Adaptive Access Path Selection for Hardware-Accelerated DRAM Loads

Markus Dreseler, Timo Gasda, Jan Kossmann, Matthias Uflacker, and Hasso Plattner

Hasso Plattner Institute, Potsdam, Germany
markus.dreseler@hpi.de

Abstract. For modern main memory database systems, the memory bus is the main bottleneck. Specialized hardware components of large NUMA systems, such as HPE's GRU, make it possible to offload memory transfers. In some cases, this improves the throughput by 30%, but other scenarios suffer from reduced performance. We show which factors influence this tradeoff. Based on our experiments, we present an adaptive prediction model that supports the DBMS in deciding whether to utilize these components. In addition, we evaluate non-coherent memory access as an additional access method and discuss its benefits and shortcomings.

1 Introduction

Current in-memory databases are significantly limited by the main memory's latency and bandwidth [2]. In the time spent for transferring a cache line from DRAM to the CPU (roughly 100 ns), a modern CPU can execute 300 instructions or more. When the compute part of database operators executes in fewer cycles, the CPU stalls and waits for more data to arrive. This gets exacerbated in NUMA setups where remote DRAM accesses take roughly 200 ns with a single NUMA hop. Scale-up systems, as used for big SAP HANA or Oracle databases, can include multiple NUMA hops and up to 48 TB of memory. These connect up to eight blades with four processors each to a single, cache-coherent network using a proprietary interconnect. In such setups, memory latency from one end to the other can reach hundreds of nanoseconds, making the influence even bigger.

Closely related to memory latency is memory bandwidth. On our test system (cf. Section 3), we measured a NUMA node-local bandwidth of slightly over 50 GB/s, while remote accesses on the same blade had a reduced bandwidth of 12.5 GB/s and remote blades of 11.5 GB/s. As such, making good use of the available physical bandwidth is vital. Doing so includes reducing the amount of data transferred by using compression for a higher logical bandwidth (i.e., more information transferred per byte) or organizing the data in a cache line-friendly way. This could be a columnar table layout where each cache line only holds values from the column that is accessed and *cache line bycatch*, i.e., data that is loaded into the CPU but never used, is avoided for column store-friendly queries.

Making the DBMS more aware of NUMA can significantly improve the performance [7]. By ensuring that data is moved across the NUMA network only

Accepted at Australasian Database Conference.
The final authenticated version is available online at https://dx.doi.org/10.1007/978-3-319-92013-9_1

when it is unavoidable, the memory access costs can be reduced. Still, there remain cases in which a load from a distant node cannot be avoided. This happens when joins access data from a remote table or when the data (and thus the load) is imbalanced and an operator cannot be executed on the optimal node.

In addition to NUMA optimization and better data layouts, developers use dedicated hardware to increase the effective bandwidth [1, 6, 9]. There are several approaches, but no one-size-fits-all technique. In this paper, we look at one specific method that is used to improve the physical bandwidth available to database operators, namely the Global Reference Unit (GRU) built into systems like SGI's UV series or HPE's Superdome Flex. The GRU provides an API that can be used to offload certain memory operations, allowing the CPU to work on other data in the meantime. Previous work [3] has shown that this can result in a performance benefit of up to 30% for table scans. We extend on this by evaluating which factors lead to an advantage of the GRU over the CPU in some cases and what causes it to be slower in others. This knowledge can be used by the DBMS to automatically choose between the CPU and GRU access paths. Furthermore, we present relaxed cache coherence as another access method.

This paper is organized as follows: Section 2 gives background information on the hardware discussed in this paper. To gather data on the memory bus utilization and better profile the physical properties of database operations, we use performance counters as described in Section 3. These are then used in Section 4 to discuss the factors that influence if one method or another gives the higher effective bandwidth. Section 5 explains how a DBMS can use these results in order to choose an access method. In Section 6, we show how relaxing cache coherency could further improve the physical bandwidth of a table scan. Related work is discussed in Section 7 and a summary is given in Section 8.

2 Hardware used for accelerating DRAM reads

In the previously mentioned scale-up systems, four NUMA nodes (i.e., processors) are grouped into blades as shown in Figure 1. Each node is connected via QPI links to other processors on the same blade, however the two diagonal QPI connections are omitted. The free QPI port on each processor is then connected to one of the two so-called HARPs that are part of each blade.

HARPs connect the entire scale-up system, as each HARP is directly connected with every other using a special interconnect called NUMAlink. This creates an all-to-all topology that allows the addition of more CPUs and more memory by attaching additional blades to the machine. In order to make the memory of one blade accessible to another blade, the HARPs participate in the QPI ring of their blades and mimic a NUMA node with a large amount of main memory, i.e., the memory of every other blade [8].

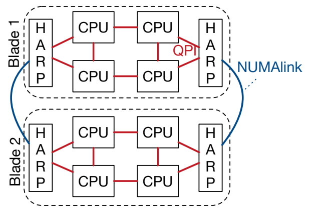

Fig. 1: General architecture of the discussed system

Accepted at Australasian Database Conference.
The final authenticated version is available online at https://dx.doi.org/10.1007/978-3-319-92013-9_1

The component in the HARP that is responsible for transparently extending the memory space and translating memory addresses is the Global Reference Unit (GRU). In addition to this transparent access, the GRU also provides a user-level API that enables developers to instruct the GRU directly.

Previous work has shown that using the GRU API for database operations improves throughput by up to 30% when scanning remote data [3]. For a more in-depth discussion of the GRU hardware and the actual implementation of the GRU scan, we refer to that paper. This performance advantage is achieved by using the `gru_bcopy` function, a block copy instruction that is executed by the GRU, and a double-buffered scan. We divide the data vector into equally sized chunks, and allocate two temporary buffers of that chunk size. The asynchronous bcopy operation is used to copy chunks from the data vector into one of the local buffers, which is processed while bcopy is filling the other buffer. The buffers are switched and the process is repeated until the entire vector is scanned.

There are three reasons why this can achieve better performance compared to a regular CPU scan: Firstly, bcopy executes asynchronously and allows the CPU to run other computations while the GRU handles the memory transfer. As a result, the CPU can process one local chunk while the GRU is loading the other chunk in the background. Secondly, the HARP can access remote memory more efficiently than the CPU. This is because a CPU's memory access performance is limited by the number of outstanding read requests that it can handle before it stalls and waits for loads to complete. Stalls are especially noticeable in large NUMA systems with high memory access latencies, because it takes longer for a read request to return and allow the next request to be issued. This means that for the systems discussed here, an increase in latency results in a decrease in bandwidth. The HARPs can handle more outstanding read requests than a CPU and can therefore achieve higher throughput. Thirdly, because the cache coherency directories (used to maintain a consistent view of data across processors) are located on the HARP, we expect the HARPs to handle cache coherency more efficiently. While this does not improve single operations, the decreased cache coherency overhead can improve memory performance over time.

3 Quantifying the Memory Bandwidth Utilization

To better utilize the available memory bandwidth, it is vital to understand how much of it is actually used and where the bottleneck is. This information is both needed by developers of database operators and by the DBMS itself when it decides which scan will be used. We use the Intel Processor Counter Monitor (PCM) to monitor the QPI traffic on relevant nodes.

For every QPI link, PCM can measure the amount of both incoming data and outgoing traffic. Incoming data (*dataIn*) only includes the transferred payload. Outgoing traffic (*trafficOut*) includes both the payload as well as any overhead such as cache coherency traffic. The incoming traffic or the outgoing data cannot be retrieved from PCM, but can be computed in most cases. Also, the information from where and to where data flows is not available.

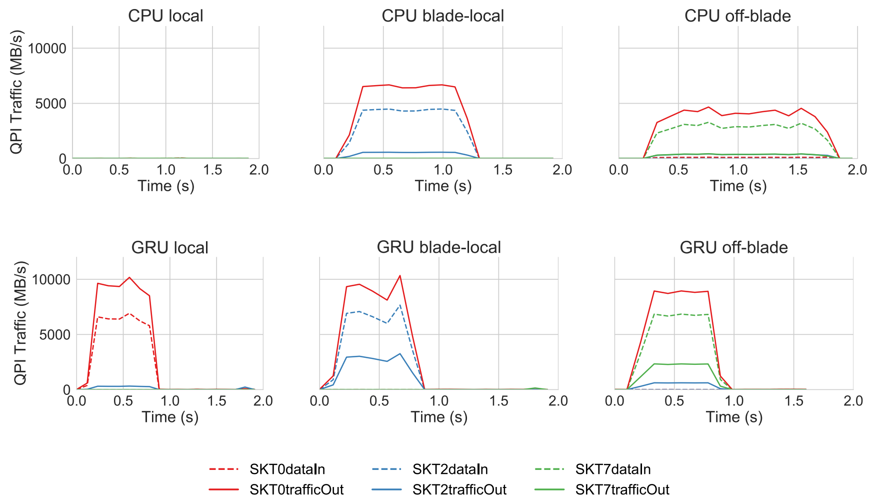

Fig. 2: QPI traffic when scanning a 4 GB vector of integers

Figure 2 displays the QPI traffic as measured[1] by PCM for a scan on a 4 GB vector of integers, i.e., approximately one billion integers. In the benchmark, the same scan is executed three times, each time with a different memory placement relative to the executing thread. After each run, the CPU caches are flushed using `CLFLUSH`. The data is always located on NUMA node 0 (”storage node”), while the executing thread is pinned to nodes 0, 2, and 7 (”execution node”) for the node-local, blade-local, and off-blade cases respectively. For the local scan using the CPU, no QPI traffic is seen because the traffic is node-local. The GRU implementation, on the other hand, copies data from the storage node to the execution node even when these are identical. This explains why a GRU scan is detrimental for local data. Continuing to blade-local CPU scans (top center), we see two significant lines, one for outgoing traffic of the storage node, and one for incoming data of the executing node. The difference between the two can be attributed to the cache coherency overhead. When using the GRU, the overall bandwidth is higher, resulting in a lower execution time. A similar image can be seen for off-blade scans. Here, the bandwidth difference between CPU and GRU is even more pronounced. The fact that Socket 2 appears in the graph is surprising at first. We attribute it to additional caches that we have no control over and take this as a reason to look into non-cache coherent loads in Section 6.

The same library can be used to get numbers on the current load on the memory bus. We gather information about the QPI utilization of all relevant nodes and feed it into the adaptive model described in Section 5.

[1] All benchmarks were executed on an SGI UV 300H with 6 TB RAM and eight Intel E7-8890 v2 processors. Our code was compiled with gcc 7.2 at `-O3`.

4 Factors that influence the CPU/GRU Tradeoff

In this section, we describe how the throughput of the GRU scan is affected by different factors and how deciding between GRU and CPU scans is only possible by looking at a combination of these factors. We classify these factors into two groups: Internal and external factors. The former are parameters that come from the scan itself, such as the size of the table. The latter are unrelated to the particular scan operator, for example the utilization of the system.

We use the table scan as an example for an operator with sequential access patterns. Compared to operators such as the join, it is strictly memory-bound, so the influences of improved memory bandwidth utilization are better to see. Latency-bound operators cannot be improved by the bcopy approach.

4.1 Internal Influences

Data Size One of the most important factors is the size of the scanned table. Figure 3 shows how the throughput of different access methods is influenced by the size of the table when accessing an off-blade table. Both approaches reach their maximum throughput only when the table has a certain size. For GRU scans, this happens when the data size reaches at least 60 MB. CPU scans deliver the maximum throughput for smaller table sizes, approximately 1 MB.

This can be explained with fixed setup costs for the table scan as well as having to wait for the first cache lines to arrive from a remote location. For the GRU scan, the additional system calls required to obtain the execution contexts and to execute the bcopy method mean that bigger tables are needed to reach the break-even point.

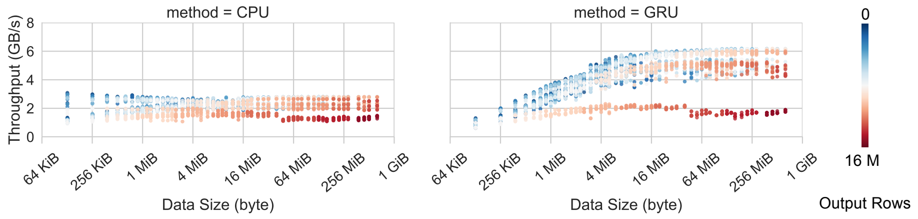

Fig. 3: Influence of the input data size for off-blade CPU and GRU scans - each dot is one measured data point

Data Locality Depending on where the input data is located relative to the executing CPU, it needs to be transferred through zero to multiple NUMA hops. Figure 4 shows that the throughput for the regular CPU scan changes depending on NUMA distance. The highest throughput of 8 GB/s is achieved on the same

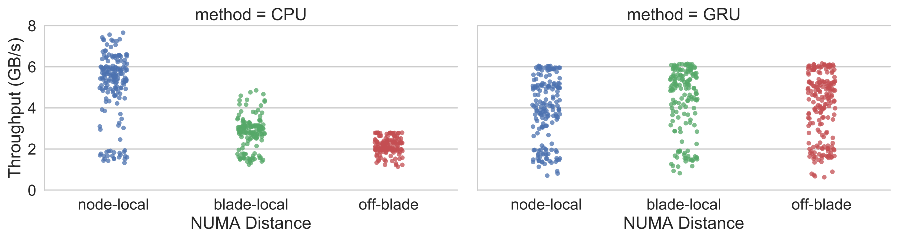

Fig. 4: Influence of the NUMA distance for CPU and GRU scans

node. With increasing NUMA distance, the throughput rates decrease. For a blade-local scan, the throughput rates reach up to 5 GB/s, and scanning an off-blade vector only nets approximately 3 GB/s. The GRU scan performance stays stable for all NUMA distances at around 6 GB/s. For both CPU and GRU, a high variance is measured. This is dependent on the other execution parameters as described in this section. It shows that there are parameters other than the data locality, especially for small tables, that play an important role in deciding if the CPU or the GRU is faster.

For the model described in Section 5, we take the latency (instead of the number of hops) between source and destination node as it describes a linear variable, not a discrete one. This makes it easier to adapt the model to other systems where the latency between hops is different.

Result Size When scanning the input data vector, i.e., the column in an in-memory database, the operation also needs to save the results. In this implementation, the scan returns a vector of indexes of the input vector where a certain search value was found. This means that both the value distribution and the given search value have an impact on how large the result gets. We have chosen to take both the data size and the result size as parameters instead of just using the selectivity. This is because the impact of the selectivity on the scan cost varies for different data sizes.

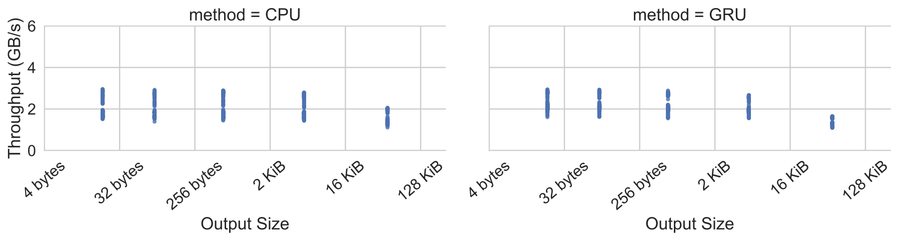

Fig. 5: Influence of the output size when scanning an off-blade vector of 512 KiB

Figure 5 shows the performance of scans with different result sizes. As the output size grows, the amount of time spent for writing the output vector slowly grows as well, and at some point, surpasses the value comparisons in terms of runtime. Consequently, after that point the benefits gained from our improved scanning method become insignificant.

4.2 External Influences

Background QPI Traffic In a production environment, the executed operator is not the only memory-intensive operation on the system. Therefore, it makes sense to also take a look at the influence of memory traffic caused by other processes. To identify how this affects performance, we use PCM (as described in Section 3) to measure QPI traffic right before executing our scan operations.

For the benchmark, we generate background traffic by using two types of background workers. One copies data between the worker and the data node and then scans the remote data with a SIMD scan on 32 bit integers. The second worker uses bcopy to asynchronously copy data back and forth between the worker and data node. This generates both QPI and Numalink traffic.

By doing so, we can vary the background traffic and measure the performance for varying QPI loads. Our measurements show that the impact of high QPI utilization is higher on CPU scans. If no parallel workers are consuming QPI bandwidth, the throughput is unaffected. In the worst case, a busy QPI interface decreases the throughput by 2 GB/s. For the GRU, scans on a high-load system only have a throughput that is 1 GB/s lower.

Background HARP Traffic Because the GRU is responsible for referencing memory of other blades, it is involved in memory accesses even when the GRU API is not used. Consequently, the load of the GRUs has an effect on other memory operations in the system as shown in Figure 6. Different from previous figures, all combinations of data sizes and data locality are combined in the graph. For GRU scans, the scan throughput stays in a small window just below 7 GB/s when the GRU is not busy. When other explicit GRU operations are run simultaneously, the throughput is reduced significantly. For the CPU scans, background GRU traffic does not affect the maximum throughput as much.

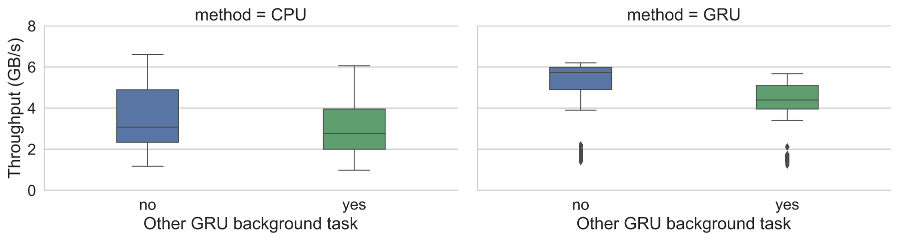

Fig. 6: Influence of a GRU background traffic for vectors bigger than 32 MB

5 Self-Tuning Access Path Selection

The measurements presented in Section 4 have shown that directly utilizing the GRU has performance benefits in some situations. In others, such as with local data, small tables, or a high number of output results, the CPU is preferred. We have identified the most important parameters that impact GRU performance with microbenchmarks. This chapter introduces a cost model for table scans. Such a model can be used by the query optimizer or scheduler of a database system to decide whether to use a GRU or regular table scan. The features of the model are the above presented influence factors. Because different database servers have different characteristics, the model can be trained for other systems easily. This allows the DBMS to adapt to varying factors, such as NUMA topologies and latencies.

5.1 Scan Decision Model

The model is part of the query optimizer where estimated execution costs are calculated before every scan execution. A vital requirement for the model is a low prediction time. In addition, models vary for different hardware characteristics. Therefore, a new model is trained for each machine individually which demands for low training times and robustness. Thus, we decided to use linear regression. For these models, inference is achieved by multiplying the observation values with their respective feature coefficients. The overhead of such calculations is negligible compared to query execution times.

We measure the execution times for a large number of randomized feature combinations. By repeating the experiments for each combination at least 50 times with and without GRU, we collect more than 200 000 observations and minimize variability effects. The resulting data is used to create models that predict the runtime of normal and GRU scans. Table 1 shows the models' weights as calculated on our system.

When generating the query plan, the optimizer collects the necessary parameters for inference from the database's statistics component and the PCM API. The retrieved parameters are applied to both models which return predicted execution times. Based on these, the faster scan is scheduled/planned.

Model	Data Size	Data Locality	Result Size	QPI Traffic	HARP Traffic
Scan Runtime CPU	174.091	33.335	96.436	-2.135	-4.049
Scan Runtime GRU	345.849	-6.756	332.176	3.215	5.942

Table 1: Weights of linear regression model features

5.2 Evaluation

To validate the models used for decision-making, we evaluate their accuracy in this section. The R^2 score is our metric to determine how well the model explains the observed runtime. We also consider the model's precision, recall, and the F1 score to judge the decision between normal and GRU scan. These metrics were evaluated with a training and testing set split of 80% and 20%.

Table 2 shows that both models can explain over 93% of the occurring variance with the above presented features. For the GRU, the model is slightly more accurate with a higher R^2 coefficient. This is mostly due to the stability of GRU operations with different NUMA distances or latencies as seen in Figure 4. Precision, recall and F1 score are used for classification models and only the combination of both models, as described above, can be used for classification.

We also compared our linear regression model to another lightweight model, a *decision tree*. Some of the presented scores outperform linear regression slightly, but our linear regression model offers more flexibility since it returns run times and not only a decision on which of the scan types is more efficient. In addition, predictions can be made faster. Since variance is not an appropriate metric for classification models, R^2 is not reported for the decision tree.

As an additional quality metric, we looked at how close to optimal the performance of such a self-adapting DBMS is. For this, we chose between CPU and GRU using the two models described above. Over all table scans in the test set, the performance was within 2% of the best possible solution.

Model	R^2	Precision	Recall	F1 score
Scan Runtime CPU	0.9347	0.9	0.77	0.83
Scan Runtime GRU	0.9671			
Decision Tree	–	0.87	0.95	0.91

Table 2: Cost model accuracy

5.3 Model adaptivity

We have discussed the overhead of predicting the runtime for new observations above. Another important factor for the viability of using the model is the overhead caused by training and observation gathering. The training for linear regression models of such size is negligible, but generating large numbers of observations is time-consuming. Since the results highly depend on the specific hardware configuration, it is necessary to calibrate the model on each particular system upfront. To speed up this process, the database system can run the measurements for only a subset of features with a reduced number of parameters at system start and train models based on this small amount of observations. Later, during normal system operation, further observations can be collected. With these new observations, we improve the model by adapting it to the specific hardware configuration. This adaptive approach also ensures that the model is accurate when the data held in the database itself changes since data characteristics are one of the considered features. Unfortunately, because we only have access to HARP-based systems with the same hardware layout, we cannot evaluate this for varying hardware configurations.

6 Non-Cache Coherent GRU Scan

So far, we have looked at cache-coherent memory accesses where the CPUs coordinate to make sure that all cores access the same version of the data. This greatly simplifies multithreading and many DBMS programmers (us included) would not want to work without it. In certain cases, however, cache coherency is not needed. If a part of the data reaches a state in which it will not be modified anymore, cache coherency increases the communication overhead and uses up bandwidth that could be used for actual data.

An example for this can be found in databases that use insert-only Multiversion Concurrency Control (MVCC) [5]. Instead of physically updating rows, the database invalidates them by setting a flag for that row and inserts the new version at the end of the table. This means that the data part of the row will not be modified until it is removed as part of some clean up job.

For cases like this, where the life time of an area of memory is well-defined, cache coherency could be seen as unnecessary overhead. Especially when data is stored in a large read-only main partition and a small differential buffer (delta partition), it is easy to see how most of the data does not require any cache synchronization. By using the GRU's `IAA_NCRAM` instead of the `IAA_RAM` access method, we can copy data without checking its cache coherency state. It is both undocumented and unsupported, likely because of the issues discussed next.

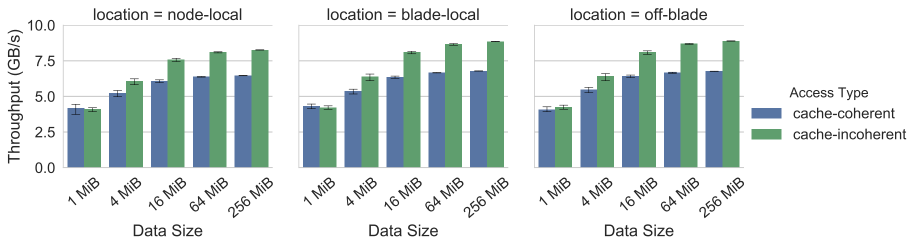

Fig. 7: Throughput of a cache-coherent and non-cache coherent GRU table scan

Figure 7 shows the performance of the previously discussed GRU bcopy scan compared to that of a non-coherent implementation. For this benchmark, the output size has been set to one element and no further background QPI traffic is present. The input data sizes and NUMA distances vary.

With an effective throughput of almost 9 GB/s for off-blade scans, the non-cache coherent scan is approximately 3.5 GB/s faster than the regular scan in the best case. Unfortunately, even though we have carefully flushed all buffers that we are aware of using `CLFLUSH`, `gru_vflush`, and `gru_flush_tlb`, the scan results are not completely correct. Most likely, there are still some buffers or directory caches that have not yet been flushed. Also, this instruction is not officially supported. As such, we did not include this access method in the previous sections and are not using it for actual table scans. Instead, we are presenting it here to show the potential of relaxed cache coherency.

7 Related Work

Copying or moving blocks of data is a very common operation, especially in applications that make use of buffering. Examples include the TCP/IP stack and applications in data centers [4]. Traditionally, copies use CPU instructions like `memcpy`, which blocks the CPU from doing other work. That is why some applications offload memory accesses to other components.

In the database world, an example are the database accelerators (DAX) built into Oracle's Sparc M7 processors [6]. These can be used to decompress, scan, filter, or translate the data in a table. Located on the memory controller, the DAX has direct access to the DRAM and, according to the documentation, access to additional bandwidth that is not used by the cores. In addition to a higher bandwidth, it can be used to free the processor to do other work and reduce cache pollution by only forwarding data that is actually needed [6]. Compared to the hardware-accelerated table scan discussed in this paper, the DAX has a bigger feature set and can be used to execute some database operators (mostly) without support from the CPU. This is different from our approach, where only the data transfer is offloaded to the GRU and the CPU still has to do the scanning.

Ungethüm et al. discuss adaptions to the Tomahawk platform, a multiprocessor system-on-a-chip [9] that add database-related primitives like hashing and sorted set operations to the instruction set of the processor. They optimize memory access by pushing down filters to a custom memory controller. It contains an intelligent DMA controller (iDMA) that is able to filter and search data before giving it to the processor. Instead of retrieving a large vector from the memory controller and have the CPU search for a specific value, this can be pushed down to the iDMA, which will then return the results to the application.

8 Summary

By directly instructing the memory hardware using the GRU API, the effective throughput of an in-memory table scan can be improved by up to 30%. In this paper, we have shown how this best-case result is influenced by a number of factors, both internal factors regarding the scan and external factors of the system. These include the size of the scanned table, the output size, but also the amount of parallel QPI traffic. For a DBMS to make effective use of this new access method, it has to take these factors into account and estimate the runtimes for normal and GRU supported scans. We have shown how a self-adapting model can be trained and how runtimes can be inferred using linear regression. This model can be used by a self-tuning database that automatically decides between regular and GRU scans. Compared to an omniscient model that always chooses the most efficient scan type, our model performs within 2% of the optimum as an evaluation on our test set demonstrates.

Furthermore, we have discussed non-cache coherent bcopy as a hardware method that could be considered for improving the effective memory throughput This delivers significantly higher bandwidth in cases where the DBMS can ensure that the data will not change during the execution of an operator. However, even though we have flushed all caches that we are aware of, the results were still slightly incorrect. Because of that, we do not use non-cache coherent bcopy at this point but suggest that this is an area worth exploring.

Future work shall be directed into exploring more hardware-supported memory access methods. We see great potential in allowing the DBMS developer to selectively relax cache coherency.

Acknowledgments. We thank Martin Boissier and Rainer Schlosser for their helpful input on the estimation model.

References

[1] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. “Integrating Compression and Execution in Column-Oriented Database Systems”. In: *ACM SIGMOD International Conference on Management of Data*. 2006.

[2] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. “Database Architecture Optimized for the New Bottleneck: Memory Access”. In: *25th International Conference on Very Large Data Bases*. 1999.

[3] Markus Dreseler et al. “Hardware-Accelerated Memory Operations on Large-Scale NUMA Systems”. In: *Eighth International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*. 2017.

[4] Annie P Foong et al. “TCP Performance Re-visited”. In: *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2003.

[5] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. “Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems”. In: *ACM SIGMOD International Conference on Management of Data*. 2015.

[6] Oracle. *Oracle’s SPARC T7 and SPARC M7 Server Architecture*. Tech. rep. URL: `https://www.oracle.com/assets/sparc-t7-m7-server-architecture-2702877.pdf` (visited on 12/06/2017).

[7] Iraklis Psaroudakis et al. “Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores”. In: *Proceedings of the VLDB* (2016).

[8] Greg Thorson and Michael Woodacre. “SGI UV2: A Fused Computation and Data Analysis Machine”. In: *International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012.

[9] Annett Ungethüm et al. “Overview on Hardware Optimizations for Database Engines”. In: *Datenbanksysteme für Business, Technologie und Web (BTW)*. 2017.