Micro-architectural Exploration of the Relational Memory Engine (RME) on RISC-V SoC using FireSim

Cole Strickler
University of Kansas
colestrickler@ku.edu

Ju Hyoung Mun
Brandeis University
jmun@brandeis.edu

Connor Sullivan
University of Kansas
connor.sullivan13@ku.edu

Denis Hoornaert
TU Munich
denis.hoornaert@tum.de

Renato Mancuso
Boston University
rmancuso@bu.edu

Manos Athanassoulis
Boston University
mathan@bu.edu

Heechul Yun
University of Kansas
heechul.yun@ku.edu

ABSTRACT

There is no ideal data layout. Even for pure analytical workloads, different queries access a different subset of each relation's columns. This is further exacerbated by Hybrid Transactional/Analytical Processing (HTAP) workloads, which have led to systems converting from row to columnar or hybrid layouts, thereby increasing memory usage and code complexity. The recently proposed Relational Memory Engine (RME) is a hardware accelerator designed to address these challenges by transparently presenting the optimal layout to the CPU. The original RME prototype, built on a PS-PL platform, had limited micro-architectural configurability and a fixed low clock speed, restricting performance analysis and ASIC portability.

In this work, we re-implement RME on a RISC-V system-on-chip (SoC) platform using FireSim to address these limitations by enabling flexible SoC design parameterization and detailed performance evaluation. We simplify and improve the prior RME hardware design and leverage the increased flexibility of our platform to further explore RME's performance characteristics under various micro-architectural settings. We show that hardware prefetching significantly enhances RME performance by effectively masking latency, even for low clock speeds. Out-of-order CPU cores further amplify performance gains, indicating a synergistic relationship between RME and high-performance core designs. We also identify a critical RME clock speed threshold, beyond which performance degradation becomes substantial. Finally, we open-source our design to facilitate further research on TileLink-based RISC-V SoCs.

VLDB Workshop Reference Format:
Cole Strickler, Ju Hyoung Mun, Connor Sullivan, Denis Hoornaert, Renato Mancuso, Manos Athanassoulis, and Heechul Yun. Micro-architectural Exploration of the Relational Memory Engine (RME) on RISC-V SoC using FireSim. VLDB 2025 Workshop: 16th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS25).

1 INTRODUCTION

Data Movement is the Main Bottleneck. Data analytics and other data-intensive applications ship large amounts of data from slow storage and main memory to the CPU. As a result, the main choke point is data movement through channels of limited bandwidth [16]. Prior work has addressed this challenge by proposing data layouts that better match the application's access patterns.

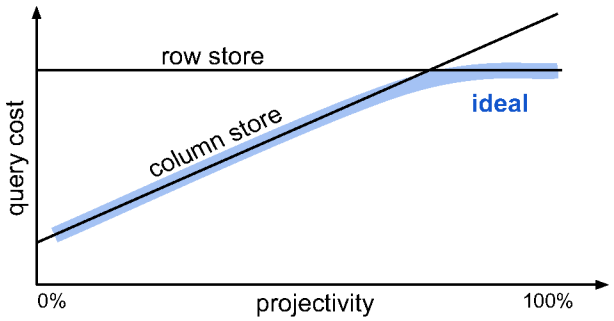


Figure 1: The goal of RME is to capture the ideal cost for any projectivity: match column-wise accesses cost for low projectivity and when tuple reconstruction makes columnar accesses inefficient, match row-wise cost as shown in the original RME work [23].

Hybrid Data Layouts. In addition to the textbook row-wise [22] and column-wise [2] data layouts, recent research on hybrid layouts [3, 7, 14] proposes flexible data layouts that match the access patterns of the queries. Unfortunately, these approaches create the need to maintain and manage multiple data copies. This has been further exacerbated by the advent of Hybrid Transactional/Analytical Processing (HTAP) [27] workloads that have mixed requirements. Many HTAP systems [3, 19, 27] ingest data in a row-wise format, then gradually optimize their layout based on query history, often resulting in a format that is neither a row store nor a column store. However, maintaining multiple copies of data in various layouts increases complexity, which makes these systems less scalable. Furthermore, history-based schemes are significantly impacted by sudden changes in column attributes.

On-the-fly Data Transformation. On the other hand, Relational Memory [23] proposes a hardware-based solution that utilizes commercially available PS-PL platforms [13], which integrate a traditional processing subsystem (PS) with programmable logic (PL). Relational Memory utilizes a single in-memory row store and a hardware engine, termed Relational Memory Engine (RME), in between the CPU and main memory, that transforms data *on-the-fly* into any desired layout, which optimally works for any query.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment. ISSN 2150-8097.

Table 1: Expanding RME evaluation setup.

Setup	Details
RME [23]	In-order core 1.5 GHz w/ prefetcher, RME @ 100MHz
Design 1	In-order core 1.0 GHz (w/ prefetcher, w/o prefetcher), RME @ 1 GHz
Design 2	Out-of-order core 1.0 GHz (w/ prefetcher, w/o prefetcher), RME @ 1 GHz
Design 3	In-order core 1.0 GHz w/ prefetcher, RME @ (1000, 500, 250, 167, 142, 125, 111, 100) MHz

RME successfully accelerates a wide range of database queries, by offering ideal access cost, irrespectively of the query projectivity as qualitatively shown in Figure 1. Previous work implements RME on a specific PS-PL platform to enable full-stack system evaluation. This presents a significant limitation: RME's performance can only be measured on this single platform with fixed architectural settings. Thus, it is impossible to analyze RME's performance under different configurations of system components, such as the CPU, prefetchers, cache, etc. Another limitation is that RME is synthesized to operate at a 100 MHz frequency on an FPGA with a 333 MHz maximum frequency. Thus, analyzing the performance improvement for RME with different—especially higher—working frequencies becomes challenging. Finally, the original design uses FPGA-specific resources such as BRAM, and it was yet to be seen if the design could be ported to an ASIC. Table 1 summarizes the setups we explore.

How does Relational Memory perform with different hardware? In this work, we re-implement RME using Chisel HDL [9] and integrate our design into the Rocket Chip [8] RISC-V system-on-chip (SoC) framework. We make changes to the original RME design to improve upon the previous design and to meet the constraints of the new environment. The Rocket Chip infrastructure allows SoC designs with various micro-architectural parameters to be readily generated, providing an ideal sandbox for testing RME. We use the Xilinx UltraScale+ VCU118 FPGA [1] and FireSim [18] simulation infrastructure to simulate our hardware designs and conduct analysis. Utilizing this setup, we want to answer the following questions: (1) *What is the impact of hardware prefetching on the performance of RME?* (2) *What is the impact of utilizing an out-of-order CPU core with RME?* (3) *What is the impact of RME's clock speed on achieved performance?* We find that prefetching is extremely beneficial and can effectively hide any extra latency incurred by RME. Even when RME is under clocking constraints, prefetching can hide any extra latency until RME is clocked 6× slower than the baseline. Additionally, we find that the out-of-order core's parallel memory requests are not able to hide latency to the same degree as prefetching. Finally, we find that a more powerful out-of-order core with prefetching was able to gain even more from RME than an in-order core with prefetching.

Contributions. In summary, we make the following contributions:

- We present an improved RME design, which is implemented in Chisel HDL and can be integrated into any RISC-V SoC that supports the standard TileLink interconnect.
- We present extensive, realistic performance evaluation results of RME under various micro-architectural settings using FireSim, an FPGA-accelerated cycle-exact full-system simulator which enables in-depth micro-architectural exploration.
- We answer the questions about the impact of hardware prefetching, the effect of using an out-of-order CPU core, and the effect of RME's clock speed—analyses that were difficult or impossible on the original, fixed-configuration PS-PL platform.
- We open-source our updated RME design[1].

Paper Organization. The remainder of this paper is organized as follows: Section 2 provides the necessary background to inform the rest of the paper. Section 3 shows how our design was integrated into the Rocket Chip SoC, including engineering challenges and architectural differences resulting from switching from AXI to TileLink. Section 4 evaluates the performance of RME under various micro-architectural settings. Section 5 briefly discusses avenues for expansion and future research. In Section 6, we conclude the paper.

2 BACKGROUND

In this section, we provide the necessary background on Relational Memory, the FireSim simulator, and the TileLink interconnect.

2.1 Relational Memory

Relational Memory [23] is a software/hardware co-design paradigm for on-the-fly data transformation to optimize the view of memory accessed by arbitrary relational queries. It consists of two parts: *Ephemeral Variables*, a lightweight abstraction to use the reorganized data, and the *Relational Memory Engine* (RME), i.e., a hardware engine located between the cache hierarchy and main memory that transforms the data into the optimal layout without duplicating data in main memory.

Ephemeral Variables allow the CPU to access the reorganized data while offering transparent control over RME. From a software point of view, an ephemeral variable is exactly the same as a regular variable. However, the ephemeral variable is not instantiated in main memory. Instead, it is mapped to a special address that can be recognized by RME, and always points to the reorganized data. The ephemeral variable model enables transparent control of the repacked data, presenting them to the CPU as if they were stored sequentially but regardless of their contiguity in main memory.

RME is the actual hardware engine for data transformation. It sits on the programmable logic (PL) in between the cache hierarchy and main memory. As illustrated in Figure 2, RME has four components: *Trapper*, *Monitor Bypass*, *Requestor*, and *Fetch Unit*. The Trapper captures access to ephemeral variables. When the CPU tries to access an ephemeral variable, the Trapper checks the availability of the requested data through the Monitor Bypass module. The Monitor Bypass has two buffers (a.k.a., scratchpads, SPM), one for metadata and one to hold cache lines of reorganized data. The metadata SPM is used to monitor the completion of each cache line construction. If the requested line is already in the data buffer, the Monitor Bypass sends the data to the CPU via the Trapper. Otherwise, the Monitor Bypass triggers the Requestor. The Requestor orchestrates the access to main memory using knowledge of the query at hand and database geometry, including row size, the number of rows to be projected, and information about the columns to be projected.

[1] https://github.com/CSL-KU/relational-memory-firesim

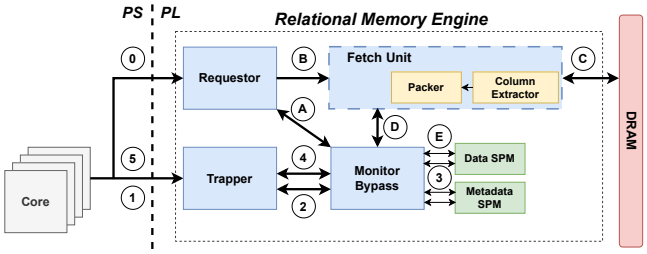


Figure 2: A high-level overview of the original RME design [23] that shows the interaction between the four core modules: Requestor, Trapper, Monitor Bypass, and Fetch Unit.

It issues a set of request descriptors toward the Fetch Unit. Each request descriptor contains addresses to read and where the useful data to extract begins/ends. Finally, the Fetch Unit is the module that reads the data, extracts the necessary parts following the descriptor, packs the extracted data into a cache line, and then sends the constructed line to the Monitor Bypass.

The original prototype of Relational Memory was deployed and evaluated on a commercially available PS-PL platform, specifically the Xilinx Zynq UltraScale+ MPSoC (ZCU102) [13]. This platform incorporates a quad-core ARM Cortex-A53 [4] CPU within its processing system (PS). RME resides in the programmable logic (PL), communicating with the CPU via an interconnect subsystem that employs Advanced eXtensible Interface (AXI) bus elements [5]. RME's experimental evaluation demonstrated that it efficiently mitigates data movement overhead by presenting an optimal data layout to the CPU through on-the-fly data transformation [23].

2.2 FireSim

FireSim [18] is an FPGA-accelerated full-system simulation platform for RISC-V architectures. FireSim offers higher simulation speed than software simulators and offers better accuracy for performance analysis when compared to conventional FPGA prototyping. In FireSim, a simulation is derived from RTL and implemented on an FPGA, with added abstractions to enable a decoupled design. Specifically, one cycle of the FPGA is equal to one or more cycles in the simulated design [21]. This feature of FireSim offers the benefit of hiding and simulating latencies to host components such as DRAM and grants the simulation substantial freedom from the constraints of the physical FPGA platform. The ability to decouple latencies from host DRAM allows FireSim to bridge any clock speed differences between the FPGA and the host and to accurately simulate DRAM access time [11], making it superior to FPGA prototyping for performance analysis of future ASIC implementations.

2.3 TileLink

TileLink [24] is a standard interconnect protocol for on-chip communication on RISC-V SoCs. The protocol allows for coherent access to shared memory resources and peripheral devices.

TileLink has three levels: TL-UL (TileLink Uncached Lightweight), TL-UH (TileLink Uncached Heavyweight), and TL-C (TileLink Cached). TL-C is the most feature-rich, providing support for cache coherence. There are five channels of communication when using TL-C edges: *A, B, C, D*, and *E*. Clients issue Read requests and write-backs over Channel A and C, respectively. Managers provide requested data and acknowledge write-backs on Channel D. Channel B allows managers to issue probes to clients.

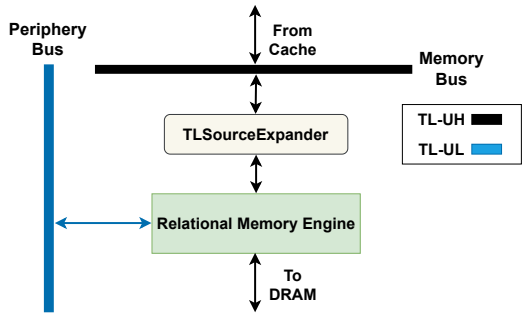


Figure 3: We integrate RME in a RISC-V SoC. We position RME between DRAM and LLC, and connect over TL-UL. This design allows RME to issue requests to DRAM, reorganize the received data, and subsequently transmit it to LLC.

TL-UL and TL-UH, on the other hand, only utilize Channel A and D. Channel A is used for both read and write requests from clients, while Channel D is used for manager responses. Neither of these TileLink specifications supports cache coherency; as such, they are meant for peripheral device and outer memory connections.

3 RELATIONAL MEMORY IN RISC-V SOC

In this section, we describe the design and implementation of RME in RISC-V and FireSim.

3.1 Design Overview

Our RME implementation is designed for integration within RISC-V SoCs that support the standard TileLink interconnect. Figure 3 shows a high-level, logical view of this integration, where RME is positioned between the DRAM and the last level cache and utilizes TL-UH for connectivity. This configuration enables RME to issue requests to the DRAM, reorganize the received data, and subsequently transmit it to the LLC.

Figure 4 shows a more detailed overview of our RME design and its data flow. Note that our RME design is largely the same as that seen in the original work [23] (see Section 2.1 for background), albeit with two key differences: the absence of a large scratchpad memory (SPM) and changes to account for semantic differences between AXI and TileLink. We elaborate on these changes in the next sections.

3.2 Design Changes

Absence of SPM. The scratchpad (SPM) memories of the original design were included to assist with the reorganization of the cache lines. When RME is implemented in an FPGA, the scratchpads can make use of block RAM, but their large size would introduce a significant area overhead that could be unacceptable for an ASIC already wrestling with constrained real estate. With this in mind, we choose to alter the design to do without them. In the absence of scratchpad memories, there is no need to check if we have a hit and bypass sending a request to main memory; hence, we rename

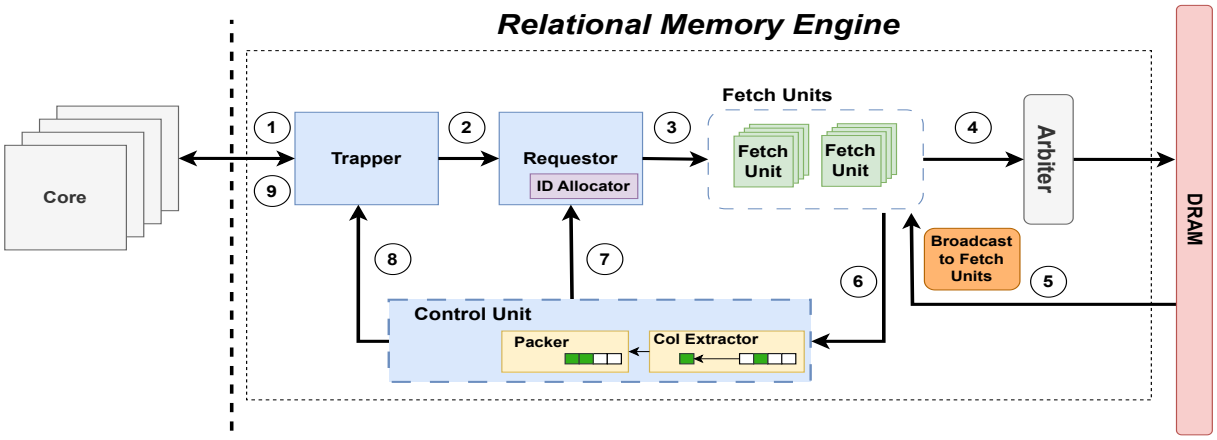


Figure 4: A high-level overview of the amended RME design integrated into RISC-V SoC using FireSim. The fetch unit is split up and each instance handles a single request descriptor. The ID Allocator is introduced to give each transaction a unique source ID and allow out-of-order request handling.

the Monitor Bypass module to *Control Unit*. The removal of these large buffers leaves us with a largely stateless design.

Interconnect Compatibility. The AXI interconnect standard, used by the original design, tags transaction sequences with a unique identifier to differentiate them. The semantics of AXI IDs dictate that transactions with the same transaction ID, belonging to a sequence, must be returned to the sender in the same order in which they were issued [6]. On the other hand, TileLink requires that every outstanding transaction between each source and sink node use a unique source ID and allows these transactions to be completed in any order [24]. These differences required us to alter RME design such that it does not rely on any AXI-specific functionality. The original design relied on this implicit ordering of AXI transactions to designate placement in the reorganized cache line. Additionally, this ordering principle allowed multiple outbound requests to reuse the same transaction ID. We do not get either of these functionalities when using TileLink; therefore, we introduce architectural changes to work around these differences.

To account for the lack of implicit ordering, we alter the design to allow for out-of-order completion of requests. We achieve this by adding extra metadata to the request descriptors generated by the *Requestor* module. This extra metadata acts as a placement marker for where the retrieved data will be placed inside the reorganized cache line. This dynamic placement of data also required changes to the Packer module implementation, and we use a bit-masking mechanism to allow mapping data received out of order to its correct location. To deal with the uniqueness requirement of TileLink, we give each request descriptor a unique source ID. Inside the Requestor module, we introduce a FIFO queue-based ID allocator to allocate these unique source IDs to each request descriptor. These IDs will be placed back in the allocator for reuse once their corresponding requests have made the round trip back from DRAM. RME generates many parallel memory requests, acting as a source with high fan-out, and by default, will quickly exhaust the 2^n source IDs available on the interconnect channel, where n is the number of bits used in the source fields of the TL-A and TL-D channels. To overcome this, we introduce a new Diplomacy [25] node that expands the bit width of the source ID field to allow for a larger number of source IDs to be used within RME and on the outgoing interconnect to DRAM, which is referred to as *TLSourceExpander* in Figure 3.

Multiple Fetch Units. We also alter the setup of the Fetch Unit module by splitting it into multiple replications of itself, where each instance will handle one request descriptor at a time. This setup enables each allocated source ID to uniquely identify the Fetch Unit instance that originated the request. When a reply comes from the memory controller on the TileLink D channel, it is then broadcast to each instance. If the source ID field of the reply matches that of the pending request in a given instance, then the reply is accepted. Splitting the Fetch Unit in this way reduces its complexity, allows for out-of-order request completion, and enables a parameterized design that can match the number of instantiated Fetch Units to the maximum memory-level parallelism of main memory. Lastly, because we split the Fetch Unit, we are forced to move the Column Extractor and Packer modules, which were originally part of the Fetch Unit [23], into the Control Unit, where all fetched data will be funneled to be constructed into a cache line.

3.3 Data Flow

In this subsection, we provide an overview of the data flow in our design, referencing the numbered steps shown in Figure 4. **(1)** A request for data is received from the TL-A channel by the Trapper module. **(2)** This request is passed into an internal queue for processing by the Requestor module. **(3)** The Requestor generates request descriptors that include the physical address of the data, an allocated source ID, where the useful data begins and ends, and a position within the cache line to be constructed. **(4)** The Fetch Units generate a TL-A compatible request, and compete for access to a round-robin arbiter to send those requests to DRAM. **(5)** When a reply is received from DRAM on TL-D, it is broadcast to all fetch units. If the reply's source ID field matches the source ID field of a valid waiting request, then it is accepted. **(6)** Once a request has

Table 2: Evaluation platform specifications

Component	Specifications
In-Order Core	Rocket, 1GHz, L1: 16K(I)/16K(D) (4-Way)
Out-of-Order Core	BOOM, 1GHz, 2-wide, ROB: 64, LSQ: 16/16, L1: 16K(I)/16K(D) (4-Way), 6 MSHRs
Shared L2 Cache	1MB (16-way), 1 Bank
Main Memory	4GB DDR3 1066 MHz, 1 Rank, 8 Banks
Prefetcher	L1 4-Ahead Multi Next Line
RME Parameters	16 Fetch Units
Operating System	Buildroot Linux v6.2.0

completed its round trip from DRAM, the fetch unit then attempts to pass the request to the Control Unit for cache line construction. The relevant data is extracted in the Column Extractor module. A bit masking scheme is used within the Packer to move the extracted data to its location within the cache line. Currently, only one cache line is allowed to be constructed at a time. The necessary logic is in place to only accept data from Fetch Units that originated from the same base request as that currently being processed by the Control Unit. This is an implementation artifact, and this portion of the pipeline could be easily widened; however, our profiling of internal RME latencies showed that this was unnecessary in our setup. **(7)** After a request descriptor has reached the control unit, the allocated source ID is retired back into the ID allocator module inside the Requestor for later reuse. **(8)** When a cache line is fully constructed, it is sent to the Trapper. **(9)** The Trapper constructs a multi-beat reply and sends the constructed cache line back to the request originator on the TL-D channel.

4 EVALUATION

In this section, we perform experiments to investigate how the architectural variations considered in Table 1 affect RME's performance. As such, we focus on the following questions:

(1) *What is the effect of prefetching on RME performance?*
(2) *What is the effect of an out-of-order core on RME performance?*
(3) *What is the effect of clock speed limitation on RME performance?*

4.1 Experimental Setup

We use FireSim, an FPGA-accelerated cycle-exact full system simulator, as our evaluation environment [18]. This allows us to accurately evaluate the performance of the proposed hardware design when deployed in an ASIC. FireSim is capable of simulating a System-on-Chip (SoC) operating at more than 1 GHz while physically running on the host FPGA at a lower clock speed.

Hardware Setup. Using FireSim, we evaluate RME under the following operating environments: in-order core, out-of-order core, in-order core with prefetcher, out-of-order core with prefetcher, and under constrained clocking. Each setup enables a different amount of request-level parallelism, placing varying levels of pressure on both the memory hierarchy and RME. As we vary each micro-architectural aspect of interest [10], we leave the rest of the system setup unchanged. Table 2 shows the basic system setup.

Workload. We use a subset of synthetic benchmarks from [23] that are designed to mimic various database queries. In particular,

Listing 1: Evaluated Database Queries

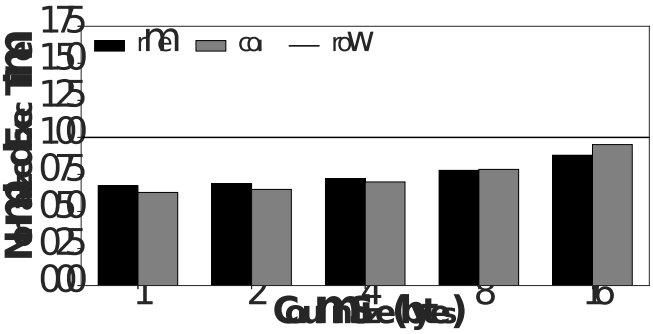
```
Q0: SELECT A1, A2, A3 FROM S;
Q1: SELECT A1, A2, ..., Ak FROM S;
```

Listing 1 reports the two queries that are the focus of our evaluation. Both `Q0` and `Q1` are projections of k columns (either contiguous or non-contiguous). For `Q0`, k is fixed at 3, while we vary the size of each column from 1 to 64 bytes. For `Q1`, on the other hand, we vary k from 1 to 11 while fixing the column size to 4 bytes. Both queries are performed on a 27 MB in-memory database. We note that our RME implementation can scale to much larger database sizes while maintaining the same performance trends. Since we simulate hardware using FireSim, the execution time is amplified, thus, we select a data size that is large enough (larger than our system's LLC) to stress RME and small enough to complete the benchmarks in a timely manner. We set the row size to 64 bytes. We measure and compare the query execution time under three setups: (1) using RME as opposed to employing a traditional (2) row or (3) column store memory layout without on-the-fly data transformation. As such, most of our experiments adopt the same convention: the 'rme' legend entry refers to the first setup, while the entries labeled 'row' and 'col' refer to setups (2) and (3), respectively. In all the figures reporting said comparisons, execution times are normalized to the row store baseline.

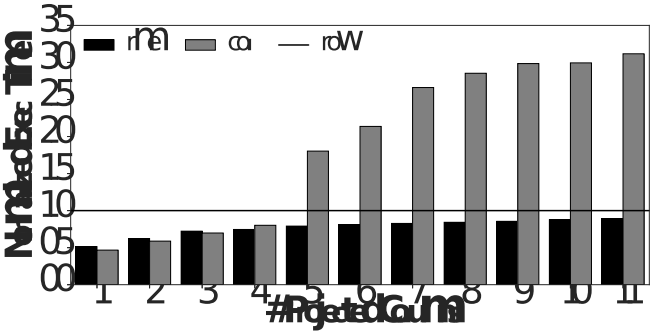
4.2 Effect of Hardware Prefetching

In this experiment, we evaluate the effect of the hardware prefetcher on the performance of RME. Because the ephemeral variable model of RME transforms complex memory access patterns into prefetcher-friendly sequential patterns, prefetchers can significantly improve RME's performance. However, the effect of prefetchers was not quantitatively studied in the original work [23], due to limitations of the hardware platform, which prevented disabling the prefetchers. In this work, we instantiate two system configurations—both utilizing an in-order Rocket Core, similar to the in-order Cortex-A53 core used in [23]—one *with* and one *without* a hardware prefetcher, using FireSim to quantitatively assess the impact of prefetching on the performance of RME.

Figure 5 shows the performance on the Rocket core *with* a hardware prefetcher. First, in Figure 5a, which shows the performance of the `Q0` query under varying column sizes, RME delivers performance comparable to the column-store baseline and even slightly outperforms it when the column size is large (16 bytes). When the column size is small, both RME and the column-store significantly outperform the row-store baseline due to their preferable data layouts that enable more efficient data delivery to the CPU. Second, in Figure 5b, which shows the performance of the `Q1` query with a varying number of enabled columns from 1 to 11, RME shows consistently better performance over the row store baseline, while the column store baseline performs poorly when the number of enabled columns exceeds 4. RME's superior performance over both the row-store and column-store baselines is due to its low column reconstruction cost, as reported in [23]. Note that Figure 5a and 5b correspond to the same evaluation scenarios (on different hardware) as Figures 7 and 8 of the original RME work [23], respectively. *The*

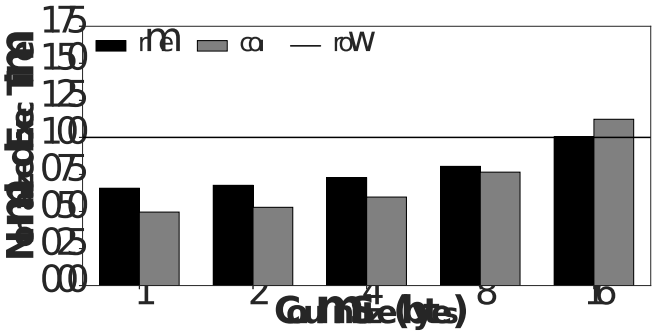


(a) Executing `Q0`: RME is always able to outperform the row-store execution. Further, it is competitive with the column-store execution and even outperforms it for a large column size.

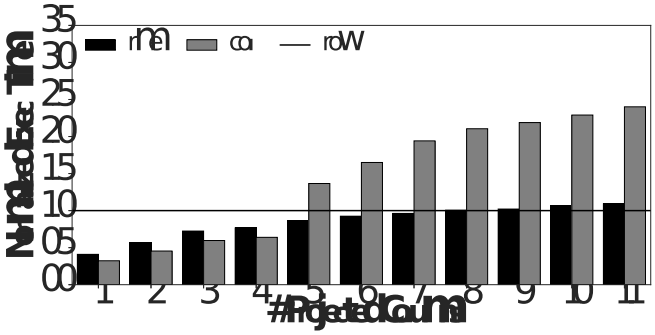


(b) Executing `Q1`: As we increase projectivity, the tuple reconstruction cost kicks in, making RME substantially faster than column-store. The jump between 4 and 5 projected columns is attributed to the prefetching capabilities of the Rocket processor. We note that RME is always faster than row-store, even for high projectivity.

Figure 5: Normalized execution time for `Q0` and `Q1` queries on Rocket (in-order) with hardware prefetching enabled.



(a) Executing `Q0`: RME outperforms row-store for column sizes up to 8 bytes. We also observe that prefetching is more beneficial for RME vs. column-store for low column sizes.



(b) Executing `Q1`: Turning off hardware prefetching affects RME, however, it does not change the high-level trends. For increased projectivity, the tuple reconstruction cost kicks in, making RME substantially faster than column-store. Further, the lack of prefetching makes row-store for high projectivity more competitive than RME.

Figure 6: Normalized execution time for `Q0` and `Q1` queries on Rocket (in-order) without hardware prefetching.

trends we observe here closely match those reported for the original RME design [23], suggesting the validity of our RME implementation as well as the similarity of the CPU's micro-architectural characteristics—both are in-order CPUs with prefetchers.

Figure 6 shows the results of the same experiments on the Rocket core *without* a hardware prefetcher. First, for the `Q0` query, shown in Figure 6a, RME's performance is somewhat worse than that of the column store baseline when the column size is small. This is likely due to RME's overhead, which cannot be hidden without the help of the prefetcher. However, as the column size increases, the gap reduces and eventually reverses, similar to the trend we observe on the CPU with a prefetcher (Figure 5a).

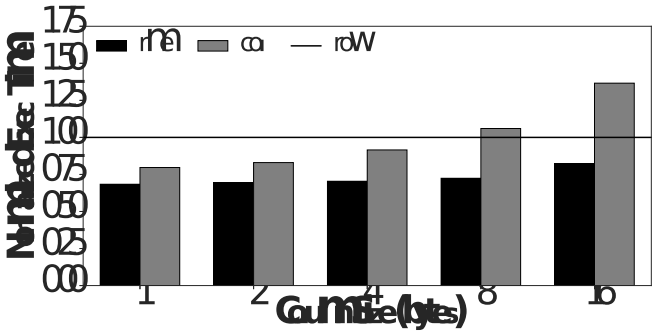
For the `Q1` query, shown in Figure 6b, RME again performs consistently well against the row store baseline, although the performance advantage shrinks and eventually reverses, ever so slightly, as the number of enabled columns increases. On the other hand, the column store baseline exhibits significant performance degradation when the number of enabled columns exceeds 4, again similar to what we observe on the CPU with a prefetcher (Figure 5b).

In summary, we find that **the presence of a data prefetcher consistently enables RME to deliver better performance**. Without a prefetcher, RME's performance is more nuanced compared to a column store, with some performance loss under low projectivity. In all cases, RME remains equivalent to or better than a row store.

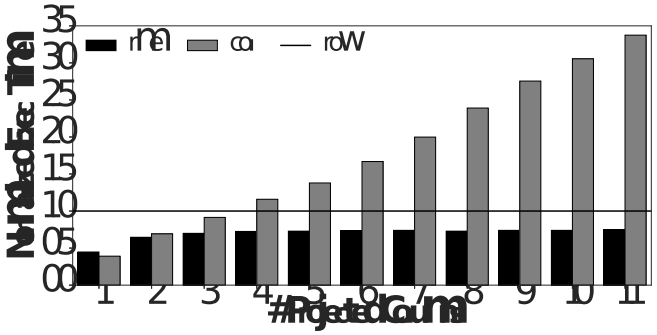
4.3 Effect of Out-of-Order Core

In this experiment, we evaluate the performance of RME using an out-of-order BOOM core [12]. An out-of-order core generally achieves higher instruction throughput than an in-order one because the former can execute many instructions in parallel and in non-sequential order. In the original work on RME [23], however, the evaluation platform was equipped with an in-order CPU and did not include an out-of-order alternative. In this work, using FireSim, we instantiated BOOM-based system configurations to study the effect of using an out-of-order CPU on the performance of RME.

Figures 7 and 8 show the results of the same experiments using the BOOM core with and without a prefetcher, respectively. We initially hypothesized that the BOOM core's ability to extract parallelism might make hardware prefetchers less important for RME. Conversely, the results show that this is not the case, as even with the BOOM core, RME still benefits significantly from hardware prefetchers, similar to the results observed with the Rocket core in the previous subsection. In fact, we find that the prefetcher provides an even higher performance boost for RME on the BOOM

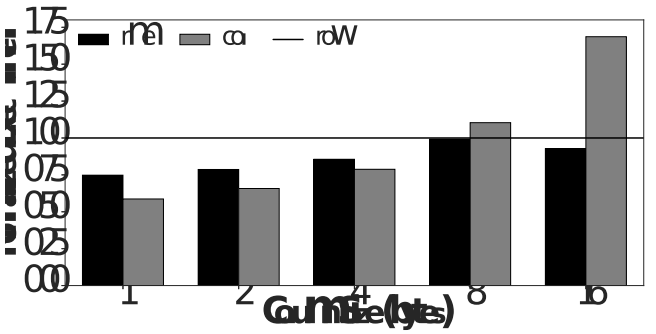


(a) Executing `Q0`: RME is always able to outperform both the row-store and the column-store.

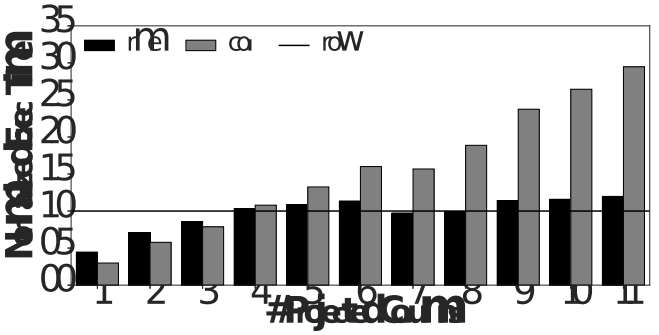


(b) Executing `Q1`: As we increase projectivity, the tuple reconstruction cost kicks in, making RME substantially faster than column-store. RME is always faster than row-store, even for high projectivity.

Figure 7: Normalized execution time for `Q0` and `Q1` on BOOM (out-of-order) with hardware prefetching enabled.



(a) Executing `Q0`: Prefetching is more beneficial for RME vs. row-store the column-store execution.



(b) Executing `Q1`: Turning off hardware prefetching affects RME, however, it does not change the high-level trends. The lack of prefetching makes row-store for high projectivity more competitive than RME.

Figure 8: Normalized execution time for `Q0` and `Q1` on BOOM (out-of-order) without hardware prefetching.

core (out-of-order) than on the Rocket (in-order) compared to the row-store baselines on the respective platforms. For instance, for `Q1` query processing with 11 enabled columns, RME's normalized execution time is 0.75 on the BOOM (see Figure 7b), while it was 0.89 on the Rocket (see Figure 5b). This is because RME can feed data to the CPU in a compact and efficient manner, allowing a high-performance core to extract more parallelism, resulting in higher performance.

In summary, we find that the **RME improves performance regardless of whether the CPU is in-order or out-of-order**, and its performance gain is maximized on a high-performance, out-of-order CPU that also includes a hardware prefetcher.

4.4 Effect of RME Clock Speed

In this experiment, we vary RME's clock speed to understand its impact on the overall performance. Note that the original RME implementation on a PL-PS system required a request to be handled by an FPGA, with a synthesis frequency that was limited to 100 MHz, while the CPU was running at a much higher 1.5 GHz clock speed [23]. In our implementation, shown in Figure 3, the default case (and normalization baseline) corresponds to a setup where RME shares the same clock domain as the memory bus, operating at a frequency of 1 GHz. The FireSim memory bus interfaces with the FASED [11] DRAM controller. It is challenging to directly vary its frequency because DRAM-specific timing parameters would require manual recalibration to account for the frequency change. Instead, we developed a clock-gating mechanism that allows us to run RME at a lower frequency (while the rest of the system still operates at the 1 GHz frequency) to study the impact of RME's clock frequency on the overall performance.

For this experiment, we utilize an in-order rocket core with prefetching enabled as the baseline hardware configuration. A multi-column projection query, `Q1`, is performed for different numbers of enabled columns across different RME clock speeds using the clock gating mechanism. Figure 9 shows the results, which are normalized to the base 1 GHz RME clock speed result. First, note the sharp increase in execution time as the clock speed is decreased below 167 MHz, or 6× slower than the original clock speed. Up to this point, query execution time is largely unaffected by the lower RME clock speed. This is because prefetching is able to hide the increased latency from RME up to this threshold. Beyond that, however, performance can be significantly impacted by the increased latency of RME. As such, in future on-chip designs, integrating RME(s) in a clock domain that is different and anywhere between 1×-6× slower than that of the main CPUs is certainly an option. This also means that the limited clock speed of the FPGA in the PS-PL system used in the original RME study [23] might have limited RME's performance depending on other factors such as memory

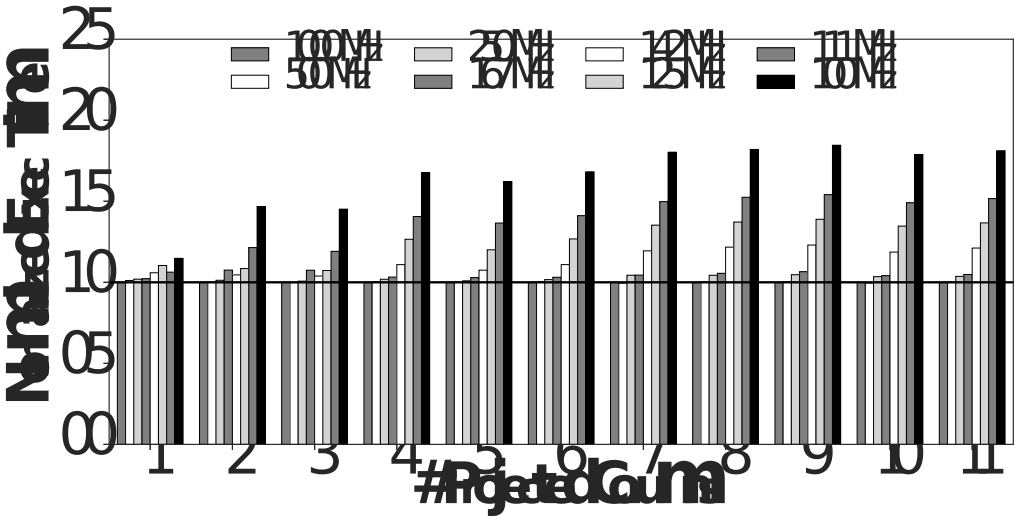


Figure 9: Effect of Clock Speed on RME Performance, normalized to the setup where RME shares the same clock domain as the memory bus, operating at a frequency of 1 GHz. RME's clock can be a performance-limiting factor. After a certain threshold, the overall RME performance is significantly affected.

bus clock speed and prefetcher efficacy. Indeed, in that study, RME was operating at a clock frequency that was 15× slower than the main CPU cluster.

In summary, we find **the clock speed of RME can be a crucial performance-limiting factor**, and a certain threshold exists, after which its impact is significantly pronounced.

5 FUTURE WORK

As discussed in the original RME paper [23], multiple Fetch Units can enable higher memory controller utilization. Since we made such changes in our new design, it is now possible to explore DRAM organization-aware policies that exploit knowledge of DRAM microarchitecture, such as bank-aware scheduling of the Fetch Units. In addition, we would also like to integrate RME with other techniques for managing the memory hierarchy. We observe that many database queries stream large amounts of data to the cache, hence we envision techniques to intelligently coordinate writeback policy at the cache level with RME. Techniques similar to *Eager Writeback* [20] can be used to reduce contention between demand fetches and dirty writebacks to maximize RME bandwidth. There is also room for coordinating with the memory controller to reduce contention at the level of shared DRAM resources. We envision that techniques like *the virtual write queue* [26] can be used to coordinate write-back policies with the memory controller such that they do not conflict with a busy RME and inflate the latency of constructing a cache line. In terms of analysis, a look at multi-threaded workloads has yet to be done. The scalability of RME across multiple parallel threads is something we plan to look into. In the future, we would also like to integrate RME and other data transformation methods more tightly into RISC-V. There is an open question of how best to accomplish this, whether it be through OS control via similar methods to the *deterministic memory abstraction* [15] and *SpectreGuard* [17], or by taking advantage of RISC-V readiness for ISA extensions. We plan to explore both routes and programming models for each in the future.

6 CONCLUSION

In conclusion, in this work we re-implemented RME in Chisel HDL and integrated the design into the RISC-V Rocket Chip SoC ecosystem. We used this new environment along with FireSim, a cycle accurate ASIC simulator, to perform an in-depth analysis of how RME interacts with various components of the system. In our evaluation we observed that RME is able to benefit systems that do not have hardware prefetching, but to a lesser degree than when it is present. We found that prefetching is able to hide nearly all of the latency incurred by RME, even under constrained clocking. We also found that an out-of-order core is able to act synergistically with RME and prefetcher and incur even greater relative speedup.

ACKNOWLEDGMENTS

This research is supported in part by NSF grants CCF-2403012, CCF-2403013, CSR-2238476, IIS-2144547, and a Cisco gift. Denis Hoornaert was supported by the Chair for Cyber-Physical Systems in Production Engineering at TUM and the Alexander von Humboldt Foundation.

REFERENCES

[1] 2024. AMD Virtex UltraScale+ VCU118 FPGA. https://www.FN11-CC30102-PG03-NA-NA-NA-NA.com/products/boards-and-kits/vcu118.html. Accessed: 2024-05-13.

[2] Daniel J Abadi, Peter A Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases* 5, 3 (2013), 197–280. http://dx.doi.org/10.1561/1900000024

[3] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H2O: A Hands-free Adaptive Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1103–1114. http://dl.acm.org/citation.cfm?id=2588555.2610502

[4] ARM. 2015. *Cortex-A53*. Technical Report. ARM. https://developer.arm.com/Processors/Cortex-A53

[5] ARM. 2019. *AMBA AXI and ACE Protocol Specification.* Technical Report. ARM. https://developer.arm.com/documentation/ihi0022/h

[6] ARM. 2025. *Transfer behavior and transaction ordering.* Technical Report. ARM Holdings. https://developer.arm.com/documentation/102202/0300/Transfer-behavior-and-transaction-ordering

[7] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 583–598. https://www.cs.cmu.edu/~jarulraj/papers/2016.tile.sigmod.pdf

[8] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[9] Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the Annual Design Automation Conference (DAC)*. 1216–1225. https://doi.org/10.1145/2228360.2228584

[10] Berkeley Architecture Research preFetchers [n. d.]. Berkeley Architecture Research preFetchers. https://github.com/ucb-bar/bar-fetchers.

[11] David Biancolin, Sagar Karandikar, Donggyu Kim, Jack Koenig, Andrew Waterman, Jonathan Bachrach, and Krste Asanovic. 2019. FASED: FPGA-Accelerated Simulation and Evaluation of DRAM. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2019, Seaside, CA, USA, February 24-26, 2019*. 330–339. https://doi.org/10.1145/3289602.3293894

[12] Christopher Celio, David A. Patterson, and Krste Asanović. 2015. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor.* Technical Report UCB/EECS-2015-167. University of California-Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECSz2015-167.html

[13] Advanced Micro Devices. 2025. *AMD Zynq™ UltraScale+™ MPSoCs.* Technical Report. Advanced Micro Devices. https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-ultrascale-plus-mpsoc.html

[14] Jens Dittrich and Alekh Jindal. 2011. Towards a One Size Fits All Database Architecture. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. 195–198. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper25.pdf

[15] Farzad Farshchi, Prathap Kumar Valsan, Renato Mancuso, and Heechul Yun. 2018. Deterministic Memory Abstraction and Supporting Multicore System Architecture. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*. 1:1–1:25. https://doi.org/10.4230/LIPIcs.ECRTS.2018.1

[16] Manos Frouzakis, Juan Gómez-Luna, Geraldo F. Oliveira, Mohammad Sadrosadati, and Onur Mutlu. 2025. PIMDAL: Mitigating the Memory Bottleneck in Data Analytics using a Real Processing-in-Memory System. *CoRR* abs/2504.01948 (2025). https://doi.org/10.48550/arXiv.2504.01948

[17] Jacob Fustos, Farzad Farshchi, and Heechul Yun. 2019. SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*. 61. https://doi.org/10.1145/3316781.3317914

[18] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy H. Katz, Jonathan Bachrach, and Krste Asanovic. 2018. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 29–42. https://doi.org/10.1109/ISCA.2018.00014

[19] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 195–206. http://dl.acm.org/citation.cfm?id=2004686.2005619

[20] Hsien-Hsin S. Lee, Gary S. Tyson, and Matthew K. Farrens. 2000. Eager writeback - a technique for improving bandwidth utilization. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 11–21. https://doi.org/10.1109/MICRO.2000.898054

[21] Albert Magyar, David Biancolin, John Koenig, Sanjit Seshia, Jonathan Bachrach, and Krste Asanovic. 2019. Golden Gate: Bridging The Resource-Efficiency Gap Between ASICs and FPGA Prototypes. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019*. 1–8. https://doi.org/10.1109/ICCAD45719.2019.8942087

[22] Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database Management Systems.* McGraw-Hill Higher Education, 3rd edition.

[23] Shahin Roozkhosh, Denis Hoornaert, Ju Hyoung Mun, Tarikul Islam Papon, Ahmed Sanaullah, Ulrich Drepper, Renato Mancuso, and Manos Athanassoulis. 2023. Relational Memory: Native In-Memory Accesses on Rows and Columns. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 66–79. https://doi.org/10.48786/edbt.2023.06

[24] SiFive. 2017. SiFive TileLink Specification. Accessed: 2024-05-10.

[25] Henry Cook SiFive. 2017. Diplomatic Design Patterns: A TileLink Case Study. https://api.semanticscholar.org/CorpusID:44937251

[26] Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C. Hunter, and Lizy K. John. 2010. The virtual write queue: coordinating DRAM and last-level cache policies. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 72–82. https://doi.org/10.1145/1815961.1815972

[27] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid Transactional/Analytical Processing: A Survey. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1771–1775. http://doi.acm.org/10.1145/3035918.3054784