Dieses Dokument ist eine Zweitveröffentlichung (Postprint) /

This is a self-archiving document (accepted version):

Tim Kiefer, Thomas Kissinger, Benjamin Schlegel, Dirk Habich, Daniel Molka, Wolfgang Lehner

ERIS Live: A NUMA-Aware In-Memory Storage Engine for Tera-Scale Multiprocessor Systems

Erstveröffentlichung in / First published in:

SIGMOD/PODS'14: International Conference on Management of Data, Snowbird 22.-27.06.2014. ACM Digital Library, S. 689–692. ISBN: 978-1-4503-2376-5

DOI: https://doi.org/10.1145/2588555.2594524

Diese Version ist verfügbar / This version is available on:

https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-803930

SLUB Wir führen Wissen.

TECHNISCHE UNIVERSITÄT DRESDEN

QUCOSA Quality Content of Saxony

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", S. 689–692, ISBN 978-1-4503-2376-5
https://doi.org/10.1145/2588555.2594524

ERIS Live: A NUMA-Aware In-Memory Storage Engine for Tera-Scale Multiprocessor Systems

Tim Kiefer*, Thomas Kissinger*, Benjamin Schlegel*, Dirk Habich*, Daniel Molka†, Wolfgang Lehner*
*Database Technology Group/†Center for Information Services and High Performance Computing (ZIH)
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, a DBMS has to allow flexible data partitioning and data placement at runtime.

In this demonstration, we present ERIS, our NUMA-aware in-memory storage engine. ERIS uses an adaptive partitioning approach that exploits the topology of the underlying NUMA platform and significantly reduces NUMA-related issues. We demonstrate throughput numbers and hardware performance counter evaluations of ERIS and a NUMA-unaware index for different workloads and configurations. All experiments are conducted on a standard server system as well as on a system consisting of 64 multiprocessors, 512 cores, and 8 TBs main memory.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design

Keywords

ERIS; NUMA; In-Memory; Storage Engine; Multiprocessors; Scalability

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 which 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, NUMA systems worsen the already bad scalability of latches and atomic instructions in multithreaded applications. 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. Thereby, a flexible data partitioning and data placement is needed to restrict memory accesses to the local main memory of a multiprocessor and to circumvent the latching of data structures.

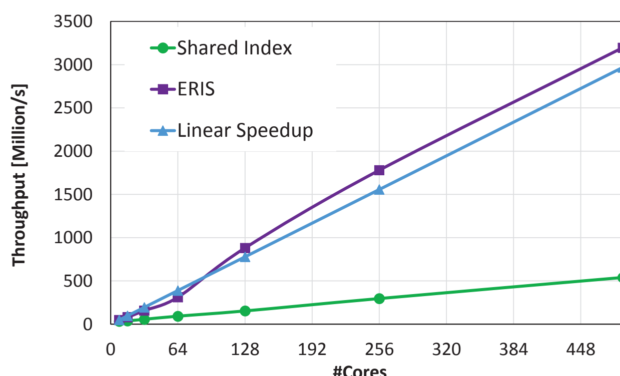

Figure 1: Super Linear Index Lookup Scalability of ERIS on a SGI UV 2000.

Our flexible in-memory storage engine—ERIS—was designed to achieve a high read and write throughput on tera-scale multiprocessor systems. ERIS picks up the concept of the data-oriented architecture (DORA) [3], which uses a thread-to-data instead of the conventional thread-to-transaction assignment. DORA demonstrated that it is beneficial to adaptively partition data objects on a single multicore system to reduce lock contention in the context of OLTP workloads. With ERIS, we extend the data-oriented architecture to scale-up on large multiprocessor systems and propose a load balancing mechanism that exploits the topology of the underlying NUMA platform to allow a quick adaption of the partitioning to a changing workload.

©2014 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in Mensch und Computer2020 (MuC'20), September 6–9, 2020, Magdeburg, Germany.
SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

DOI: http://dx.doi.org/10.1145/2588555.2594524.

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", S. 689–692, ISBN 978-1-4503-2376-5
https://doi.org/10.1145/2588555.2594524

In our evaluation, ERIS achieves a super linear speedup for the 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. While ERIS is able to scale up on such a platform, a NUMA-unaware storage engine that shares data objects between transactions struggles with NUMA-related issues that prevent it from scaling. In contrast, the data-oriented approach of ERIS employs logical partitioning of the data object, which aligns to the topology and thus avoids remote memory accesses, utilizes the individual multiprocessor caches better, reduces cache coherence overhead, and allows accesses to the data structure without any latches or atomic instructions like compare-and-swap.

For this demonstration, we built an interactive GUI that connects to an instance of ERIS (or the NUMA-unaware storage engine) running on a NUMA system. The GUI displays live measurements of the request throughput (in this demo, we focus on inserts and lookups) as well as hardware performance counters of link and memory controller usages. Furthermore, the GUI can be used to interactively change setup, workload, and load balancing parameters to observe the behavior of ERIS under changing conditions.

2. ERIS

In this section, we describe the architecture of ERIS and its individual components as visualized in Figure 2. The central components of the storage engine are the Autonomous Execution Units (AEU). AEUs can be implemented in many different ways. For this demonstration, we decided to use prefix trees [1], which are order-preserving and—contrary to the B+-Tree—unbalanced. 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 share the same local main memory and are thus able to quickly exchange data partitions. 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., inserts, lookups, and scans) on these partitions. To efficiently route data commands between AEUs, ERIS includes a NUMA-optimized data command routing component. The 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.

2.1 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 the data is highly discouraging when running on NUMA platforms, because data is distributed in an uncoordinated way across the memories of different multiprocessors. This in turn causes a high number of remote memory accesses by the transaction threads.

For that reason, ERIS employs a data-oriented architecture where each data object is logically partitioned. Each AEU gets a set of disjoint partitions (range partitioning) assigned—each one 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.

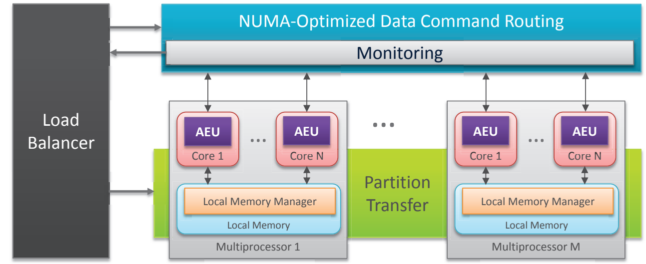

Figure 2: Architectural Overview of ERIS.

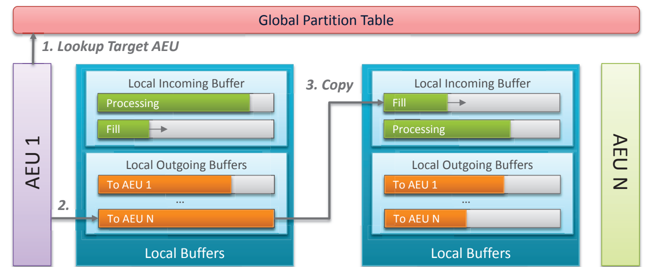

Figure 3: Two-Level Data Command Routing.

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. 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 to the local main memory. Additionally, our memory managers use thread-local caching to scale with a high number of cores per multiprocessor.

2.2 NUMA-Optimized Data Command Routing

The data command routing is an essential part of ERIS, because AEUs have to be supplied with data commands just in time. Thus, the main goal of the data command routing is to distribute data commands at a high throughput, otherwise the routing would become a bottleneck and would limit the scalability of ERIS. Our data command routing mechanism is shown in Figure 3. The core component is the global partition table (GPT), which keeps track of the partitioning of the individual data objects. To implement the GPT of ERIS, we decided to use a CSB+-Tree [4], because it works fast for sparsely distributed data and it scales better with an increasing number of ranges, respectively AEUs, compared to a simple array.

Besides the GPT, our NUMA-optimized data command routing uses a two-level buffering strategy. Each AEU uses a set of small outgoing buffers—one for each running AEU in the system—and two bigger incoming buffers. Both 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, it starts with looking up the responsible AEU(s) for that data command in the GPT (step 1 in Figure 3). As soon as the the target AEU is determined, the source AEU writes the data command to its corresponding outgoing buffer, which is pri-

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", S. 689–692, ISBN 978-1-4503-2376-5
https://doi.org/10.1145/2588555.2594524

vate to the AEU (step 2). If an outgoing buffer is either full or a timeout is reached, the specific outgoing buffer 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.

While outgoing buffers are private to an AEU and thus do not require any concurrency control, incoming buffers are written by different AEUs and are read by the host AEU in at the same time. For that reason, incoming buffers need an efficient and ideally latch-free concurrency control mechanism. Here, we employ an adapted version of the latch-free multi-buffer proposed in LLAMA [2].

2.3 Load Balancing

ERIS requires a load balancer component to adapt the partitioning to changing workloads. Based on the access frequencies captured in the monitoring component (cf. Figure 2), the load balancer periodically checks the load of ERIS for imbalances. If necessary, the load balancer executes a load balancing algorithm that calculates a new partitioning. Based on the optimization goal, the partitioning can either be optimized for throughput or for energy-efficiency. In the case of a throughput optimization, hot data is evenly distributed across the AEUs. To optimize for energy-efficiency, some AEUs only receive cold or even no data at all to allow the operating system to throttle the corresponding cores or to turn cores or multiprocessors completely off. 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.

Load Balancing Algorithms.

The load balancing algorithm receives the approximated access frequency distribution of the recent sample period as well as the current partitioning as inputs and outputs the targeted partitioning. The most aggressive, but also most costly approach is taken by the *One-Shot* load balancing algorithm. This algorithm computes the average access frequencies of all partitions and calculates a target partitioning that is fully balanced. The *One-Shot* algorithm is suitable for workloads that change rarely but heavily. An alternative algorithm uses a moving average (*MA*) over a subset of partitions to calculate the target partitioning from the current access frequencies. The *MA* algorithm 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. The aggressiveness of the *MA* algorithm depends on the window size and turns into the *One-Shot* algorithm for a sufficiently large window.

Partition Transfer.

If the load needs to be balanced, each AEU that has to grow or shrink its local partition receives a balancing command. Besides the information about the new partition range, a balancing command includes a set of transfer commands. There are two different types of transfers, depending on the distance between source and target AEU. When both are located on the same multiprocessor, the cheap *link* mechanism can be used, whereas the *copy* mechanism is used for transfers between multiprocessors. For the *link* transfer mechanism, the source AEU unlinks the respective portion of the partition and the target AEU adds it to its own partition. This operation is very cheap for the prefix trees used in our AEU implementation. A *copy* operation requires a cooperation of source and target AEU to avoid remote memory accesses. The target 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 and links it to its existing partition.

3. DEMONSTRATION

Figure 4 shows a screenshot of our demo application. The various details shown in the GUI as well as the demo experience are detailed in the following sections.

3.1 Demo Setup

In our demo, we directly connect to a NUMA system to measure and display the transaction throughput (1 in Figure 4) achieved by ERIS or a NUMA-unaware shared index. In addition to throughput values, our GUI shows link and memory controller utilization (2 in Figure 4) using hardware performance counters to reason about the algorithms' behaviors. To convey our load balancing algorithms, repartitioning commands (3 in Figure 4) are also visualized during the demo.

We use two different NUMA systems for our demo. The first machine (*AMD machine*) has 8 multiprocessors, 64 cores, and 64 GBs main memory. The second machine (*SGI machine*) is a SGI UV 2000 [5] with 64 multiprocessors, 512 cores, and 8 TBs main memory.

3.2 Demo Walkthrough

The visitor of our demo will be able to choose one of the two available machines (4 in Figure 4) as well as one of the two basic implementations (5 in Figure 4), i.e., ERIS and the NUMA-unaware shared index. Both machines show NUMA effects to a different extend and comparing the rather small AMD machine with the large SGI machine demonstrates the scalability of ERIS and the lacking scalability of the shared index, respectively.

When the visitor has decided in favor of a machine and an algorithm, the demo tool will start with filling the index with random keys. This prepares the following lookup of keys and at the same time the insert throughput can be observed. Following the insert phase, the demo tool will query random keys, evenly distributed over the key range.

At all times, the GUI charts live throughput values as well as link utilization and memory controller bandwidths. Comparing different setups, the demo visitor will see that ERIS causes very little link activity compared to the shared index. At the same time, ERIS is able to achieve a considerably higher memory controller bandwidth and hence transaction throughput. Both hardware performance measures reflect the significantly better data locality and hence lower communication penalty of ERIS.

Starting from a basic setup, the demo visitor has several options to interactively influence the course of the demo and to observe the algorithms' behaviors.

Number of Partitions/Multiprocessors.

The first choice is the number of AEUs (6 in Figure 4) and therefore partitions. At the same time, the number of

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", S. 689–692, ISBN 978-1-4503-2376-5
https://doi.org/10.1145/2588555.2594524

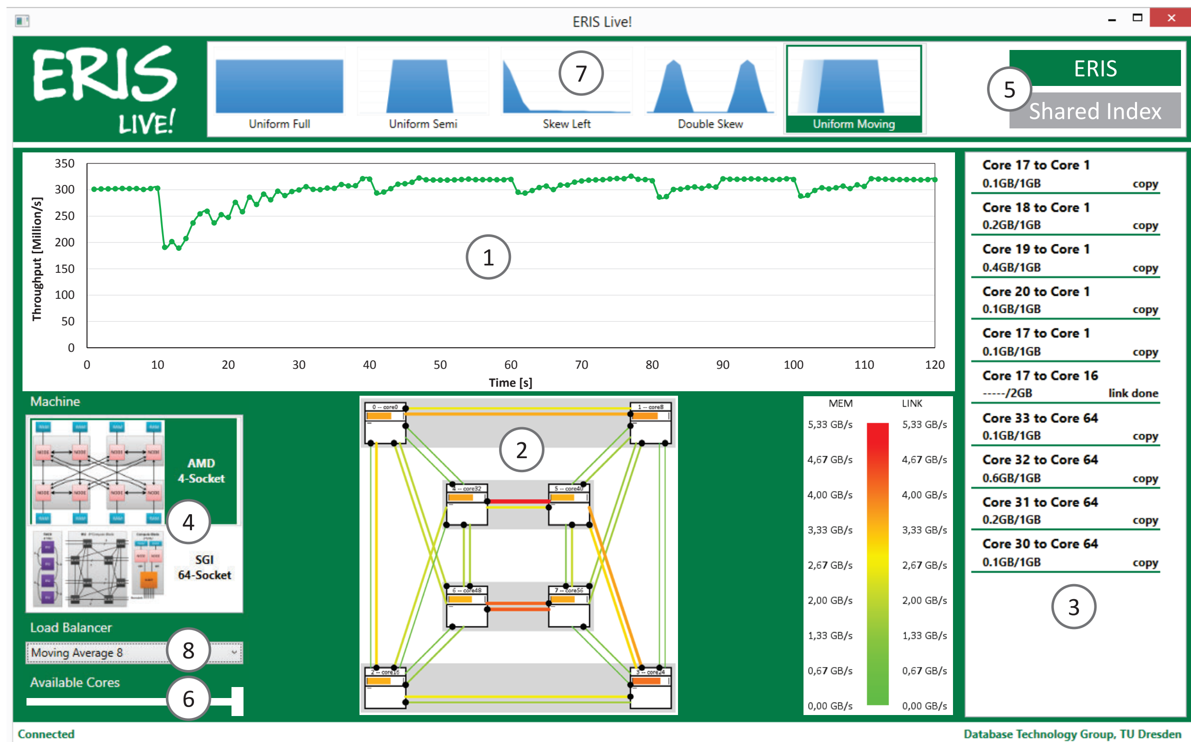

Figure 4: Screenshot of the ERIS Live Demo Application.

partitions directly determines the number of multiprocessors that are used by the algorithm.

By adding or removing partitions and multiprocessors respectively, the demo visitor can observe that the throughput of ERIS scales linearly with the number of multiprocessors.

Skewed Workloads.

The second interactive choice is the workload (7 in Figure 4). The demo visitor can switch between a workload that evenly queries the whole key range and several skewed workloads with hot spots in the key range. While the shared index is agnostic to workload changes and has an equally low throughput, the demo visitor can see how the load balancer of ERIS adapts to new workload conditions. Load balancing commands (i.e., commands to transfer a range of keys from one AEU to a different one) are visualized on a logical level. At the same time, the charted throughput values as well as the link and memory controller utilizations reflect the load balancing activities.

Load Balancing Algorithms.

The third choice in our demo setup is the specific load balancing algorithm (8 in Figure 4) and hence related to the choice of the workload. The demo visitor can choose between no load balancing at all or from different load balancer implementations such as the *One-Shot* algorithm or the *MA* algorithm with different parameters. Whenever the workload is changed, the behavior of the currently selected load balancing algorithm can be observed and the demo visitor can easily see their differences like gentle or steep performance drops and quick or slow recoveries.

4. CONCLUSIONS

In this demo proposal, we introduced ERIS, our NUMA-aware in-memory storage engine. ERIS is designed for flexible data partitioning and data placement and restricts memory accesses to the local main memories of the multiprocessors. To achieve scalability on tera-scale NUMA systems, ERIS implements NUMA-aware thread and memory management, a highly optimized command routing, and NUMA-aware load balancing mechanisms.

5. 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).

6. REFERENCES

[1] 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.

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

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

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

[5] SGI. Technical Advances in the SGI UV Architecture. white paper, SGI, 2012.

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden