ERIS: A NUMA-Aware In-Memory Storage Engine for Analytical Workloads

Thomas Kissinger*, Tim Kiefer*, Benjamin Schlegel*, Dirk Habich*, Daniel Molka†, Wolfgang Lehner*
*Database Technology Group / †Center for Information Services and High Performance Computing
Technische Universität Dresden
01062 Dresden, Germany
{firstname.lastname}@tu-dresden.de

ABSTRACT

The ever-growing demand for more computing power forces hardware vendors to put an increasing number of multiprocessors into a single server system, which usually exhibits a non-uniform memory access (NUMA). In-memory database systems running on NUMA platforms face several issues such as the increased latency and the decreased bandwidth when accessing remote main memory. To cope with these NUMA-related issues, NUMA-awareness has to be considered as a major design principle for the fundamental architecture of a database system.

In this paper we present ERIS, a NUMA-aware in-memory storage engine that is based on a data-oriented architecture. In contrast to existing approaches that focus on transactional workloads on a disk-based DBMS, ERIS aims at tera-scale analytical workloads that are executed entirely in main memory. ERIS uses an adaptive partitioning approach that exploits the topology of the underlying NUMA platform and significantly reduces NUMA-related issues. We evaluate ERIS on widespread standard server systems as well as on a system consisting of 64 multiprocessors and 512 cores. On these platforms, we achieve a more than linear speedup for index lookups and scalable parallel scan operations that are only limited by the available local bandwidth of the multiprocessor. Moreover, we measured a performance gain of up to 200% (index lookups) respectively 660% (column scans) in the memory-bound case compared to a NUMA-agnostic storage subsystem.

1. INTRODUCTION

As a consequence of the high main memory capacities in today's servers, modern database systems are very often in the position to store their entire data in main memory. Latency and bandwidth of the main memory are the major bottlenecks of such in-memory DBMSs. The significance of these bottlenecks increases when we consider the current trend towards tera-scale multiprocessor systems that exhibit a non-uniform memory access (NUMA). On NUMA platforms, each multiprocessor has its own local main memory that is accessible by other multiprocessors via a communication network. Database systems running on NUMA platforms face several issues such as the increased latency and the decreased bandwidth when accessing remote main memory. Additionally, we observe a worsening of the scalability of latches and atomic instructions. This is a result of the cache coherence maintenance overhead induced by the NUMA system. These issues are already measurable on wide-spread server systems consisting of four or eight multiprocessors. The demand for more parallel hardware forces vendors to put even more multiprocessors into a single server system (e.g., Oracle SPARC M6 with up to 96 multiprocessors or SGI UV2000 that are sold as HANA-Boxes). We also expect emerging technologies like 3D DRAM/CPU stacking to let NUMA characteristics appear in a single multiprocessor, where each core has its local low latency and high bandwidth main memory [13, 18]. To allow database systems to scale-up on today's and future platforms, NUMA-awareness has to be considered as a major design principle for the fundamental architecture of a database system.

Recent research revealed that a data-oriented architecture (DORA) enables disk-based database systems to scale-up on multicore systems in the context of transactional workloads [19]. DORA uses a thread-to-data instead of the conventional thread-to-transaction assignment and thus dramatically reduces contention on lock tables as well as latch contention on data objects. Since such an architecture relies

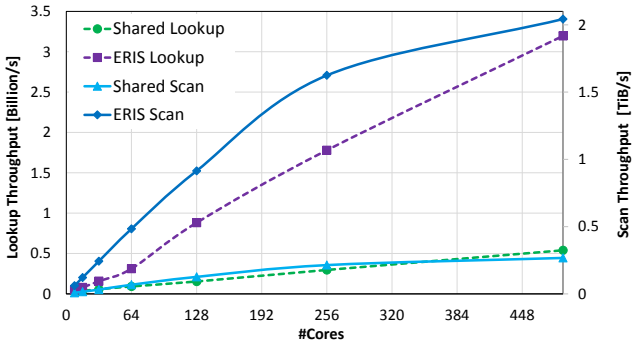


Figure 1: Index Lookup and Column Scan Scalability of ERIS on a SGI UV 2000.

Proceedings of the VLDB Endowment, Vol. 7, No. 14

on logical partitioning, load balancing is necessary to adapt the partitioning to workload changes. More recent works extended the data-oriented approach by a physically partitioned disk page buffer pool [20] and a NUMA-aware as well as workload-aware data placement algorithm, which tries to minimize inter-socket communication [21].

In this paper we present ERIS, a NUMA-aware all-in-memory storage engine for tera-scale analytical workloads. ERIS is also based on a data-oriented architecture and thus splits data objects into partitions, which are assigned exclusively to individual workers. Workers are pinned on a designated core of the platform and execute data commands (i.e., scan, lookup, or insert/upsert) on their partitions. In contrast to existing approaches, ERIS aims at tera-scale analytical workloads that are executed purely in main memory and fundamentally differ from transactional workloads. First, transaction-oriented systems try to cluster partitions that belong to the same transaction class to minimize inter-socket communication as well as interference between transactions running in parallel. However, analytical queries require a high degree of parallelism to execute with low latency and thus data is best spread out as much as possible. Second, analytical queries often generate huge amounts of intermediate results and inevitably generate inter-socket communication on NUMA systems. Hence – in opposition to transactional workloads – the effective handling of intermediate results, whose size grows with the size of the base data, and the routing between workers are mission critical components of our NUMA-aware storage engine. Third, analytical workloads are usually read-only and thus the storage subsystem should not implement comprehensive locking and latching mechanisms, which could force a large scan operation to block. The more feasible use of a non-blocking multi-version approach or the use of staging tables eliminates the need to optimize for minimal lock table contention. Finally, analytical workloads execute scans or lookups on large data objects. Hence, a storage engine that relies on logical partitioning needs a topology-aware load balancing that is able to quickly balance huge data objects with minimal impact on the overall query throughput.

With ERIS, we address these issues by proposing a NUMA-optimized high-throughput routing layer, that is able to efficiently distribute data commands over the different worker threads as well as to coalesce similar data commands to a single storage operation (e.g., scan sharing [23]). Additionally, we propose different load balancing mechanisms to move large amounts of data between workers either on the same multiprocessor or on different multiprocessors of the NUMA system and a corresponding configurable load balancing algorithm. In our evaluation, ERIS achieves a more than linear speedup for index lookup throughput (1 billion keys) on a NUMA system equipped with 64 multiprocessors and a total of 512 cores as depicted in Figure 1. Moreover, we measured a performance gain of up to 200% (index lookups) respectively 660% (column scans) on ERIS in the memory-bound case compared to a NUMA-agnostic storage subsystem.

Contributions

The contributions of this paper are as follows:

(1) We give an overview of the different topologies of current NUMA server systems including a system with 64 multiprocessors (512 cores in total). We show bandwidth and latency measurements for local and remote memory accesses and describe their negative effect on in-memory database systems.
(2) We describe the architecture of our NUMA-aware in-memory storage engine ERIS, which supports three basic storage operations that are required to run analytical queries: scan, lookup, and insert/upsert. Besides reading operations, fast writing operations are required, especially to materialize large intermediate results of a query in a NUMA-aware fashion.
(3) We propose a NUMA-optimized high-throughput routing layer that supports unicast and multicast to efficiently distribute data commands between workers. Additionally, the routing layer implements query processing primitives and is able coalesce similar data commands to the same partition.
(4) We describe NUMA-aware load balancing mechanisms to efficiently move large portions of a partition between workers. Moreover, we present a configurable load balancing algorithm that offers a tunable level of balancing aggressiveness.
(5) We evaluate ERIS and compare our approach with a NUMA-agnostic in-memory storage engine and show that ERIS scales even on large-scale NUMA systems. Additionally, we use hardware instrumentation techniques to reason about our results.

Structure

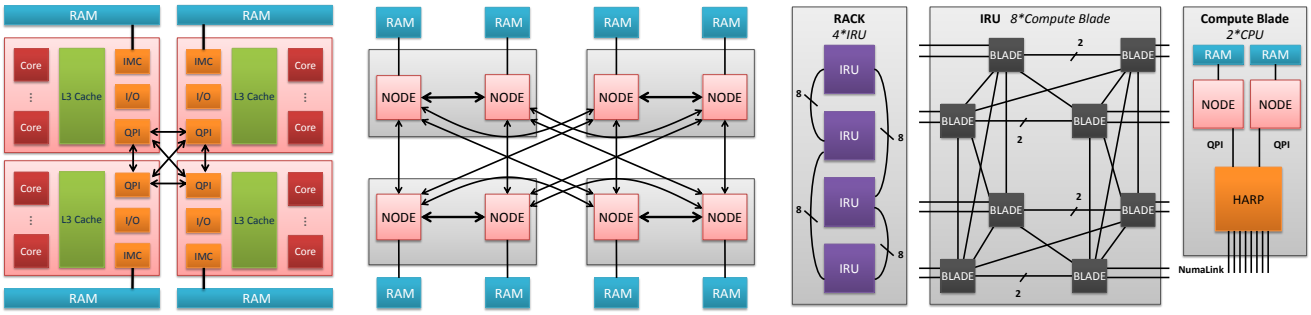
The remainder of this paper is structured as follows: In Section 2 we describe current NUMA platforms. Section 3 gives an overview of the architecture of ERIS and details on its components. In the following Section 4 we describe implementation details and evaluate ERIS on different hardware platforms. In Section 5 we compare our approach to the existing related work. Finally, Section 6 concludes the paper and outlines future work.

2. NUMA SYSTEMS

In this section, we introduce NUMA system architectures and present low level benchmark results of the NUMA machines used in our experimental setup. As a reference throughout the section, these systems are depicted in Figure 2. We derive fundamental design principles for NUMA-aware storage engine architectures at the end of the section.

2.1 NUMA System Architecture

NUMA systems consist of several interconnected multiprocessors, that are also referred to as *nodes*. Each multiprocessor contains multiple processing units (cores) and an integrated memory controller (IMC). Consequently, the installed main memory is distributed among the IMCs in different multiprocessors. However, each multiprocessor can access each memory location. Thus, latency and bandwidth of memory accesses depend on the distance between the requesting multiprocessor (source node) and the multiprocessor that contains the data (home node). The *local memory* associated with each multiprocessor is accessed with low latency at a high bandwidth. In contrast, *remote memory* is accessed via point-to-point connections [9, 10] between the multiprocessors that add latency and limit the achievable bandwidth. In the worst of our cases the latency of remote access is approximately 10 times higher and the bandwidth is limited to about 11% in comparison to local accesses.



(a) Intel Machine (Detailed). (b) AMD Machine (Topology View). (c) SGI Machine (Topology View).

Figure 2: NUMA Machines used for Evaluation.

Intel machine	**AMD machine**	**SGI machine**
4x Intel Xeon E7-4860	4x AMD Opteron 6274 (dual node)	64x Intel Xeon E5-4650L
40 cores (80 HW threads)	64 cores	512 cores
128 GB memory (32 GB per node)	64 GB memory (8 GB per node)	8 TB memory (128 GB per node)
24 MB LLC per sockets	12 MB LLC per socket (2x 6 MB)	20 MB LLC per socket
QPI: 12.8 GB/s per link	HyperTransport: 12.8 GB/s per link	QPI: 16 GB/s to HARP NumaLink6: 2x 6.7 GB/s between HARPs
Ubuntu 13.4 server (3.8.0-29)	Ubuntu 13.4 server (3.8.0-31)	SLES 11 SP2 (3.0.93-0.5)

Table 1: Machine Specification Overview.

Multiple levels of caches are commonly used to mitigate the performance impact of the above-mentioned latency and bandwidth constraints. The caches are distributed over the multiprocessors as well. All currently available NUMA systems enforce cache coherence to maintain a consistent view of all processing units on the shared address space. Small-scale NUMA systems with a managable amount of nodes typically rely on snooping based cache coherence protocols that involve frequent broadcasts of requests to all multiprocessors. It has been shown in prior work [8] that the overhead of the coherence protocol caused by accesses to shared data can be very severe in such systems. In contrast, larger systems like the SGI UV 2000 usually implement directory based cache coherence protocols between the nodes. SGI, e.g., uses NumaLink to connect blades with one another while the two nodes in each blade are connected to a hub via their Intel QPI links. The hub presents itself to the nodes as an external memory controller that participates in the snooping based coherence protocol. However, the requests are not broadcasted to all the hubs in the system. Instead, requests are only forwarded if the corresponding directory entry indicates a remote copy.

Naturally, data placement is an important aspect to consider with NUMA systems and data should be located close to the multiprocessor that accesses it frequently. The default data placement policy of Linux is called *first touch*. Newly allocated memory is placed local to the thread that actually writes (touches) it for the first time. It is, however, possible that memory is allocated on remote memories. Moreover, the default thread scheduler in Linux operating systems may migrate threads frequently to different multiprocessors, although it prefers intra-node thread migrations to inter-node migrations. This leads to remote memory accesses, even when the memory was allocated locally in the first place. Hence, the operating system leaves many opportunities for suboptimal (i.e., remote) memory access patterns. This is especially true, when many threads access a large portion of the main memory.

2.2 Low-Level Benchmark Results

For our algorithm engineering and performance experiments, we use three different NUMA systems ranging from 4 multiprocessors and 64 GBs of main memory to 64 multiprocessors and a total of 8 TBs of main memory. The hardware specifications of the *Intel machine* with 4 multiprocessors, the *AMD machine* with 8 multiprocessors (on 4 sockets), and the *SGI machine* with 64 multiprocessors are summarized in Table 1. To gain deeper insights in the performance of the three different NUMA machines, we conducted several low level benchmarks. The best-case bandwidth and latency performances are an upper bound for the achievable performance and will help us to reason about the performance of our own algorithms. All measurements are performed with the BenchIT tool [8]. The results are shown in Table 2.

2.2.1 *Intel machine*

The Intel machine with 4 multiprocessors is the smallest system in our setup. The nodes of the Intel machine are fully connected via QPI links [10] as depicted in Figure 2(a). The results of our experiments show that the latency of remote memory accesses is only 50% higher than for local accesses. The impact of the QPI link on the achievable bandwidth is more severe as it results in 2.5 times lower data rates compared to local memory. However, the effects of the non-uniform memory access are small compared to the other two

Intel machine			AMD machine			SGI machine		
distance	bandwidth (GB/s)	latency (ns)	distance (link width)	bandwidth (GB/s)	latency (ns)	distance	bandwidth (GB/s)	latency (ns)
local	26.7	129	local	16.4	85	local	36.2	81
1 hop QPI	10.7	193	1 hop HT (full link)	5.8	136	2nd processor	9.5	400
			1 hop HT (split,single)	4.2	152	1 hop NUMALink	7.5	505 - 515
			1 hop HT (split,dual)	2.9	152	2 hop NUMALink	7.5	625 - 635
			2 hop HT (split,single)	3.7	196	3 hop NUMALink	7.1	745 - 755
			2 hop HT (split,dual)	1.8	196	4 hop NUMALink	6.5	870

Table 2: Memory Read Bandwidth in GB/s and Read Latency in ns. Bandwidths are measured with concurrent sequential reads from all cores of the multiprocessor in order to maximize the amount of outstanding requests. Latencies are measured with a single thread that performs a pointer-chasing routine on memory allocated at different multiprocessors.

machines as communication between any two multiprocessors requires only one hop via QPI.

2.2.2 AMD machine

The second machine in our setup is an AMD machine. As shown in Figure 2(b), it is actually a 4 socket system where each socket houses a *dual node package*. The two nodes in a package communicate via HyperTransport [9] which practically results in a system with 8 multiprocessors. Each multiprocessor has four HyperTransport ports to connect to either the I/O subsystem or to other multiprocessors. As a unique feature of the AMD machine, HyperTransport links can be split into sublinks to connect a node with two other nodes with just one HyperTransport link. However, this results in different link bandwidths for different links. Additionally, even with split links, the AMD machine is not fully connected and certain routes require two hops.

As indicated in Figure 2(b), the two nodes that share a socket are connected via a dedicated (not split) HyperTransport link and can therefore utilize the full 16 bit link widths. Connections between other nodes are realized with 8 bit sublinks and hence have a lower connection bandwidth. Furthermore, some of the split links only have one sublink populated (denoted by *split,single* in Table 2) while both sublinks are occupied on other links (denoted by *split,dual*). Our experiments mirror these characteristics; depending on the distance of memory and accessing thread, we measure six different bandwidths and four different latencies. The disparities between local access and the furthest remote access are a factor of 9.1 in bandwidth and 2.3 in latency.

2.2.3 SGI machine

The third machine in our setup is an SGI UV 2000 with 64 multiprocessors and a total of 8 TBs main memory. An overview of the topology is shown in Figure 2(c). Our system consists of 1 rack that houses 4 Individual Rack Units (IRUs). Each IRU consists of 8 Compute Blades, that in turn contain 2 multiprocessors each. Each socket is equipped with an 8-core Intel Xeon CPU with 128 GBs of local main memory.

The two multiprocessors in a Compute Blade are connected via QPI to a communication hub called HARP. The HARPs are NumaLink hubs that connect the multiprocessors in a Compute Blade to other Compute Blades in the same as well as in other IRUs. As shown in Figure 2(c), each blade in our system has 8 connections to other blades. Each connection consists of two NUMALink6 links, one for each multiprocessor in the blade. The 8 blades in an IRU are connected as a 3D enhanced hypercube [25]. Each blade in an IRU is additionally connected to two blades in other IRUs. This topology leads to connections with up to four hops and six different bandwidths.

Measuring all possible distances reveals that the differences in bandwidth and latency between local access and the furthest remote access are as high as factor 5.5 and 10.7, respectively.

2.3 Design Principles for NUMA-Aware Storage Engines

From the general NUMA architecture as well as our benchmark results, we derive that a NUMA system should be treated like a distributed system and that a scalable in-memory storage engine must be designed to maximize local memory accesses. Reading from or writing to remote memory suffers from up to ten times higher latency and significantly lower bandwidths, hence remote accesses should be avoided whenever possible and batching should be considered for inevitable accesses to hide the bad latency. Furthermore, remote and concurrent memory accesses lead to cache concurrency as well as worse cache locality and hence higher cache coherence overhead. As a conclusion, a scalable storage engine for NUMA systems must provide partitioning of the data that adapts quickly to changing workloads by employing efficient load balancing algorithms. Moreover, by means of data and thread placement, the storage engine must minimize remote memory accesses by primarily working in data object partitions that are located in the local main memory. In turn, this leaves sufficient link capacities for remote accesses caused by inevitable communication during query processing and by load balancing operations. The operating system cannot provide sufficient locality due to its insufficient knowledge about the application and its partitioning.

3. ERIS

In this section we describe the architecture of ERIS and its individual components as visualized in Figure 3. The central components of the storage engine are the worker threads, which we call Autonomous Execution Units (AEU). Each core, respectively hardware context, of the platform runs exactly one AEU. All AEUs pinned on the same multiprocessor use a common memory manager, because they

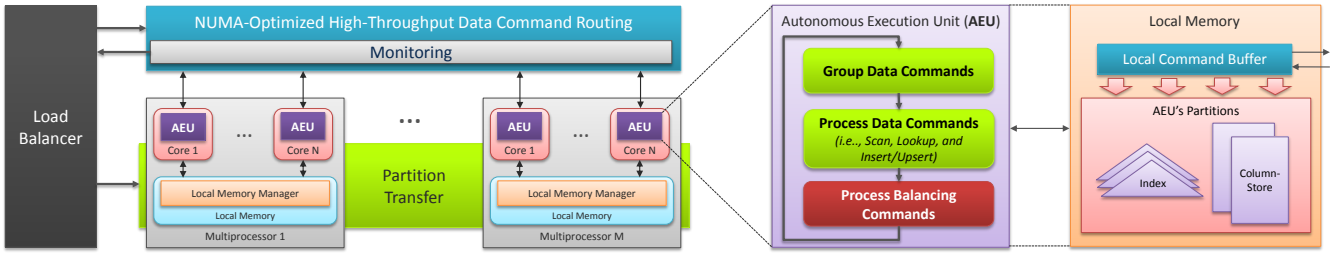


Figure 3: Architectural Overview of ERIS and AEU Details.

share the same local main memory and are thus able to quickly exchange data partitions during load balancing. A set of partitions—each belonging to a different data object—is assigned to each AEU. The AEU's main task is to manage its partitions and to process incoming data commands (i.e., scans, lookups, and inserts/upserts) on these partitions. To efficiently route data commands during query processing between AEUs, ERIS includes a NUMA-optimized high-throughput data command routing layer. The NUMA-aware load balancer of ERIS observes the current load of the AEUs via a monitoring component and triggers balancing commands in case of an uneven AEU utilization. In the following we describe the central components of ERIS in more detail.

3.1 AEUs and Memory Management

Traditional architectures bind transactions to a number of threads and use a global memory manager (per data object). This way of accessing and storing data is highly discouraging when running on NUMA platforms, because data is distributed in an uncoordinated way across the memory of the different multiprocessors. In turn, this causes a high number of remote memory accesses by the transaction threads that are accessing the data object.

For that reason, ERIS employs a data-oriented architecture where each data object is logically partitioned. Each available core of the system runs an AEU, which is bound to be only executed on this single core or hardware context respectively. Every single AEU gets assigned a set of disjoint partitions—each belonging to a different data object—and is exclusively responsible for that portion of the individual data object. This approach restricts memory accesses of an AEU to the multiprocessor's local main memory and data objects do not have to be protected against concurrent accesses via latches. ERIS primarily uses range partitioning to split data objects into partitions. We decided against hash partitioning, because it is not order preserving and thus disallows efficient range scans and hinders an efficient load balancing. Nevertheless, ERIS supports hash tables by using different hash functions on a per-partition level. In scenarios where a table is solely completely scanned, we employ physical data size partitioning instead of range partitioning, because there is no suitable attribute as partitioning criteria available. Here, ERIS only keeps track of those AEUs that actually store a partition of the corresponding data object and uses the multicast capabilities of the routing layer to distribute data commands.

Regarding the memory management, a global memory manager (per data object) is not feasible on a NUMA platform. Instead, ERIS deploys one memory manager per multiprocessor (and data object). Per-multiprocessor memory managers help to reduce the contention on the memory management subsystem, which is often the bottleneck during writing operations to a data object. Moreover, this approach limits allocations of AEUs to the local main memory and enables the load balancer to perform an efficient intra-node balancing. To scale with a high number of cores per multiprocessor, our memory managers use thread-local caching mechanisms and thus decrease contention on the local memory management.

On the right hand side of Figure 3 we illustrate the AEU loop as well as the local memory organization of an AEU. The AEU mainly keeps local data command buffers and the actual data object partitions (either stored as a column-store or an index). In the first stage of the loop, the AEU scans its data command buffer, which is periodically filled by the routing layer, and groups commands by the accessed data object and the command type. This optimization step is beneficial to coalesce the same type of access to the same partition. For instance, an AEU is able to execute multiple scan commands on the same partition with a single scan and is thereby implementing scan sharing in combination with MVCC to ensure isolation. Moreover, the command grouping allows us to execute multiple index lookup or insert/upsert operations in a single batch operation to hide the main memory latency. Following the grouping step, the AEU actually processes its data command buffer, which is the most time consuming part of the loop. Afterwards, the AEU checks its command buffer for pending balancing or transfer commands. Such commands force an AEU to grow or shrink its partition or to transfer a range of its partition to another AEU. We discuss the details of the load balancing process in Section 3.3.

3.2 NUMA-Optimized High-Throughput Data Command Routing

The data command routing is the most essential part of ERIS, because AEUs have to be supplied with data commands just in time. Especially during the execution of analytical queries, large amounts of data commands have to be routed between AEUs (e.g., lookup operations during a join). Thus, the main goal of the data command routing is to distribute data commands at a high throughput. A data command consists of a storage operation type (i.e., scan, lookup, or insert/upsert), a data object identifier, a reference to a callback function, a data segment that contains all the necessary parameters for the storage operation (e.g., a batch of keys for the lookup or filters for a scan), and additional data that is necessary for the query processing. Our data command routing mechanism is shown in Figure 4. The

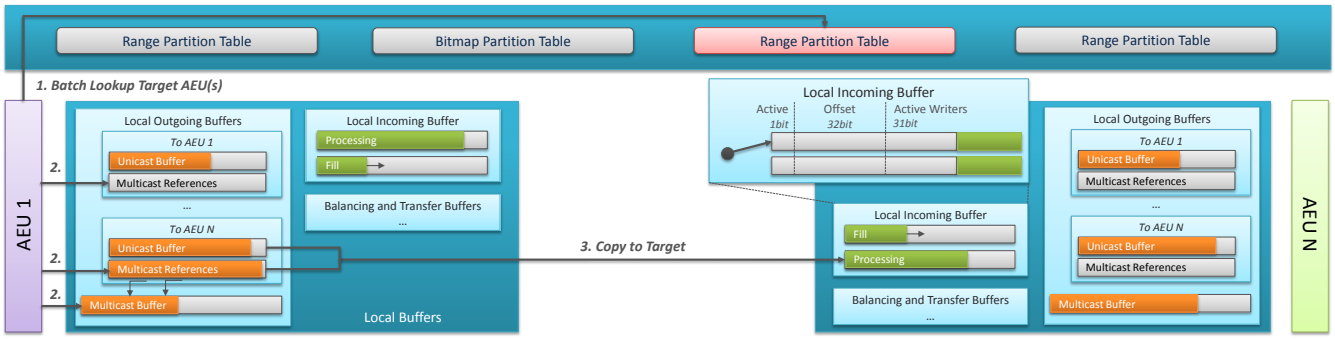


Figure 4: NUMA-Optimized High-Throughput Data Command Routing.

core components are the partition tables, which keep track of the partitioning of individual data objects. As already mentioned, a data object is either clustered, respectively sorted, on one or more of its attributes or it is distributed without any partitioning criteria. In the clustered case, the routing table stores the attribute range to AEU mapping (range partition table). If the data object is not partitioned on any attribute, the routing table only saves whether or not an AEU stores a partition of that data object (bitmap partition table). Since the routing tables are small data structure that are rarely updated (only during load balancing) and are frequently read, they usually fit into the caches of all multiprocessors and are thus not causing any remote memory accesses.

Besides the routing tables, our data command routing uses a comprehensive local buffering strategy. Each AEU uses a set of outgoing buffers—one unicast buffer and one multicast reference buffer for each running AEU in the system—, a multicast buffer, and two bigger incoming buffers. All buffer types are stored in the local main memory of each AEU to provide fast access to them.

Every time an AEU generates a data command during the processing stage, it starts with a batch lookup of the responsible AEUs for that data command in the corresponding routing table of the target table (step 1 in Figure 4). The routing tables use the content of the data segment of the data command to lookup the designated target AEUs. As soon as the target AEUs are determined, the routing layer splits the command into smaller pieces, for instance if a lookup data command contains keys in its data segment that belong to different partitions. Data commands for a single AEU are written to the corresponding outgoing buffer of the source AEU (step 2). If multiple AEUs are responsible for a data command (e.g., a scan that needs to be distributed to different AEUs), the command itself is written to the multicast buffer and references to this data command are stored in the individual multicast reference buffers. If an outgoing buffer is either full or the AEU starts over its processing loop, the specific outgoing buffer including its multicast data commands is copied to the incoming buffer of the target AEU (step 3). This local pre-buffering dramatically increases the data command routing throughput, because the contention on the incoming buffers is reduced and multiple data commands can be copied sequentially. Thus, the high latency of remote memory accesses on the NUMA platform does not become the bottleneck.

While outgoing buffers are private to an AEU and thus

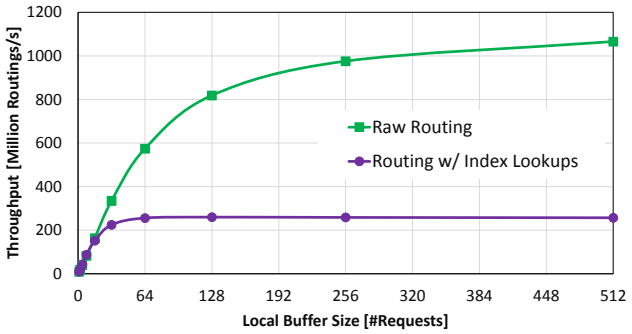


Figure 5: Data Command Routing Throughput as a Function of Local Buffer Size.

do not require any concurrency control, incoming buffers are written by different AEUs and are read by the host AEU at the same time. Hence, incoming buffers need an efficient and ideally latch-free concurrency control mechanism. We employ an adapted version of the latch-free multi-buffer proposed in LLAMA [16]. Each AEU has two incoming buffers of an equal size. One buffer is currently writable for all AEUs and the other one is currently the processed data command buffer of the owning AEU. To implement incoming buffers latch-free, each of them contains a 64bit wide buffer descriptor that uses 1bit for determining whether the buffer is still active or not, 32bit to save the current offset inside the buffer, and the remaining 31bit for storing the number of active writers to the buffer. If an AEU wants to write to an incoming buffer, it first determines the writable buffer, increases the offset by the size of data that needs to be written, increments the number of active writers, and finally atomically updates the buffer descriptor using a CAS instruction. If the atomic buffer descriptor update fails, the entire process is repeated. This approach allows multiple AEUs to write to the incoming buffer in parallel. After the AEU has successfully written its data commands, it atomically decrements the number of active writer to the buffer. The owning AEU of the incoming buffers swaps both buffers each time it enters the data commands processing stage. The AEU updates the pointer to the new writable buffer and atomically flips the active bits of both buffers. Afterwards, it holds on until all AEUs have finished writing their data to the new data command processing buffer.

We evaluated the effect of the outgoing buffer size on the

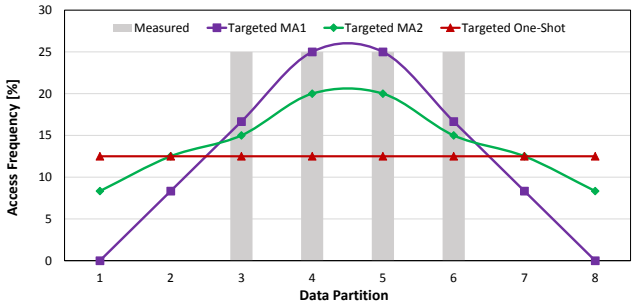


Figure 6: Configurable Load Balancing Algorithm of ERIS.

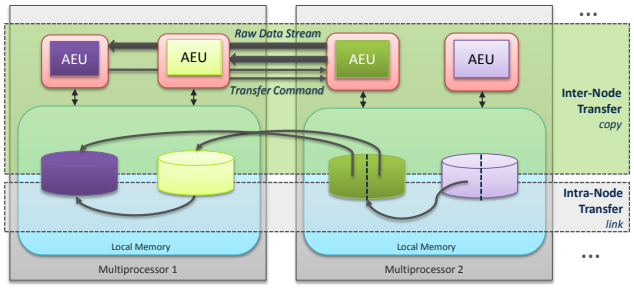


Figure 7: NUMA-Aware Partition Transfer via Link And Copy.

routing throughput on the AMD machine (see Section 2) and visualize the results in Figure 5. Regarding the raw routing throughput, where AEUs skip the processing phase, we observe that the throughput doubles with the size of the outgoing buffers until the bandwidth of the NUMA interconnects start to saturate. If we enable the processing phase and generate index lookup data commands, the peak throughput is already reached for an outgoing buffer size of 128 data commands, because the throughput is now dominated by the index lookups during the processing stage of the AEUs. A comparison of both measurements demonstrates the effectiveness of our NUMA-optimized high-throughput data command routing.

3.3 Load Balancing

Besides data command routing, ERIS requires a NUMA-aware load balancer component to adapt the partitioning to a changing workload. Since ERIS aims at analytical workloads, the maximization of parallelism is the main objective of the load balancer. Thus, there is no need for inter-data-object balancing, for instance to colocate certain partitions of different data objects on the same AEU as it is beneficial for transactional workloads. We distinguish between two major scenarios:

(1) The data object is always scanned in its entireness and is thus not partitioned by a specific attribute.
(2) The data object faces lookups or scans in certain ranges and is thus partitioned by one or more attributes.

In the first case, the physical partition size is the considered metric for the load balancer, because the scan works only efficient if all AEUs have to scan the same amount of data. In the second scenario, we use the access frequency as primary metric, because lookups and range scans only involve a certain set of AEUs. Additional metrics for the latter scenario are the mean execution time of a data command for a specific partition. Different execution times are mostly a result of different depths of tree-based index structures, or column store partitions that are frequently accessed but fit into the cache of a multiprocessor, or effects of data command coalescing.

The ERIS adaption loop starts with the monitoring of the different metrics on a per data object level. Based on the captured metrics, the load balancer periodically checks the load of ERIS for imbalances. If the standard deviation between the different AEUs exceeds a given threshold, the load balancer executes a load balancing algorithm that calculates a new target partitioning. With the help of the current and the targeted partitioning, the load balancer computes a series of balancing commands that are routed to the involved AEUs. Such balancing commands include the new data respectively key range the AEU is responsible for and a set of transfer commands that instruct the AEU where it has to fetch the missing partition data from. Next, we will describe our configurable load balancing algorithm and the NUMA-aware partition transfer mechanisms in more detail.

3.3.1 *Configurable Load Balancing Algorithm*

The load balancing algorithm receives the approximated metric distribution of a single data object of the recent sample period as well as the current partitioning as input and outputs the targeted partitioning for the respective data object. Figure 6 shows an exemplary metric distribution measurement (the access frequency in this specific case) that was sampled per partition and is represented as a histogram. In this specific scenario, partitions 3 to 6 each got 25% of the accesses, which is a severe imbalance. The most aggressive, but also most expensive approach is taken by the *One-Shot* load balancing algorithm configuration. This algorithm configuration computes the average access frequencies of all partitions and calculates a target partitioning that is fully balanced. The *One-Shot* configuration is suitable for workloads that change rarely but heavily. An alternative configuration uses the moving average (*MA*). For instance, the *MA1* configuration computes for each partition the moving average of the partition's direct neighbors including itself and adjusts the target partitioning appropriately. The *MA* configuration adapts more slowly to the new workload, but does not cause as much balancing overhead as the *One-Shot* algorithm and is thus suitable for highly dynamic workloads. As depicted in Figure 6, the aggressiveness of the *MA* configuration depends on the range the moving average is calculated over and turns into the *One-Shot* algorithm when configured as *MA7* in our setup, because it equally calculates the full average across all partitions. As soon as the load balancing algorithm has finished the calculation of the target partitioning, the latter is compared to the current partitioning and the load balancer generates a series of balancing commands for that data object.

3.3.2 *NUMA-Aware Partition Transfer*

If the load needs to be balanced, each AEU that has to grow or shrink its local partition of the data object receives a balancing command. Such a balancing command first includes the new partition ranges for the AEU. The AEU updates the corresponding routing tables and saves the lower and upper bounds of its ranges internally, because it has to

compare each incoming data command against its bounds to check its validity. If the AEU encounters an invalid data command (i.e., a data command that references keys outside its updated range) it forwards this data command to the AEU that is now responsible for the range. If a data object is balanced that is not partitioned by a specific criteria, the balancing command includes the number of tuples that have to be fetched or handed over to another AEU. To avoid situations of overlapping partition ranges, all AEUs that are involved in the current balancing cycle have to be synchronized for updating a data object's routing table.

Besides the information about the new partition ranges, a balancing command includes a set of transfer commands. We continue with the example of Figure 6 and look at the balancing of partitions 1 to 4 using the *One-Shot* load balancing algorithm. The balancing of partitions 5 to 8 is very similar, because this specific workload is symmetric. In Figure 7, we illustrate the corresponding partition transfer process for that example. For reasons of simplicity, we assume a NUMA system consisting of four multiprocessors and two cores per multiprocessor in the example. Because the current range of partitions 1 and 2 is not accessed by the new workload, partition 1 receives a first transfer command instructing the AEU to take over the entire range of partition 2. Since both partitions reside in the same local memory and thus in the same memory management domain, AEU 1 uses the cheap *link* mechanism to transfer partition 2. To do so, AEU 1 firstly unlinks the respective portion of the partition (the complete partition in our case). Afterwards, AEU 1 simply links (e.g., in case of a tree-based index) respectively appends (in case of a column-store) partition 2 to its own partition 1. For the transfer of half of partition 4 to partition 3, ERIS also uses the *link* mechanism, because both partitions are located on multiprocessor 2. The remaining two transfers from partition 3 to partitions 1 and 2 are inter-node transfers and thus use the *copy* mechanism for the partition transfer. Such a *copy* operation requires a cooperation between source and target AEU, if the data object is stored as an index to avoid the high latency of remote memory accesses while traversing through the tree-based index. In this case, the source AEU forwards the transfer command to the source AEU, which flattens the partition to an exchange format and streams it sequentially to the target AEU. The target AEU converts the data stream back to an index and links it to its existing partition. If the data object is already stored in a flat format such as a column store, the target AEU directly copies the data from the source AEU. As soon as an AEU has processed all its transfer commands, it becomes ready to continue normal operation and when all AEUs have completed their balancing command, the balancing loop starts over again.

4. IMPLEMENTATION DETAILS AND EVALUATION

In this section, we detail our implementation of an AEU and investigate the behavior of ERIS. We evaluate ERIS' scan, lookup, and upsert performance by comparing it to the NUMA-agnostic shared index respectively shared scan as baseline. For the baseline experiments we use the same data structures as for the AEUs. The difference is that those data structures are not partitioned and are thus synchronized via atomic instructions for updates, because they are accessed by different transaction threads in parallel.

An AEU implements a simple column store as well as a prefix tree [7] as index. We decided to use a prefix tree, because this index structure is order-preserving (applies not to a hash table), in-memory optimized, and offers a high update performance (does not apply to a B+-Tree). To implement the range partition tables of ERIS, we decided to deploy a CSB+-Tree [24], because it works fast for sparsely distributed data and it scales with an increasing number of ranges, respectively AEUs, compared to a simple array.

For our evaluation, we compare different configurations (e.g., different index sizes) and reason about the observed results. Furthermore, we compare ERIS to different memory allocation strategies for column data and evaluate their scan performances. For certain experiments, we additionally present results of hardware event measurements to gain deeper insights in the algorithms' behaviors.

We have already demonstrated the ability of ERIS to scale with an increasing number of multiprocessors, and hence cores, in Figure 1. Moreover, in Figure 5 we presented experiment results that show that the NUMA-aware high-throughput routing is not the bottleneck of ERIS.

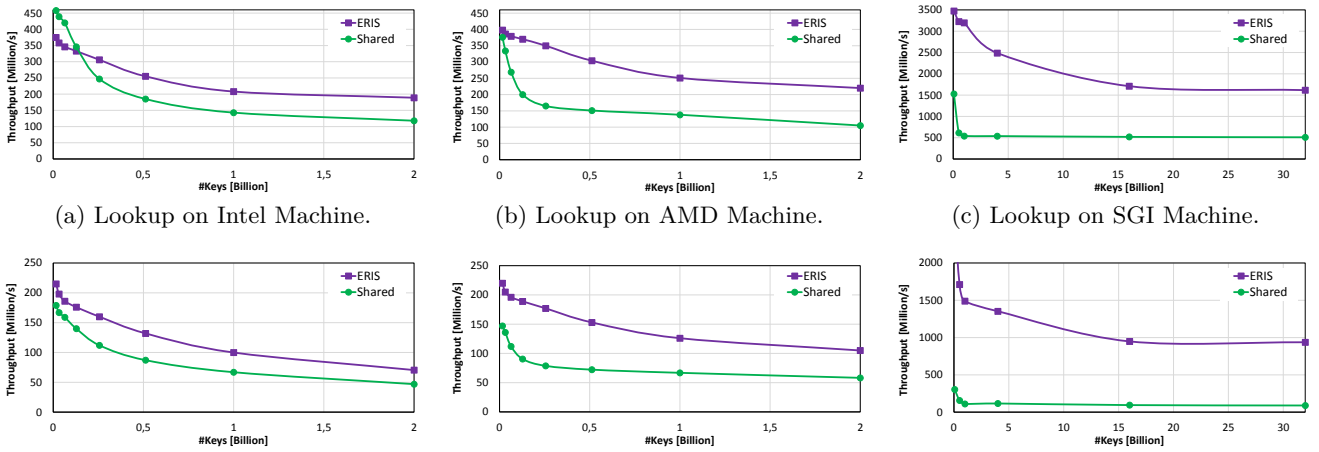
4.1 Evaluation Setup

All our experiments are executed on the three machines that were introduced in Section 2.2 , i.e., the Intel machine, the AMD machine, and the SGI machine (cf. Table 1). The executable files are compiled with the `g++` compiler, using optimization level `O3`. The shared index experiments are executed with `numactl -interleave=all` to interleave the memory across all available multiprocessors. Interleaving the memory resulted in slightly higher throughputs of the shared index compared to memory agnostic executions. A single benchmark run for upsert/lookup performance comprises two phases; (1) random keys are inserted into the index for about one minute and afterwards, (2) random keys are read from the index for another minute. Insert and lookup throughputs are reported for the two phases respectively. In the static workload cases, keys are uniformly distributed across the dense key domain. If not stated otherwise, the prefix trees are configured with a prefix length of 8bit. In a scan performance benchmark run, a column with random entries is generated and afterwards scanned repeatedly for one minute.

The throughput of storage operations is measured and reported by the application itself. We use different tools to measure the utilization of the links that connect multiprocessors, the open source tool `likwid` [1] on the Intel and the AMD machine and the SGI tool `linkstat-uv` on the SGI machine. Hardware performance counters are evaluated with `likwid` on the Intel and the AMD machine and `VampirTrace` [2] on the SGI machine.

4.2 Static Workload Experiments

For all static workload experiments, the load balancer is deactivated and the workload does not change over the whole benchmark run, i.e., keys to upsert or lookup are evenly distributed across the key domain and scans are over the full key domain.



(a) Lookup on Intel Machine. (b) Lookup on AMD Machine. (c) Lookup on SGI Machine.

(d) Upsert on Intel Machine. (e) Upsert on AMD Machine. (f) Upsert on SGI Machine.

Figure 8: Lookup/Upsert Throughput Depending on Index Size.

4.2.1 *Point Access with Different Index Sizes*

In the first set of experiments, we compare the lookup and upsert throughput of ERIS to the shared index for different index sizes. The results are shown in Figure 8. The index sizes vary from 16 million keys to 2 billion keys on the AMD and Intel machine and from 16 million keys to 32 billion keys on the larger SGI machine. As a reference, 1 billion keys require approximately 25GBs in memory and in our largest experiment, the index is as big as 0.8TB. Figure 8(a) shows that for small indexes on the small machine, the shared index outperforms ERIS. The reason for this is the small overhead that is introduced by the NUMA-optimized data command routing in ERIS. However, as the number of multiprocessors increases and with larger indexes, ERIS clearly supersedes the shared index. While on the eight node AMD machine, ERIS has a throughput that is about 1.6 times higher than the shared index (Figure 8(b), 1 billion keys), on the larger SGI machine, ERIS executes already 3.5 times as many lookups per second as the shared index (Figure 8(c), 16 billion keys). Finally, Figure 8 shows that the upsert performance behaves similar to the lookup performance, except the lower absolute throughput values.

4.2.2 *Scan Performance*

In the second experiment, we compare ERIS' scan performance with two different other memory allocation strategies for column data. The results on the SGI machine are shown in Figure 9. In the experiment, all AEUs or parallel thread respectively[1] scan a column with about 8 billion entries. The memory for the column data is allocated (1) on one single multiprocessor (*Single RAM*), (2) interleaved on all multiprocessors (*Interleaved*), or (3) on the multiprocessor where the AEU is executed (*ERIS*). In the *Single RAM* case, the scan performance is bound by the read bandwidth of the memory controller (cf. Table 2). The scan performance with interleaved memory is bound by the different link bandwidths. Only ERIS is able to achieve optimal scan performance. Figure 9 shows that ERIS achieves a 6.6 times

[1]488 cores, or 61 multiprocessors, is largest possible working set in the batch system on our SGI machine.

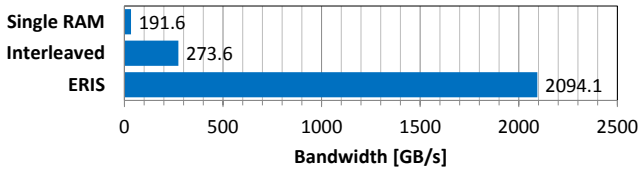


Figure 9: Scan Bandwidth of ERIS Compared to Naïve Memory Allocation Strategies on SGI.

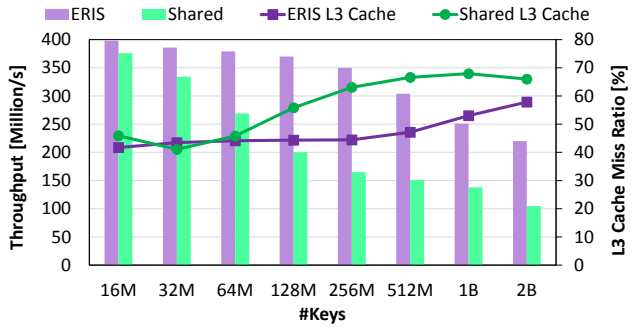


Figure 10: L3 Cache Miss Ratio on AMD.

higher bandwidth than does reading from memory that is interleaved over all multiprocessors. In the past, interleaving has often been proposed as a method of choice to overcome NUMA effects. However, our experiment clearly shows the drawback of such an approach.

4.2.3 *L3 Cache Usage*

To better understand the lookup performance of ERIS and the shared index, we investigate the L3 cache usage in this experiment. For smaller index sizes, larger portions of the upper levels of the prefix trees fit in the caches. For larger index sizes, the last level cache plays a minor role and the performance is memory bound. It can be seen in Figure 8 that for increasing index sizes, the performance of the shared index is earlier memory-bound than the performance of ERIS. The explanation is that ERIS makes better

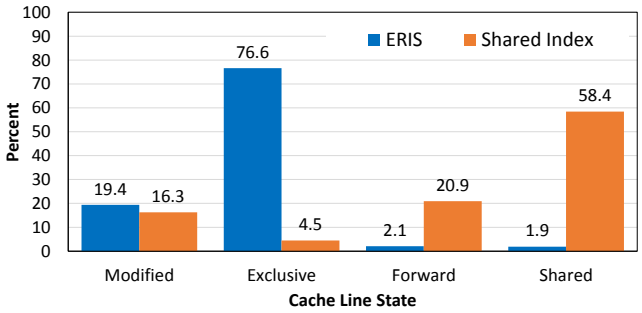


Figure 11: L3 Cache Line States on Intel - Percentage of all Hits (1B Keys).

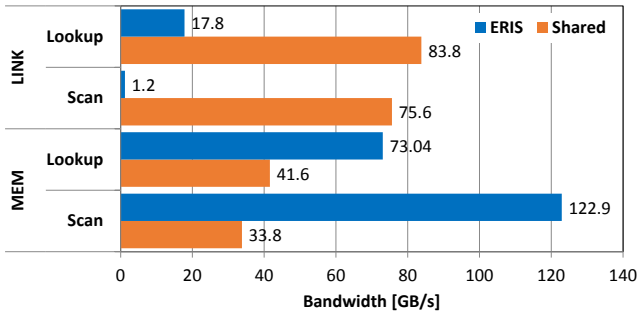


Figure 12: Link and Memory Controller Activity on AMD (Scan: 8GB, Lookup: 1B Keys).

use of the L3 cache. Because each AEU in ERIS serves a distinct partition and hence a subset of the tree, there is better data locality and less concurrency for the L3 cache. Consequently, the upper levels of the tree fit in cache for larger index sizes (see, e.g., [12] for details on cache concurrency effects). To verify our theory, we have calculated the L3 cache miss ratio for different index sizes on the AMD machine[2]. Furthermore, we have evaluated the state of the cache line for each L3 cache hit (for availability of the respective counters, this is evaluated on the Intel machine[3]). Figure 10 shows that the shared index causes a higher miss ratio for smaller indexes, compared to ERIS. This is supported by Figure 11, which shows that for the shared index 79.3% of all hits are on *Shared* or *Forward* cache lines which implies that the same cache line is present in another cache. Cache lines that are kept in multiple caches reduce the effective size of all caches and increase the miss ratio. ERIS on the other hand has significantly better data locality, which can be seen in Figure 11 where 97% cache hits go to cache lines in *Modified* or *Exclusive* states.

4.2.4 Link and Memory Controller Usage

ERIS is designed such that it reduces communications between multiprocessors. The significantly higher throughputs of ERIS as well as the L3 cache usage already suggest that this goal is achieved. However, to further verify our theses, we measure the average link usage over all links in a 10 seconds steady state window[4]. The results for the AMD machine are shown in Figure 12. The shared index has to transfer a total of 83.8 GB/s to fulfill all remote memory requests. At the same time, ERIS only transfers 17.8 GB/s (mainly caused by the data command routing facility) while at the same time achieving a higher throughput and thus performing more memory operations. The shared scan with interleaved memory allocation transfers a total of 75.6 GB/s, compared to 1.2 GB/s transfered by ERIS. These numbers together with the low level results in Section 2 explain the bad throughput of the shared setup. Each remote access suffers from the worse latency and bandwidth compared to local memory accesses.

[2]The L3 cache miss ratio is calculated as the quotient of the following hardware counters: `L3 Cache Misses` and `Request to L3 Cache` [4].

[3]The cache line states are measured using the `LLC_HITS` counter extensions in the C-Box of the Intel CPU [11].

[4]We measure the link usage by evaluating the `Link Transmit Bandwidth` counters of the AMD CPU [4].

Together with the link utilization, we have measured the transfer bandwidth of the memory controllers[5]. The results are also shown in Figure 12. On average, only 1 out of 8 memory requests of the shared setup go to local memory and remote memory requests suffer from higher latencies. Therefore, the shared index can only issue enough memory requests to transfer an average of 41.6 GB/s from all memory controllers while ERIS is able to transfer 73.0 GB/s. The shared scan produces a transfer rate of only 33.8 GB/s, compared to 122.9 GB/s transferred by ERIS. The transfer rate of ERIS' scan operator equals 93.6% of the possible accumulated memory bandwidth of the system (cf. Section 2).

4.3 Dynamic Workload Experiments

In this section, we show that ERIS is able to keep a high throughput even under changing workload conditions. For our experiments, we use a workload that randomly accesses the full key range (lookup) of 512 million keys for an initial period of 10 seconds. After this period, the workload changes drastically such that only half of all keys (in the range from 128M to 384M) are accessed afterwards. In the remaining time of the experiment, the workload is changed 4 more times with 20 seconds between any two changes. These remaining changes are only slight changes which are simulated by shifting the key range of interest by 8 million to the left.

Figure 13 shows the lookup throughput of ERIS over time. The above described workload changes are easily recognizable as short drops in the throughput curve. The chart contains performance numbers for a baseline run without load balancer and for three different load balancing algorithms (see Section 3), the One-Shot algorithm as well as Moving Average algorithms with window sizes of 1 and 8. The One-Shot algorithm causes the deepest drop of the throughput after each workload change, because all repartitionings that are necessary to regain a fully balanced workload are executed at once. This causes large overhead (some partitions need to be copied) but at the same time results in the fastest recovery time. The chart shows that the throughput reaches its maximum again shortly after each workload change. The other extreme is the MA1 algorithm, which only slightly adapts the partitioning in each evaluation period. Hence, the performance does not drop that drastically, but it takes more time before the maximum throughput is reached again. The MA8 algorithm appears to be the best

[5]We measure the memory controller by evaluating the `DRAM Accesses` counter of the AMD CPU [4].

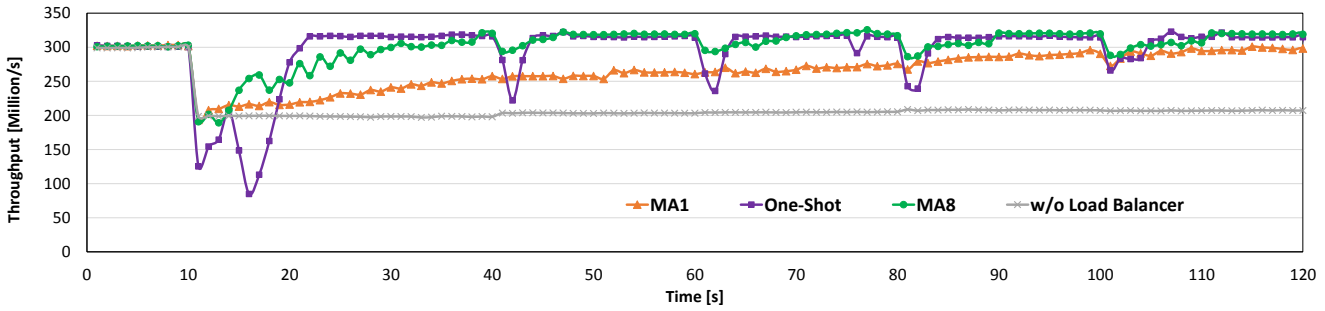


Figure 13: Load Balancer Experiments on AMD Machine.

compromise in this setup between performance drop and recovery time on that specific system.

As a conclusion, we note that the MA load balancing algorithm, with a parameter that depends on the machine, offers the best performance. Moreover, the parameter can be used to shift the behavior between gentle performance drops and quick recovery times, depending on the constraints of the application running on top.

5. RELATED WORK

Non-uniform memory access as well as thread-placement in multiprocessor systems have been studied in the operating systems and high-performance computing communities for several years. A more recent result is from Hackenberg et al. who have investigated the low-level memory performance and cache coherence effects at the granularity of single cache lines [8]. Blagodurov et al. propose a NUMA-aware scheduler and user-level scheduling on NUMA systems, though not specific to a certain application domain [5, 6].

The consequences of NUMA architectures for database management systems have been investigated as well, e.g., by Porobic et al. [22]. The authors perform a detailed analysis of different shared and distributed data deployments in NUMA systems. To show the performance impact of the NUMA architecture on a DBMS, they use Shore-MT as a scalable storage manager and TPC-C as an OLTP workload. Kiefer et al. have investigated memory access characteristics and cache effects in NUMA systems based on a synthetic benchmark that mimics a database system's behavior as well as a real database benchmark [12].

Several papers have investigated NUMA-aware algorithms or single database operators. Albutiu et al. propose a parallel, NUMA-aware sort-merge join for main memory database systems [3]. Lang et al. presented a NUMA-aware hash join operator for an in-memory DBMS [14]. Pandis et al. argue in favor of NUMA-aware algorithms in database systems by investigating a shuffle algorithm [17]. All papers conclude similar design principles for NUMA-aware algorithms which we also propose at the end of Section 2. However, ERIS is designed to push down NUMA-awareness deep into the storage layer to support a wider range of problems (i.e., queries) by highly optimizing the building blocks, specifically scan, lookup, and insert/upsert.

We have already mentioned the DORA system and its NUMA-aware extension ATraPos [21] in Section 1. In contrast to ERIS, these systems focus on disk-based database systems and transactional workloads, which differ in fundamental point from purely in-memory analytical workloads.

Finally, Leis et al. took a first step towards a NUMA-Aware query processing by proposing a NUMA-aware query evaluation framework [15]. This approach divides the base data of an operator into morsels (batches) and employs a work stealing approach to elastically schedule operators at runtime. To achieve NUMA awareness, the runtime scheduler tries to run operators close to the data and operators store their results in the local main memory. However, the paper does not cover the crucial point of this approach that is the partitioning of the base data that needs to be adapted efficiently at runtime. Moreover, intermediate results are not load balanced and critical data structures like hash tables are stored in a NUMA-agnostic fashion.

6. CONCLUSIONS AND FUTURE WORK

We showed with a detailed analysis of NUMA system architectures and low-level benchmarks that even today's server systems suffer from up to 10 times higher latencies and as little as 11% of the maximum bandwidth when accessing remote memory compared to local memory access. Based on these measurements, we derived that a NUMA system should be treated like a distributed system and that any application must be designed for memory locality and with optimized explicit communication between multiprocessors to assure a scalable performance on NUMA platforms. In this paper we presented ERIS, a NUMA-aware purely in-memory storage engine for tera-scale analytical workloads that is based on a data-oriented architecture. ERIS uses a NUMA-optimized high-throughput data command routing as well as a configurable NUMA-aware load balancing algorithm to achieve a maximum of parallelism to execute analytical queries with low latency. Our analysis showed that ERIS greatly improves memory locality and cache usage and thus scales even on large-scale NUMA platforms.

Since ERIS only provides storage operation primitives, we plan to implement a query processing framework on top of ERIS to evaluate the performance of more complex queries. Query processing with ERIS requires techniques for distributed systems and poses additional challenges for load balancing. Since a full balancing is not always possible, we want to to explore how AEU idle times can be leveraged for storage maintenance and reorganization. Another important research direction is how to realize energy awareness on such a data-oriented architecture, because AEUs always run at full speed and are thus consuming a high amount of energy. Here, we want to investigate the impact of frequency scaling, different scheduling policies, foreign memory accesses, and load balancing on the energy consumption.

7. ACKNOWLEDGMENTS

This work is partly funded by the German Research Foundation (DFG) in the Collaborative Research Center 912 "Highly Adaptive Energy-Efficient Computing" and under project number LE 1416/22-1, as well as by the Bundesministerium für Bildung und Forschung via the research project CoolSilicon (BMBF 16N10186).

8. REFERENCES

[1] likwid. `http://code.google.com/p/likwid/`.

[2] VAMPIRTRACE. `http://www.tu-dresden.de/zih/vampirtrace`.

[3] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. In *VLDB*, 2012.

[4] AMD. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors, Rev 3.12*, 10 2012.

[5] S. Blagodurov and A. Fedorova. User-level scheduling on NUMA multicore systems under Linux. In *Linux Symposium*, 2011.

[6] S. Blagodurov, S. Zhuravlev, A. Fedorova, and M. Dashti. A Case for NUMA-aware Contention Management on Multicore Systems. In *PACT*, 2010.

[7] M. Böhm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner. Efficient In-Memory Indexing with Generalized Prefix Trees. In *BTW*, 2011.

[8] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *MICRO*, 2009.

[9] Hypertransport Technology Consortium. *HyperTransport I/O Link Specification*, revision 3.10c edition, 2010.

[10] Intel. *An Introduction to the Intel QuickPath Interconnect*, 2009.

[11] Intel. Intel Xeon Processor E7 Family Uncore Performance Monitoring Programming Guide. Technical Report April, Intel Corporation, 2011.

[12] T. Kiefer, B. Schlegel, and W. Lehner. Experimental Evaluation of NUMA Effects on Database Management Systems. In *BTW*, 2013.

[13] D. Kim et al. Design and Analysis of 3D-MAPS (3D Massively Parallel Processor with Stacked Memory). *IEEE Transactions on Computers*, 2013.

[14] H. Lang, V. Leis, M.-C. Albutiu, T. Neumann, and A. Kemper. Massively Parallel NUMA-aware Hash Joins. In *INDM*, 2013.

[15] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In *SIGMOD*, 2014.

[16] J. J. Levandoski, D. B. Lomet, and S. Sengupta. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *PVLDB*, 6(10), 2013.

[17] Y. Li, I. Pandis, R. Müller, V. Raman, and G. Lohman. NUMA-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[18] I. Loi and L. Benini. An Efficient Distributed Memory Interface for Many-core Platform with 3D Stacked DRAM. DATE, 2010.

[19] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-Oriented Transaction Execution. In *VLDB*, 2010.

[20] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. PLP: Page Latch-free Shared-everything OLTP. In *VLDB*, 2011.

[21] D. Porobic, E. Liarou, P. Tözün, and A. Ailamaki. ATraPos: Adaptive Transaction Processing on Hardware Islands. In *to appear in ICDE*, 2014.

[22] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki. OLTP on Hardware Islands. In *VLDB*, 2012.

[23] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-Memory Scan Sharing For Multi-Core CPUs. *PVLDB*, 1, 2008.

[24] J. Rao and K. A. Ross. Making B+-Trees Cache Conscious in Main Memory. *SIGMOD Rec.*, 29, 2000.

[25] Technical advances in the sgi uv architecture. white paper, SGI, 2012.