1993. *Building FIFO and Priority Queuing Spin Locks from Atomic Swap.* Technical Report. TR 93-02-02, University of Washington.

[10] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything You Always Wanted to Know About Synchronization But Were Afraid to Ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2522714, 33–48. https://doi.org/10.1145/2517349.2522714

[11] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, 1243–1254. https://doi.org/10.1145/2463676.2463710

[12] Dave Dice, Virendra J. Marathe, and Nir Shavit. 2011. Flat-Combining NUMA Locks. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 65–74. https://doi.org/10.1145/1989493.1989502

[13] David Dice, Virendra J. Marathe, and Nir Shavit. 2015. Lock Cohorting: A General Technique for Designing NUMA Locks. *ACM Trans. Parallel Comput.* 1, 2 (2015), 1–42. https://doi.org/10.1145/2686884

[14] Peter Dinda, Thomas Gross, David O'Hallaron, Edward Segall, and Jon Webb. 1994. *The CMU Task Parallel Program Suite.* Report. Carnegie-Mellon University Pittsburgh, Dept. of Computer Science.

[15] Markus Dreseler, Thomas Kissinger, Timo Djürken, Eric Lübke, Matthias Uflacker, Dirk Habich, Hasso Plattner, and Wolfgang Lehner. 2017. Hardware-Accelerated Memory Operations on Large-Scale NUMA Systems.. In *ADMS at VLDB*. 34–41.

[16] Jose M. Faleiro and Daniel J. Abadi. 2017. Latch-free Synchronization in Database Systems: Silver Bullet or Fool's Gold?. In *In Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR '17)*. 1–12.

[17] Gen-Z Consortium 2019. *Gen-Z Core Specification 1.1.* Gen-Z Consortium.

[18] Goetz Graefe, Arnd Christian König, Harumi Anne Kuno, Volker Markl, and Kai-Uwe Sattler (Eds.). 2010. *Robust Query Processing, 19.09. - 24.09.2010.* Dagstuhl Seminar Proceedings, Vol. 10381. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany.

[19] index 2017. Index Microbench. https://github.com/speedskater/index-microbench, accessed: 2019-07-09.

[20] Intel Corporation. 2019. *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4.* Report.

[21] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: a High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (2008), 1496–1499. https://doi.org/10.14778/1454159.1454211

[22] Thomas Kissinger, Tim Kiefer, Benjamin Schlegel, Dirk Habich, Daniel Molka, and Wolfgang Lehner. 2014. ERIS: A Numa-Aware In-Memory Storage Engine for Analytical Workloads. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1–12. https://doi.org/10.1.1.475.3378

[23] Sven O. Krumke and Clemens Thielen. 2013. The Generalized Assignment Problem with Minimum Quantities. *European Journal of Operational Research* 228, 1 (2013), 46–55. https://doi.org/10.1016/j.ejor.2013.01.027

[24] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 743–754. https://doi.org/10.1145/2588555.2610507

[25] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. 302–313.

[26] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. 2012. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*. USENIX Association, 6–6.

[27] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, 371–386. https://doi.org/10.1145/2882903.2915251

[28] Oguzhan Ozmen, Kenneth Salem, Jiri Schindler, and Steve Daniel. 2010. Workload-Aware Storage Layout for Database Systems. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. Association for Computing Machinery, 939–950. https://doi.org/10.1145/1807167.1807268

[29] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-Oriented Transaction Execution. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 928–939. https://doi.org/10.14778/1920841.1920959

[30] Orestis Polychroniou and Kenneth A. Ross. 2014. A Comprehensive Study of Main-Memory Partitioning and its Application to Large-Scale Comparison- and Radix-Sort. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 755–766. https://doi.org/10.1145/2588555.

2610522

[31] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pınar Tözün, and Anastasia Ailamaki. 2016. Characterization of the Impact of Hardware Islands on OLTP. *The VLDB Journal* 25, 5 (2016), 625–650. https://doi.org/10.1007/s00778-015-0413-2

[32] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2016. Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *Proc. VLDB Endow.* 10, 2 (2016), 37–48. https://doi.org/10.14778/3015274.3015275

[33] Jun Rao and Kenneth A. Ross. 1999. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK.* 78–89.

[34] Jun Rao and Kenneth A. Ross. 2000. Making B^+-Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.* 475–486. https://doi.org/10.1145/342009.335449

[35] James Reinders. 2007. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism.* O'Reilly Media, Inc.

[36] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. 2016. Design Principles for Scaling Multi-core OLTP Under High Contention. In *Proceedings of the 2016 International Conference on Management of Data.* ACM, 1583–1598. https://doi.org/10.1145/2882903.2882958

[37] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. 2017. Ffwd: Delegation is (Much) Faster Than You Think. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17).* ACM, 342–358. https://doi.org/10.1145/3132747.3132771

[38] Utku Sirin, Ahmad Yasin, and Anastasia Ailamaki. 2017. A Methodology for OLTP Micro-Architectural Analysis. In *Proceedings of the 13th International Workshop on Data Management on New Hardware.* ACM, 3076116, 1–10. https://doi.org/10.1145/3076113.3076116

[39] tbb [n. d.]. Threading Building Blocks (TBB). https://www.threadingbuildingblocks.org/, accessed 2019-07-25.

[40] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Thomas Fahringer, Kostas Katrinis, Erwin Laure, and Dimitrios S. Nikolopoulos. 2018. A Taxonomy of Task-Based Parallel Programming Technologies for High-Performance Computing. *The Journal of Supercomputing* 74, 4 (2018), 1422–1434. https://doi.org/10.1007/s11227-018-2238-4

[41] Robert Virding, Claes Wikström, Mike Williams, and Joe Armstrong. 1996. *Concurrent Programming in ERLANG (2nd Ed.).* Prentice Hall International (UK) Ltd.

[42] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data - SIGMOD '18.* ACM, 473–488. https://doi.org/10.1145/3183713.3196895

[43] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring Into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proceedings of the VLDB Endowment* 8, 3 (2014), 209–220. https://doi.org/10.14778/2735508.2735511