Hardware-Accelerated Memory Operations on Large-Scale NUMA Systems

Markus Dreseler Timo Djürken
Matthias Uflacker Hasso Plattner
Hasso Plattner Institute
Potsdam, Germany
{first.last}@hpi.uni-potsdam.de

Thomas Kissinger Eric Lübke
Dirk Habich Wolfgang Lehner
Database Systems Group
Technische Universität Dresden
{first.last}@tu-dresden.de

ABSTRACT

As NUMA systems grow in complexity, the average distance of memory increases. Because the number of parallel read requests is limited, this reduces the bandwidth available to the CPUs. Also, the cost of cache coherency increases as atomic operations are synced across multiple hops. We explore how these problems can be alleviated by offloading NUMA accesses to the interconnect hardware and show how databases can profit. For cross-NUMA table scans, we report a performance improvement of up to 30%; for atomic increments as used for transaction sequencing up to 10x, and for latches up to 8x. These experiments were performed on an SGI UV300 system but demonstrate the general value of explicit memory instructions.

1. INTRODUCTION

Modern business applications rely on responsive data analytics and transaction processing with a single source of truth. Motivated by this need and the availability of large main memory capacities, more and more customers of enterprise software move to state-of-the-art in-memory database systems. Due to the rapid growth of data as well as the increasing comprehensiveness of queries, the compute resources of a single server system are not sufficient anymore. Especially for in-memory database systems, main memory capacity and bandwidth needs to be scaled beyond the limit of a single box. For this, databases can either be scaled out or scaled up. While the scale-out solution requires significant efforts to adapt the database software (e.g., fault tolerance, explicit communication, Two-Phase Commit), large scale-up solutions provide an elegant way to scale existing software at lower costs [8].

The SGI UltraViolet family (recently acquired by HPE) is one example of such a large scale-up system. Those systems consist of multiple *individual rack units (IRU)* that are connected via a *NUMAlink* system interconnect. Figure 1 shows a block diagram of a single IRU of an SGI UV300. As illustrated, the *HARP* ASIC is a key component that connects the IRU's processors to the systemwide NUMAlink interconnect. Additionally, the HARP is responsible for maintaining cache coherency and providing a common address space across the IRUs. Since NUMAlink adds an additional layer to the interconnect hierarchy, remote memory accesses are becoming more costly, similar to a scale-out setup. Hence, such large scale-up NUMA systems face network topology related problems in terms of (1) *reduced bandwidth* and *increased latency* for remote memory accesses and (2) the strongly *limited scalability of atomic memory operations* (e.g., latches) as a consequence of the comprehensive cache coherency protocol. To cope with this issue, recent database research proposed adaptive partitioning strategies [2, 3, 5, 6] to decrease the number of remote memory accesses and strategies to reduce the need of atomic memory operations in general [1, 4]. However, due to varying workload patterns as well as shared data structures and intermediate results, neither remote memory accesses nor atomic memory operations can be completely avoided.

Aside from these software-based approaches, hardware vendors are also developing new features to efficiently address the network topology related problems. For instance, the HARPs in a SGI UV system employ a *Global Reference Unit (GRU)* facilitating a proprietary API to accelerate respectively offload memory operations within a NUMA ar-

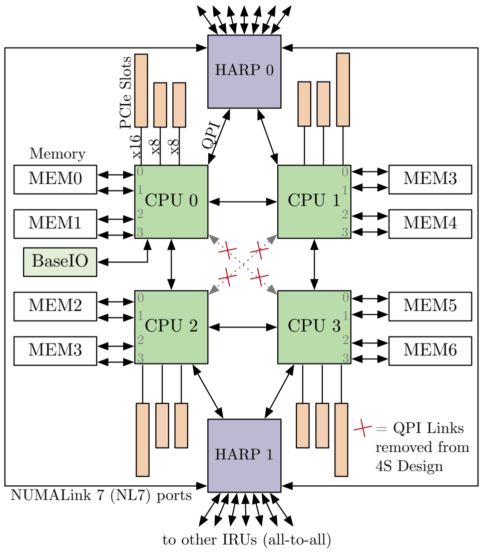


Figure 1: SGI UV300 block diagram, based on [9].

Accepted at Eighth International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS), in conjunction with VLDB

chitecture. In particular, the GRU provides functionality to asynchronously copy memory between processors and to accelerate atomic memory operations. In this paper, we treat the SGI UV and its GRUs as a playground to investigate the overall potential of hardware-accelerated main memory operations for in-memory database systems. In detail, we investigate the capabilities of the GRU to speed up database operations and remove bottlenecks in a typical in-memory database system.

The **contributions** of this paper are:

(1) We give an in-depth overview of the SGI UV architecture and detail on the functionality provided by the *Global Reference Unit (GRU)* API. Moreover, we demonstrate how to program the GRU.
(2) We evaluate the capability of the GRU to accelerate and offload processor-to-processor main memory copy operations.
(3) We evaluate GRU-accelerated atomic memory operations to eliminate usual bottlenecks in database systems such as obtaining transaction timestamps or protecting critical sections.

Outline. The remainder of this paper is structured as follows. In Section 2 we discuss the necessary background of the SGI UV and the GRU API. Afterwards, we evaluate the processor-to-processor copy operations and use cases for databases in Section 3 as well as atomic memory operations in Section 4. Finally, we discuss the related work in Section 5 and conclude the paper in Section 6.

2. BACKGROUND

As already mentioned, the HARP is the main component that connects multiple four-socket units (called IRUs or blades). This is achieved by bringing together two interconnect networks: QPI on the side of the IRUs, and the proprietary NUMAlink between all HARPs in the system. To plug into the coherent interconnect between the four CPUs within one IRU, the QPI connections between CPUs 0-2 and 1-3 are removed. This frees up one QPI port, which is then used to connect the CPUs to the HARPs (see Figure 1). Each HARP serves two CPUs, so that two HARPs are needed per IRU. On the NUMAlink side, the HARPs are fully connected. The two HARPs within one blade are also connected with each other via two NUMAlink connections.

A key component of the HARP is the Global Reference Unit. Every HARP contains two GRUs, each connected with one of the IRU's nodes via a QPI link. Their main function is allowing the connected CPU to address off-blade memory. The GRU presents itself as another participant in the QPI network. As such, it is responsible for translating addresses that are outside of the local IRU. This is done by looking up the data's location in the coherency directory (or a cached version thereof). Additionally, the GRU is responsible for transparently wrapping QPI stores and loads into NUMAlink network packages and unwrapping them on the receivers side.

In normal operation, HARP and GRU work together to provide a transparent extension of the memory space. Programmers do not need to be aware that the target system is comprised of multiple, cache-coherently connected four-socket systems. During execution, the only hint that the program is executed on such a machine is the increased latency of off-blade memory accesses, which is just under 500 nanoseconds. This latency also affects the maximum bandwidth because the number of concurrent read requests issued by the CPU is limited and longer waits mean more expensive memory stalls.

To better utilize the NUMAlink network, it can be profitable not to use the transparent translation, but to explicitly instruct the GRU. For this, SGI provides an API that can be used both in user and in kernel mode to directly interact with the GRU. The first part of the API consists of methods that create and manage the hardware resources needed for explicit GRU memory operations. The steps needed to prepare for GRU operations are documented in `man 7 gru`. We will focus on the second part of the API, which includes the memory instructions as described in Table 1.

In this paper, we will use the memory transfer and the atomic memory operations. These will be described in the appropriate sections in detail. All discussions are based on experiments that we ran on a UV 300 with four IRUs, each having four Intel E7-8890 v2 processors with 768 GB of DRAM at 1333 MHz. This sums up to 240 physical cores (480 logical) and 12 TB of main memory.

3. BCOPY

The first operation that we evaluate towards its potential for in-memory databases is `gru_bcopy`. Its function signature can be found in Table 2. Not to be confused with the deprecated string copy method from the C library, it is a method that uses the GRU to copy data from system memory to system memory. As such, its end result can be compared to that of `memcpy`. While `memcpy` causes the executing processor to execute a number of load operations from remote memory and blocks execution until all cache lines have been transferred, `gru_bcopy` handles the memory transfer asynchronously and outside of the processor. Since the nodes

Memory Transfer gru_bcopy, gru_bstore, gru_[i]vload, gru_[i]vset, gru_[i]vstore	Transferring data between system memory to the GRU or to a different area of system memory, initializing system memory with a set value; `"i"` versions use indirect (i.e., indexed) addressing.
Atomic Memory Operations (AMO) gru_game[r], gru_gami[r], gru_gamxr, gru_ivramiw, gru_ivramew	Support atomic changes to remote memory. Other than regular x64 atomic operations, which retrieve the memory and perform operations locally, these are executed by sending the appropriate command to the remote GRU.
Message Passing gru_mesq	A method specifically for MPI message queues, this method sends a message to a remote queue. Compared to software implementations that take five network traversals, this hardware-supported method only needs two.
Operative gru_nop, gru_vflush	As all GRU methods are executed asynchronously, nop is needed as a method to abort running instructions. vflush flushes one or more cacheline(s) from all nodes in the network.

Table 1: List of the data manipulation methods supported by the GRU API

`void gru_bcopy(gru_control_block_t *cb, const gru_addr_t src, gru_addr_t dest, unsigned int tri0, unsigned int xtype, unsigned long nelem, unsigned int bufsize, unsigned long hints);`	
`cb`	Pointer to the GRU ressource that controls execution, a so called control block.
`src`	Pointer to the data source.
`dest`	Pointer to the data destination.
`tri0`	Offset in the current GRU buffer. Can usually be ignored, i.e., set to zero.
`xtype`	Type of elements that should be copied. It's easiest to use `XTYPE_B`, i.e., bytes, here and specify the number of bytes in the next parameter, `nelem`. Using a larger type, e.g., `XTYPE_DW`, does not seem to have any impact on the performance.
`nelem`	Number of `xtype` elements that should be copied.
`bufsize`	Number of cache lines that can be used as a buffer. This size has significant influence on the performance. With the current maximum of 128 cache lines for a buffer size, i.e. around 8K, we observed a 22× higher throughput compared to the smallest possible buffer size of 4 cache lines.
`hints`	Bit mask for execution hints, currently only for ordering certain read references. Unused in this paper.

Table 2: The function parameters of `gru_bcopy`.

within one IRU are connected via QPI, using `gru_bcopy` can only bring a benefit when out-of-blade memory is accessed. How this is done is shown in Figure 2: (1) The `gru_bcopy` instruction is issued by the requesting CPU and sent to its HARP. (2) The GRU within the HARP identifies the remote node where the data is located and requests the cache line (CL) from the appropriate HARP. (3) The remote HARP uses its QPI network to load the cache line from the appropriate CPU. (4-6) That CPU uses a regular memory load to retrieve the memory from its system memory and returns it to its HARP. (7) The remote HARP sends the cache line to the local HARP, where it is stored in the HARP's buffer. (8-9) The local HARP transfers the cache line from its buffer to the targeted memory address.

`gru_bcopy` has three advantages over regular CPU loads from remote nodes: First, because its execution is asynchronous and can transfer multiple cache lines at once, the CPU gets freed up and can perform other computation in the time that would otherwise be lost in memory stalls. Second, CPUs are limited in their memory access performance by their maximum number of outstanding read requests. For current x86 CPUs, this means that ten L1D cache misses can be worked on at the same time. After this, the core has to wait for one request to finish. The effects of this can be seen when measuring the maximum bandwidth of a single core, which does not reach the possible bandwidth of the entire CPU. This problem gets worse with increased NUMA distances, as experienced in a massive scale-up system. When the memory latency doubles, the maximum number of outstanding requests would have to double as well to keep the bandwidth the same. Since this is not the case, the bandwidth of a single core is limited. The HARP, on the other hand, supports a significantly higher number of outstanding requests and can achieve a higher bandwidth when accessing remote memory. Third, using the GRU utilizes the directory-based cache coherency protocol in a more efficient way. While this does not directly affect a single memory access, it reduces the coherency effort, increasing the amortized performance. We will now look at how these three reasons translate into a measurable performance impact.

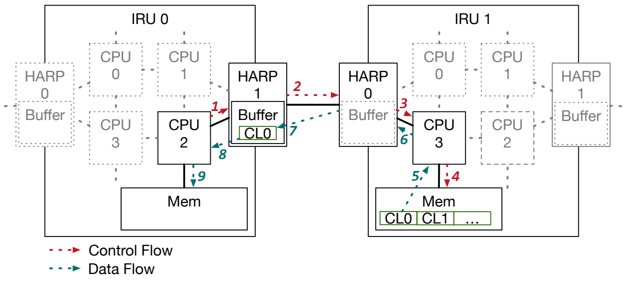


Figure 2: How `gru_bcopy` moves a cache line (CL) from remote memory to local memory.

3.1 Performance Comparison with memcpy

To get a baseline for the benefit of replacing CPU reads with GRU-supported memory transfers, we measured the maximum bandwidth when copying data from off-blade memory to local system memory. The results are shown in Figure 3, with the x-axis displaying the size of the transferred memory block and the y-axis the achieved memory throughput on a single core. Our findings are the following: (1) As expected, memcpy does not saturate the maximum NUMAlink bandwidth of 7.5 GB/s [9]. (2) Using `gru_bcopy` brings us significantly closer to the theoretical maximum, at least when large amounts of memory are transferred. (3) In that case, `gru_bcopy` has a 2.4x performance advantage over memcpy, which uses regular CPU stores and loads. For smaller chunks of data, `gru_bcopy` suffers from a performance hit of up to 4x.

These benchmarks have been executed without taking the constant cost of initializing the GRU control structures (i.e., control blocks and contexts) into account. This is because for databases, the initial subsecond cost is incurred only once during DBMS startup. The spike measured between 10^5 and 10^6 Bytes for `memcpy` is seen only on our test machine and has not been observed on single-blade systems. As it is not relevant to our findings, we will not discuss it further.

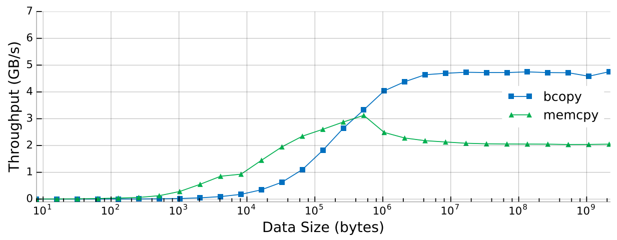


Figure 3: Performance comparison of `gru_bcopy` and `memcpy`.

3.2 GRU-Accelerated Off-Blade Column Scan for Database Systems

A main advantage of a column-oriented in-memory database is the support for unindexed table searches. This is made possible by having the prefetcher load data before it is needed, not loading parts of cache lines that hold irrelevant data (i.e., data from other columns), and a combination of an efficient dictionary compression and bit-packed attribute vectors [10]. Fast scans allow users to formulate queries that the database has not been optimized for and, in return, allows them to explore their data with new levels of freedom. As such, optimizing their performance translates to direct benefits for the users. While the maximum performance has already been achieved for node-local scans, this is not the case for off-blade scans in scale-up systems. These suffer significantly from the higher latency, which translates to a loss in bandwidth due to the memory stalls caused by a low number of supported parallel reads.

One might argue that in these cases, the execution of the scan should be moved closer to the data, and that optimizing the off-blade scan performance is unnecessary. Of course, performing only local scans is preferable. Still, there are cases in which, even with a NUMA-aware database, off-blade scans cannot be avoided: (1) if the cores of the remote node are busy themselves and moving the execution would lead to higher load imbalances, (2) if the scan is part of a larger chain of operators that is better suited for a different node, e.g., as an input to a following operator or in a JIT-compiled operator chain.

In these cases, `gru_bcopy` helps us by providing a higher bandwidth from the remote memory. Instead of directly accessing the remote memory, we first copy it to local, fixed-size buffers using a double buffering approach. This is shown in Figure 4. While `gru_bcopy` is moving data into one buffer, we can scan the data in the other. This is made possible by the asynchronous nature of the GRU operations. The scan is done by calling the regular scan operation on the buffers. As long as the buffer boundaries do not cut any values in half, this does not require any code modifications in the scan itself. For database vendors, this means that the deviation in the code base is limited to a single place, the GRU proxy. Listing 1 gives an example.

3.3 Benchmarks

For evaluating the performance of the GRU-accelerated column scan, we use the proprietary scan implementation of a commercial DBMS. Three questions are of interest here: (1) What is the maximum performance gain? (2) For which column sizes does the GRU scan make sense? (3) How must parameters, such as the GRU buffer size, be tuned for optimal throughput?

Figure 5 shows the results of scanning a single off-blade

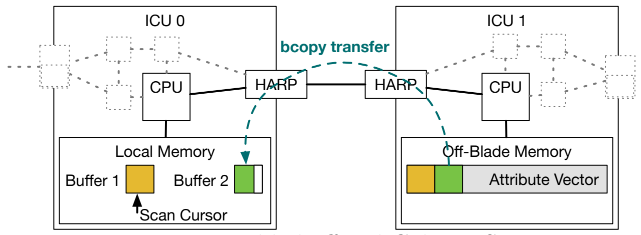


Figure 4: Double-buffered Column Scan.

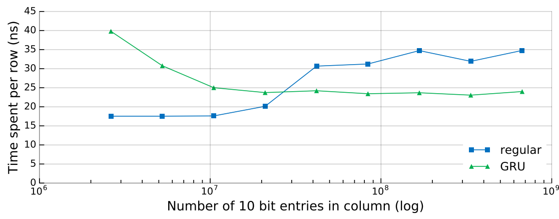


Figure 5: Double-buffered Off-Blade Scan with GRU.

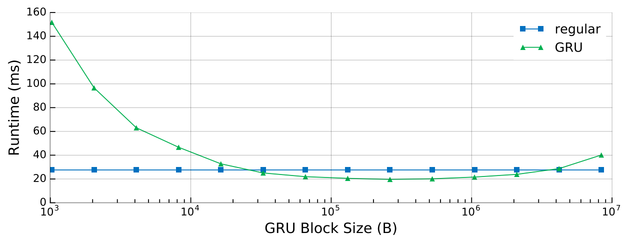


Figure 6: Runtime of a Column Scan depending on the chosen GRU Block Size.

column with a varying size being scanned either with or without GRU support. The y-axis displays the time spent *per row* during the scan. This allows for easier interpretation than absolute numbers, which would obviously increase with growing vectors. The results are very similar to those shown for the `memcpy` comparison in Figure 3. Again, the GRU shows a significant performance improvement over regular CPU loads. Here, we were not able to reach the maximum of 2.4x; instead, we see a performance improvement of 30% for two reasons. Firstly, the `memcpy` benchmark has a read:write ratio of 1:1, while the ratio of the scan depends on its selectivity (here 1000:1). For the CPU case, this means that more time can be spent reading data. Thus, the worst-case bandwidth of the CPU is 3.5 GB/s, compared to 2 GB/s. The `gru_bcopy` instruction, on the other hand, performs a streaming copy, which is why this effect does not play as much of a role. Secondly, the test machine only supports AVX, not AVX2, and we expect a more significant performance improvement with AVX2 [11]. A second finding is that using the GRU only makes sense for vectors with more than twenty million entries. With a bit case of 10, this makes $10/8 * 20,000,000 \approx 23MB$. While this appears to be a lot at first, many tables in an enterprise system as targeted by this architecture are bigger by orders of magnitude and easily reach multiple gigabytes.

We found one parameter to be of high importance for good performance with `gru_bcopy`: the size of our local buffers. As there is a certain fixed cost associated with each GRU transfer, the time needed to copy to a buffer that is too small is dominated by the fixed cost. For a buffer that is too large, offloading and double-buffering does not work properly, as there are longer time slots in which either the GRU or the CPU are idle. Figure 6 shows this effect. In this case, the optimal block size is around 262K.

4. ATOMIC MEMORY OPERATIONS

The *GRU API* provides instructions for executing atomic memory operations on a *single double word* or on *multiple double words*. In the following, we will focus on the single double word instructions as they are commonly used in

```
uintX_t *buffer_1 = (uintX_t)(numa_alloc_local(block_size));
uintX_t *buffer_2 = (uintX_t)(numa_alloc_local(block_size));
uintX_t *buffers[] = {buffer_1, buffer_2};

gru_bcopy(cbs[0], remote, buffers[0], 0, XTYPE_B, block_size, 128, 0);

for (size_t block = 0; block < block_count; ++block) {
  if (block < block_count - 1) {
      // fetch next
      gru_bcopy(cbs[(block + 1) % 2], remote + elements_per_block * (block + 1),
      buffers[(block + 1) % 2], 0, XTYPE_B, block_size, 128, 0); }

  // process current
  gru_wait(cbs[block % 2]);
  uintX_t *current_buffer = buffers[block % 2];

  for (int block_offset = 0; block_offset < elements_per_block; ++block_offset) {
    // regular scan operator comes here - this one is very simple
    if (current_buffer[block_offset] == search_value) {
        matches_out.emplace_back(RowID{0, block_offset}); }}}
```

Listing 1: A simplified implementation of a GRU-supported column scan

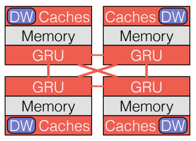
database systems and experimentally evaluate their feasibility. In particular, we will present the results for two critical components of an in-memory database system:

Atomic Counter Increments Atomically incrementing a single counter value is a critical operation for the transaction management but usually does not scale within a single processor and even less so in an entire NUMA system. For instance, multi-version concurrency control (MVCC) relies on counters for obtaining a transaction timestamp in a sequential order.

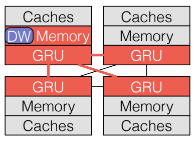
Latches Database systems employ latches to protect non-parallelized code sections against the concurrent access of multiple threads. Those code sections are usually global critical sections or fine-grained latches within single data structures such as trees or columns.

Table 3 lists the four available atomic single double word instructions of the GRU. Each instruction takes a pointer to the target double word as well as the specific operation as parameters. The return value of an instruction is written to the issuing *control block* after a successful completion. The respective instructions differ in two ways from each other:

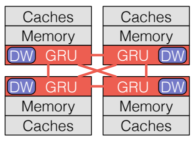
Explicit Operands. Operands for the respective instruction are either given implicitly or explicitly. Hence, instructions with implicit operands only support rather basic operations such as *fetch-and-increment* where the operand is implicitly given as a 1. Instructions with explicit operands allow more sophisticated operations such as *compare-and swap*, which needs two additional operands.



(a) CPU



(b) GRU



(c) Registered

Figure 7: Internal functioning of atomic memory operations. Schematic view of a 4-IRU NUMA system. Red system components are involved in the execution.

Registered/Buffered. While non-registered instructions directly operate on the main memory the targeted double word is located in, registered operations try to buffer referenced double word (DW) within the local GRU memory. Thus, the double word is likely to be cached in multiple GRUs.

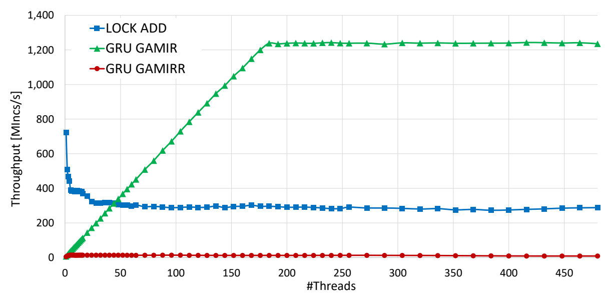
In Figure 7, we visualized the difference between *CPU*, *GRU*, and *registered GRU* atomic memory instructions. It schematically depicts a NUMA system consisting of four directly connected IRUs as well as the internal memory hierarchy of an IRU ranging from GRU memory, over local socket main memory, up to the local CPU caches.

The traditional way of executing atomic memory operations is via *locked CPU instructions* (e.g., a `LOCK ADD`). As shown in Figure 7(a), the actual operation is performed by the local hardware threads causing the referenced double word to be cached by multiple processors on multiple IRUs. The major drawback of this approach is that the processors and GRUs spend high efforts for maintaining the cache coherency across the individual caches of the hierarchy, which results in a high interconnect activity and instruction latency. In contrast, a basic GRU atomic memory instruction is exclusively executed by the GRU that is connected to the processor respectively memory that hosts the referenced double word. As shown in Figure 7(b), the referenced double word is cached nowhere, which saves the high overhead of the cache coherency protocol. Finally, the registered GRU instruction depicted in Figure 7(c) additionally buffers the referenced double word in the local memory of the individual GRUs and does not operate on the memory itself. Hence, the effective atomic memory operation is executed on the instruction issuing GRU, which requires additional efforts to ensure the internal memory coherency across the GRUs.

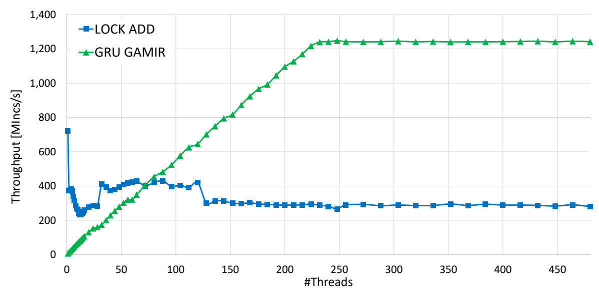
Experimental Setup. The overall aim of our following evaluation is to measure the scalability as well as the absolute throughput of the three different ways of executing atomic memory operations. As previously discussed (cf., Figure 7), the performance of the instructions depend on where the executing threads and the collocated GRUs are placed. Therefore, we employ the following two thread allocation orders for our evaluation:

GRU Instruction	Explicit Operands	Registered/Buffered	Supported Operations
`GRU_GAMIR`			Fetch, Clear, Fetch-and-Incr, Fetch-and-Decr
`GRU_GAMIRR`		✓	
`GRU_GAMER`	✓		Swap, Or, And, Xor, Add, Compare-and-Swap
`GRU_GAMERR`	✓	✓	

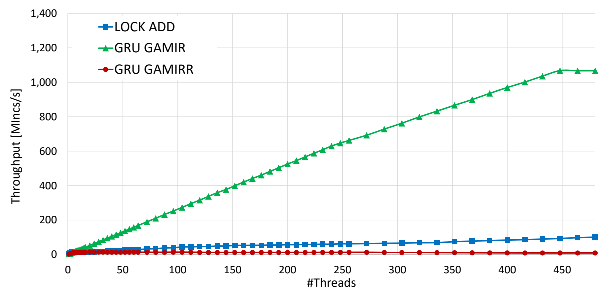
Table 3: Overview of single double word atomic memory operations.



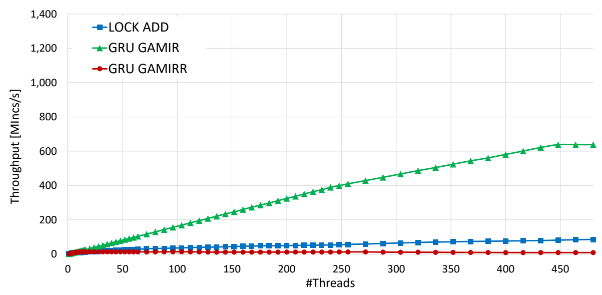
(a) *High* contention; *interleaved* thread allocation.



(b) *High* contention; *natural* thread allocation.



(c) *Medium* contention; *interleaved* thread allocation.



(d) *Low* contention; *interleaved* thread allocation.

Figure 8: Comparison of atomic counter increments using `LOCK ADD`, `GRU GAMIR` and `GRU GAMIRR`.

Natural Thread Allocation. This thread allocation order starts with filling the first socket (physical cores followed by HyperThreading siblings) and afterwards continues with the next socket.

Interleaved Thread Allocation. The interleaved thread allocation order allocates the threads in a round-robin fashion across the sockets, also starting with the physical cores followed by the HyperThreading siblings.

Moreover, we modify the contention on the referenced double word by adding artificial delays during the atomic memory operations. While the *high contention* setup uses no delay, the *medium contention* and *low contention* setups induce a medium respectively high delay. In the following, we will present our scalability and throughput results for *atomic counter increments* and *latches* as use cases for database systems.

4.1 Atomic Counter Increments

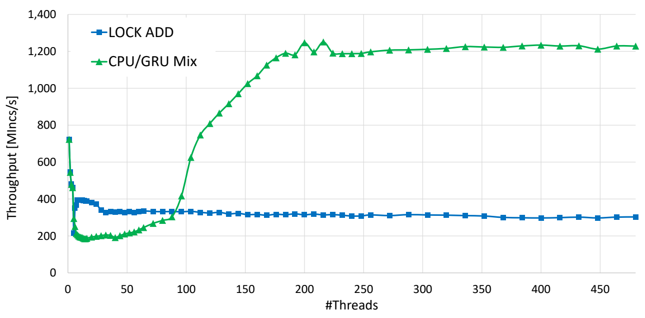
Obtaining sequentially ordered values respectively timestamps at a global level is a crucial bottleneck for database systems. Hence, in this section, we investigate if GRU-based instructions are able to relax the bottleneck compared to traditional atomic CPU instructions. In our experiments, the active threads try to increment a global counter while obtaining the previous value. The appropriate CPU instruction is a `LOCK ADD`[1]. The corresponding GRU instruction is `GRU_GAMIR` using the *fetch-and-increment* operation.

Figure 8 visualizes the results of the experiment. Each chart shows the throughput in millions of atomic increments per second for the CPU instruction (`LOCK ADD`), GRU instruction (`GRU_GAMIR`), and registered GRU instruction (`GRU_GAMIRR`). For all experiments, we varied the number of active threads allocated in the respective order.

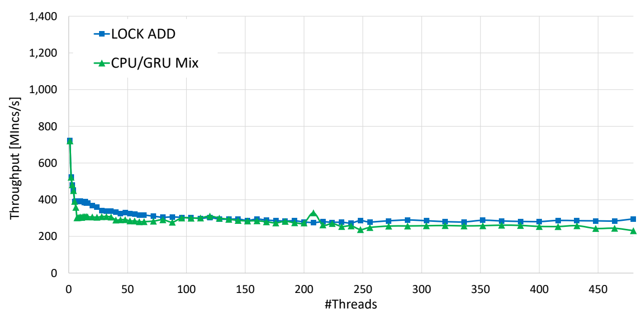
Figure 8(a) gives the measurements for the *high contention* setup using the *interleaved* allocation order. We observe that the CPU instruction achieves its peak performance with only a single thread and is continuously decreasing when activating more threads. In contrast, we observe an ideal scalability of the GRU instruction until the point of 184 threads is reached and the throughput remains almost constant. Due to the additional overhead of issuing GRU instructions compared to CPU instruction, we also observe a higher throughput of the CPU instruction with a low number of threads until the equilibrium is reached with 44 threads. The registered GRU instruction exhibits a low absolute throughput as well as a bad scalability.

Figure 8(b) shows the results for the *natural* thread allocation order. In general, we observe the same behavior as for the *interleaved* allocation order. Nevertheless, there

[1]GCC `__sync_fetch_and_add` intrinsic.

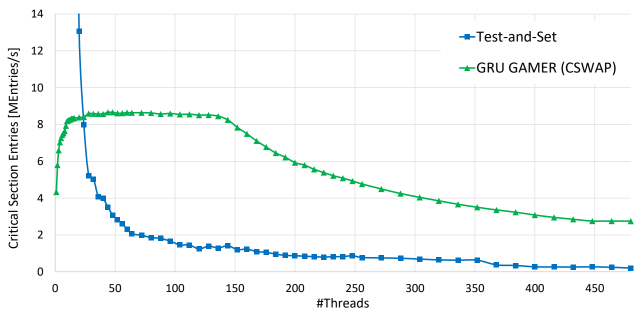


(a) NUMA socket 1 - 4 (IRU 1 only).

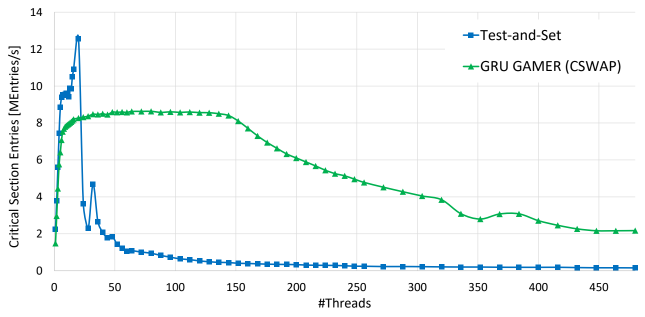


(b) NUMA socket 1 - 5 (IRU 1 and 2).

Figure 9: Comparison of atomic counter increments for a mix of `LOCK ADD` and `GRU GAMIR`. One thread of the respective NUMA sockets uses `LOCK ADD`. Remaining threads use `GRU GAMIR`. High contention and interleaved thread allocation.



(a) *High* contention.



(b) *Low* contention.

Figure 10: Comparison of a latch using `test-and-set` and `GRU GAMER (CSWAP)`. Interleaved thread allocation order.

exist two differences. (1) The GRU instruction scales stepwise, because threads of the same socket share the same GRU, and thus reaches its maximum throughput later with 232 threads. (2) For the CPU instruction, we observe two points where the throughput suddenly rises (31 threads) respectively drops (121 threads). Since each socket comprises of 30 hardware threads, the first effect happens when switching to the second socket of the first IRU and the second effect occurs as soon as a thread on the second IRU is activated.

In Figure 8(c) and 8(d) we stay with the *interleaved* allocation order and present the results for the *medium* and *low contention* setup. As both measurements show, the GRU instruction clearly outperforms the CPU instruction in terms of scalability and absolute throughput. However, the CPU instruction still exhibits a better throughput with a low number of threads, but the turning point is already reached with 6 threads. The registered GRU instruction remains at a low throughput.

In a final experiment, we reflect on the internal functioning of the GRU by mixing CPU and GRU instructions. As a specific test setup, we use the *interleaved* allocation order under *high contention*. However, we modify the experiment in a way that all threads use the GRU instruction, except for the first thread on a socket which uses the CPU instruction. This causes the referenced double word to be cached by the respective processor.

In Figure 9 we visualize the measurements for the CPU instruction-only setup (`LOCK ADD`) as well as the GRU instruction mixed with some CPU instructions (`CPU/GRU Mix`). In the experiment shown in Figure 9(a), we execute CPU instructions only on the first thread of socket 1 - 4, which belong to the same IRU. We observe that the mix of CPU and GRU instructions starts with a higher absolute throughput, because the first four threads use the CPU instruction. Nevertheless, adding threads that use the GRU instruction results in a worse scalability, but still reaches the same peak throughput as the GRU instruction-only experiment (cf., Figure 8(a)). If we additionally allow the first thread of the 5th socket (different IRU) to use the CPU instruction instead of the GRU instruction, we obtain the measurements shown in Figure 9(b). As shown, both configurations – CPU-only and instruction mix – behave almost the same leading to the conclusion that the cache coherency protocol over QPI and NUMAlink is the root cause for the lack of scalability and solely using GRU instructions bypasses this bottleneck.

Conclusions. With the help of GRU instructions we are able to speed up *atomic counter increments*, which are used in database systems to obtain timestamps, up to an order of magnitude. Our experiments revealed that GRU instructions scale up almost ideally for realistic medium and low contention scenarios. Moreover, we identified the cache coherency protocol as the main bottleneck CPU instruction-based counter increments.

4.2 Latches

Highly parallel in-memory database systems try to avoid latches as much as possible. However, the usage of locks is still inevitable, because threads need to synchronize the work and their access to shared data structures, for instance, while materializing intermediate results. Hence, we investigate scalability and throughput of latches in this section by comparing CPU instruction-based and GRU instruction-based latch implementations.

Both latch implementations are user space spinlocks. The CPU instruction implementation uses the `test-and-set` to acquire the lock instruction and sets the latch to zero to release it. The GRU implementation uses the *compare-and-swap* operation via the `GRU_GAMER` instruction for lock acquisition and the *clear* operation via the `GRU_GAMIR` instruction for lock release. During the experiments, we measure the number of *successful entries* and the *failed attempts*.

Figure 10(a) and 10(b) show the measurements (latch entries per second) of both implementations for the *interleaved* allocation order and both *high* and *low contention* setups. For both contention setups, we observe a similar behavior. The CPU instruction-based implementation shows a higher throughput for a low number of threads until the break-even point of 20 threads is reached. While the CPU instruction's throughput continuously decreases, the GRU instruction-based implementation is able to scale up until its plateau is quickly reached and starts to slowly decrease afterwards. Nevertheless, starting from the break-even point, the GRU implementation always outperforms the CPU implementation. The chance to successfully acquire the lock (normalized to the number of threads) is 32% for the CPU implementation and 99% for the GRU implementation.

Conclusions. Employing GRU instructions for implementing inevitable latches in database systems significantly increases the number of successful latch entries per second. This performance advantage originates from bypassing the cache coherency efforts in the CPUs, which induces a significant overhead for CPU instruction-based latch implementations as the normalized chance to enter a latch reveals.

5. RELATED WORK

Recent works propose adaptive partitioning strategies to reduce remote memory accesses [2, 3, 5, 6]. The same approach applies for atomic operations, which are distributed or avoided by better parallelization strategies [1, 4]. However, since operators need to exchange data, synchronization and cross-socket data transfers are usually inevitable.

The discussed HARP that hosts the GRUs is a representative of the ASIC family. For instance, Wu et al. proposed a new instruction set specifically designed for database operations implemented in a ASIC called Q100 [12]. The instruction set is very similar to SQL operators and can handle most TPC-H queries showing a performance increase of factor 2-3, while only consuming 15% of the energy drawn by CPU implementations. Such ASICs are usually very focused on a specific set of application. In contrast, the HARP ASIC virtually improves any scenario that heavily relies on moving large amounts of data across the NUMA systems.

A comparable solution for scale-out database setups is RDMA [7] sharing the paradigm of offloading memory transfers between individual nodes of a cluster. The GRU is the pendant for scale-up setups and additionally provides a coherent cache across all nodes.

6. CONCLUSIONS AND FUTURE WORK

In this work, we showed how databases on large NUMA systems can profit from offloading their distant memory loads and utilizing explicit access instructions to increase their effective throughput. We showed how using SGI's `gru_bcopy` operation can bring a 30% performance improvement to full table scans, how the GRU's atomic memory operations can improve transaction sequencing by an order of magnitude and reduce the cost of latches by a factor of eight. Furthermore, we explain what factors have an influence on the effectiveness of these improvements and in which cases they are best used. These results warrant further research into coupling core database operations closer with the memory hardware.

With regards to the bcopy operation, we are looking into using it for more algorithms. More specifically, we are experimenting with a GRU-accelerated join that moves intermediary results between nodes using bcopy. This is relevant for large databases, where intermediary results are sometimes in the hundreds of gigabytes or even terabytes and their move across the NUMA network cannot be avoided.

References

[1] Ryan Johnson et al. "Shore-MT: a Scalable Storage Manager for the Multicore Era". In: *EDBT*. 2009, pp. 24–35.

[2] Thomas Kissinger et al. "ERIS: A NUMA-Aware In-Memory Storage Engine for Analytical Workloads". In: *ADMS @ VLDB*. 2014, pp. 74–85.

[3] Viktor Leis et al. "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age". In: *SIGMOD*. 2014, pp. 743–754.

[4] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. "Cicada: Dependably Fast Multi-Core In-Memory Transactions". In: *SIGMOD*. New York, NY, USA, 2017, pp. 21–35.

[5] Danica Porobic et al. "ATraPos: Adaptive transaction processing on hardware Islands". In: *ICDE*. 2014, pp. 688–699.

[6] Iraklis Psaroudakis et al. "Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores". In: *PVLDB* (2016), pp. 37–48.

[7] Wolf Rödiger et al. "High-Speed Query Processing over High-Speed Networks". In: *PVLDB* 9.4 (2015), pp. 228–239.

[8] Peter Rutten and Matthew Marden. *The Value Proposition of Scale-Up x86 Servers: Cost-Efficient and Powerful Performance for Critical Business Insights.* [Online] http://www.sgi.com/pdfs/4582.pdf. 2016.

[9] SGI. *SGI UV 300H for SAP HANA*. [Online] https://www.sgi.com/pdfs/4554.pdf.

[10] Thomas Willhalm et al. "SIMD-scan: Ultra Fast In-memory Table Scan Using On-chip Vector Processing Units". In: *PVLDB* (Aug. 2009), pp. 385–394.

[11] Thomas Willhalm et al. "Vectorizing Database Column Scans with Complex Predicates." In: *ADMS @ VLDB*. 2013, pp. 1–12.

[12] Lisa Wu et al. "Q100: The architecture and design of a database processing unit". In: *ACM SIGPLAN Notices* (2014).

Accepted at Eighth International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS), in conjunction with VLDB