# Exploring Architectural-Aware Affinity Policies in Modern HPC Runtimes

Ian Di Dio Lavore
Pacific Northwest National Laboratory, USA
Politecnico di Milano, Italy
ian.didio@polimi.it

Vito Giovanni Castellana
Pacific Northwest National Laboratory, USA
vitogiovanni.castellana@pnnl.gov

John Feo
Pacific Northwest National Laboratory, USA
john.feo@pnnl.gov

Marco Santambrogio
Politecnico di Milano, Italy
marco.santambrogio@polimi.it

## ABSTRACT

Modern commodity and High-Performance Computing (HPC) systems are evolving with complex CPU architectures. These architectures now feature higher core and NUMA domain counts and implement features such as hyperthreading. When considering significant differences in hardware configurations, library availability, and hardware-tailored system/software stacks, which could substantially vary from one system to another, performance portability is hard to achieve. Throughout the years, this trend resulted in an increasingly high burden on application developers to fine-tune their workloads for each architecture. This work explores how hardware-dependent aspects such as locality/process/thread affinity affect performance in modern CPU architectures. We focus our study on the Global Memory and Threading (GMT) distributed runtime system as a representative of Partitioned Global Address Space (PGAS) software stacks commonly adopted for productivity. In particular, to appreciate performance implications, we evaluate GMT's thread affinity policies, and, introduce two new ones which exploit architectural awareness. Finally, we explore alternative NUMA configurations via different process bindings and perform a scalability study on three HPC clusters with varying CPU architectures and NUMA layouts. Our analysis indicates that more complex architectures are more affected by affinity and binding policies and highlights the importance of setting proper runtime configurations to achieve superior performance.

## CCS CONCEPTS

• **Computer systems organization** → *Multicore architectures*; • **Software and its engineering** → *Process management*; *Software performance*; *Distributed programming languages*; *Parallel programming languages*.

## KEYWORDS

Thread Affinity, Process Binding, High-Performance Computing, Distributed Runtime Systems

*PEARC '24, July 21–25, 2024, Providence, RI, USA*
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0419-2/24/07
https://doi.org/10.1145/3626203.3670566

**ACM Reference Format:**
Ian Di Dio Lavore, Vito Giovanni Castellana, John Feo, and Marco Santambrogio. 2024. Exploring Architectural-Aware Affinity Policies in Modern HPC Runtimes. In *Practice and Experience in Advanced Research Computing (PEARC '24), July 21–25, 2024, Providence, RI, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3626203.3670566

## 1 INTRODUCTION AND BACKGROUND

In the never-ending pursuit of achieving higher and higher performance, major manufacturers such as AMD, Intel, and ARM are developing novel, increasingly complex architectures for their CPUs [10, 12–14]. They offer various processor designs on consumer and server-grade products, which can substantially differ in characteristics, even within the same CPU family. For example, they can have different Non-Uniform Memory Access (NUMA) configurations and hyperthreading capabilities, which could even be adjusted by the end-user or system administrator [1]. On the one hand, these hardware design concepts raise the theoretical performance peak obtainable by the hardware; on the other, they highly affect software performance portability and make tuning workloads to the underlying hardware more tedious. These challenges appear even more prominent in High-Performance Computing (HPC) environments, where additional components such as communication networks and accelerators further contribute to the the system's overall complexity. While several aspects jointly affect the workloads' performance, *process binding* and *thread affinity* are among the most impactful when considering alternative CPU designs or configurations.

The vast majority of HPC programming environments are based on the Message Passing Interface (MPI), whose implementations, such as OpenMPI, allow to specify processor affinity through process binding at different granularities (e.g., `core`, `socket`, `numa`) [6]. Each MPI process and its threads are thus *bound* to a specific subset of processing resources. Although resulting in severe performance penalties, it is also possible to oversubscribe the system, with multiple processes sharing hardware resources. Moreover, *affinity/binding* can be set through dedicated options when launching an MPI program. Consequently, the end developer is responsible for figuring out the optimal affinity configuration to increase, for example, the data locality inside its application, avoiding the placement of collaborative threads/processes in distinct NUMA domains, sockets, or even entirely different nodes. Unfortunately, those hand-tuned workloads and configurations developed on a specific CPU architecture won't necessarily perform as expected on a different

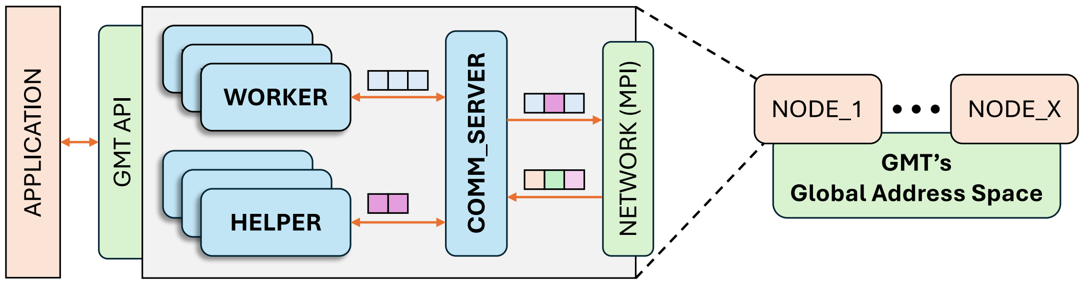

**Figure 1: Bird's eye view on the architecture of the GMT runtime.**

one [14]. This is particularly relevant as it is common practice to develop and validate HPC applications on small(er)-scale systems and later deploy them in large-scale supercomputers. Another aspect that may make applications fine-tuning more challenging is that developers often rely on runtime systems or productivity libraries that, by design, don't provide direct control over threading in favor of high-level APIs. In this paper we discuss the Global Memory and Threading (GMT) runtime system from the Pacific Northwest National Laboratory (PNNL) [11]. GMT offers a Partitioned Global Address Space (PGAS) through global arrays, and, software multi-threading. The library differentiates between `worker`, `helper`, and `comm_server` threads, all of which are internally mapped to actual POSIX threads. A simplified representation of the software architecture is shown in Figure 1. While runtimes like GMT have higher-level APIs than barebone MPI, they are still not an ideal solution for developing complex applications. Several software and libraries have been built to fill this productivity gap, often relying on lower-level runtimes. The Scalable High-Performance Algorithms and Data-structures (SHAD) C++ library offers APIs and semantics analogous to the C++ Standard Template Library (STL) and uses GMT as the backend [4]. SHAD's containers and methods are purposely designed to scale in size and performance on distributed HPC environments.

In this work, we study how *process binding* and *thread affinity* affect the scalability and performance of applications running on top of GMT as a representative of distributed C++ HPC runtimes. To explore multiple configurations, we have introduced additional workload-agnostic affinity policies in the runtime. We conduct our experimental campaign on HPC clusters with different processors, spanning multiple CPU architecture generations. Our analysis emphasizes the urge for HPC runtimes and libraries to exploit architectural awareness and, possibly, to provide the end-users with utilities to tune affinity *at runtime* without the need to modify their codes. Our study indeed indicates that a workload and architecture-agnostic solution would fail to provide optimal performance out of the box, further highlighting the need for user-level, runtime customization options.

## 2 ENABLING ARCHITECTURAL AWARENESS IN THE GMT RUNTIME

While offering its own API and PGAS, GMT ultimately maps on MPI; thus, an application written with the runtime is indeed an MPI program. As such, GMT executables can be launched with different MPI affinity policies, e.g., binding processes to `board` (entire node), `socket`, or `NUMA domains`. Setting a specific binding for the processes restricts its available Processing Units (PUs) and, most importantly, changes the locality profiles of containers, such as GMT's global arrays or SHAD's STL data structures. Consider, for example, the case of a CPU with multiple NUMA domains and processes bound to `board`. Memory allocated on a specific NUMA domain is still accessible by all the threads within the process, potentially resulting in crossing NUMA boundaries, with significant overall performance penalties. On the other hand, increasing the number of processes at will brings diminishing returns because of the cost of inter-process communication and, in the case of over-subscription, sharing PUs across processes. While we can explore different bindings directly through MPI's options, evaluating the effects of thread affinity requires working directly inside GMT. GMT currently offers two affinity options for `worker`, `helper`, and `comm_server` threads, ignoring hardware topology information. We remark that, at runtime, each process has its independent set of specialized threads. The overall number of threads is set to the number of the process' PUs, although the user can override this by oversubscribing the system. Setting an affinity corresponds to defining, at runtime, the thread *affinity mask*, which indicates to the OS scheduler the set of cores where a thread is allowed to run. GMT's `default` affinity policy sets the mask to be the same as the thread parent process. Alternatively, users can pin each thread to a single, specific core using the `pin` policy. However, the legacy implementation has severe limitations since it assumes a linear enumeration of core IDs and a single PU per core. While these assumptions held for older CPU architectures, they do not for more modern designs, which commonly adopt different enumeration schemes and include multiple PUs per core, e.g., when hyper-threading or Simultaneous Multithreading (SMT)

| | DevCluster | Deception | Perlmutter |
|---|---|---|---|
| **CPU** | Intel E5-2670 | AMD 7502 | AMD 7763 |
| **Sockets** | 2 | 2 | 2 |
| **NUMA/Socket** | 1 | 1 | 4 |
| **Core/NUMA** | 10 | 32 | 16 |
| **PU/Core** | 1 | 1 | 2 |
| **Memory** | 768 GB | 256 GB | 512 GB |
| **Network** | Mlnx MT27500 | Mlnx HDR100 | HPE Slingshot11 |

**Table 1: Systems Specifications.**

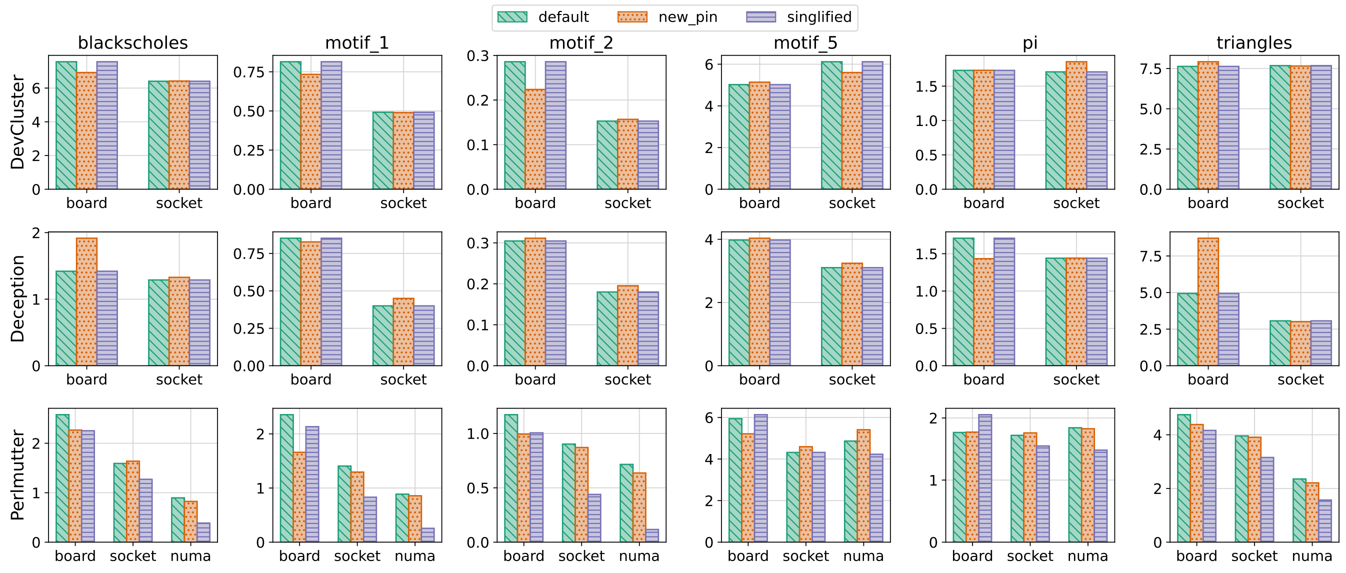

**Figure 2: Execution time (in seconds) on 8 nodes for each system while varying process bindings and affinity policies.**

is enabled. To this end, we introduce two additional affinity policies in GMT. Our implementations use the Portable Hardware Locality (hwloc) library APIs to capture hardware characteristics, including topology [3]. The first one, `singlified`, differs from `default` by filtering out from the processes' affinity masks additional *virtual* PUs within the same core hierarchy (e.g., in hyper-threaded cores). This avoids frequent context switching, which might ultimately hinder overall performance [5]. Finally, `new_pin` restricts threads to a single, physical core exploiting topology information. Our current implementation assigns the `comm_server` to the core closest to the network interface card, when applicable, and pins the rest of the threads so that helpers and workers are grouped into disjoint sets of physically adjacent cores.

## 3 EXPERIMENTAL STUDY

We evaluate the effects of process binding and thread affinity policies on GMT's applications performance on three supercomputers with different CPU architectures employed respectively in an HPC development cluster (*DevCluster*), PNNL's institutional cluster *Deception* and NERSC's *Perlmutter*. Table 1 details the specifics of each system. *DevCluster* and *Deception* are equipped respectively with Intel and AMD-based processors with 2 sockets and one NUMA domain per Socket. *DevCluster*'s CPU architecture features 10 cores per socket, while *Deception* has 32 cores per socket. *Perlmutter*'s compute nodes are instead equipped with two, more recent, AMD Milan CPUs with SMT enabled resulting in 2 PUs per physical core. *Perlmutter*'s CPU configuration is much more complex, with the design laid out in 4 NUMA domains per socket, for a total of 8 NUMA domains per node, compared to the 2 NUMA domains in *DevCluster* and *Deception*. Finally, while *DevCluster* and *Deception* feature Infiniband, *Perlmutter* features a Slingshot interconnect. We use GMT v1.1 with OpenMPI in all experiments. We study the performance of six workloads implemented using the SHAD C++ library for increased productivity. `Black-Scholes` [8] is a financial model used to predict the price of stock options. The application's input is a synthetic dataset with 6 billion option values. `PI` computes an high-precision value of *pi* with a 10 billion points simulation. `triangle-counting` [2] is a structural pattern-matching algorithm that enumerates triangles in a graph. Here, we consider a synthetic R-MAT graph with 8 million vertices and 134 million edges. Finally, we implement three temporal graph homomorphism kernels from [9] on the *ask-ubuntu* [7] dataset (159K vertices and 964K temporal edges). In particular, we selected the two motifs with a simpler structure (`motif_1` and `motif_2`), and the most time-consuming one to match (`motif_5`). We select these application because of their diverse memory, communication and load balance profiles. `Black-Scholes`, `PI` and `triangle-counting` all use global arrays which are evenly distributed. However, while the first two are mostly embarrassingly parallel, with limited communication, the latter is affected by poor load balance. `motif_1`, `motif_2` and `motif_5` instead, store graphs in sparse hash-based data structures, and suffer from both load unbalance and high network traffic. For each configuration, we report the average execution time from 13 consecutive runs, accounting for the first three as warm-up. We explore MPI process bindings to `board`, `socket`, and `NUMA domain`. For the latter, we report results only on *Perlmutter*, since in *DevCluster* and *Deception*, which only have one NUMA domain per socket, `socket` and `NUMA` bindings coincide. Figure 2 plots execution times when running the applications on 8 nodes of each system. In these experiments, we don't report execution times with GMT's legacy pinning strategy (`pin`) because it generates, on the newer generation AMD Milan CPUs, a sub-optimal thread assignment with heavily degraded performance. At first glance, we observe that in older or simpler CPU architectures, different thread affinity policies (i.e., `default`, `singlified`, and `new_pin`) only marginally affect performance. However, while in most cases

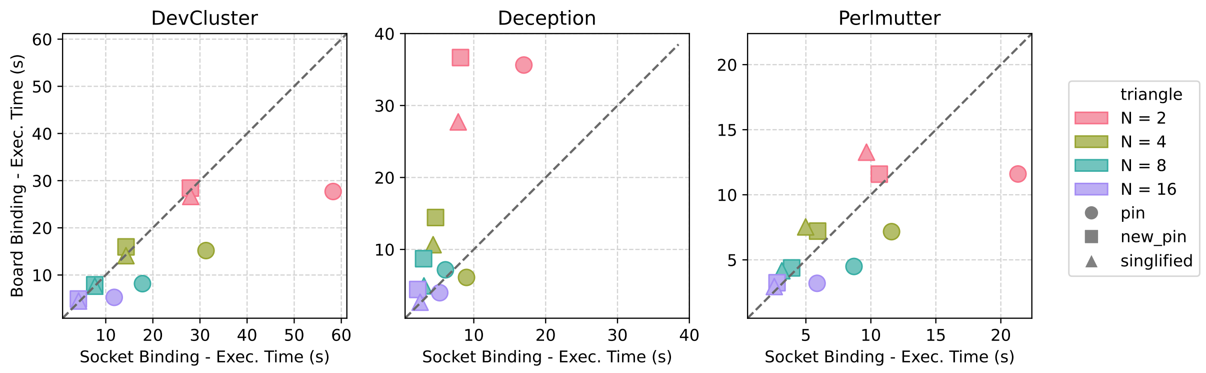

**Figure 3: Scaling impact of board vs. socket bindings under different thread affinity policies for the `triangle-counting` workload.**

the performance difference is negligible (≤ 1%), we still identify outliers in this behavior, i.e. *motif_2* which exhibits a significant 22% slowdown when preferring `singlified` or `default` to `new_pin` on *DevCluster*. On *Deception* instead, we observe that for the majority of the workloads (4 out of 6), the worst performing affinity is `new_pin`, which in turn provided, on average, a marginal performance improvement on *DevCluster*. This is particularly apparent with `black-scholes` and `motif_2`. On more complex hardware, as in *Perlmutter*, affinity impacts performance even more. First, we notice that each workload's performance significantly varies with each affinity, with a much more pronounced difference on several configurations. Nevertheless, we highlight that on this machine we were able to identify a configuration that always outperforms the others, that is, `singlified` affinity with processes bound to `numa`, with speedups up to 4.88*x* for `motif_2` with respect to the `default` policy. As for the thread affinity, we highlight that the impact of process binding on performance is more notable on more modern architectures, with *Deception* and *Perlmutter* typically perfoming better with finer grained bindings, i.e. to `socket`/`NUMA`. This configuration brings significant performance improvements on *DevCluster* only for `motif_1` and `motif_2`.

Finally, we evaluate the scaling properties of `triangle-counting` on all the systems, varying both the nodes counts (from 2 to 16) and binding/affinity configurations. Figure 3 shows experimental results with the `new_pin`, `singlified` and the legacy GMT pinning policy (`pin`), and, process bindings to `socket` (x axis) and `board` (y axis). We remark that `pin`'s implementation is based on assumptions on the system software (Section 2), which don't hold in the configurations under test. In particular, with binding to `socket`, this affinity results in multiple threads to be accidentally bound to the same core, oversubscribing the hardware. For this reason, the strategy consistently results in degraded performance. Our analysis highlights two major considerations. Firstly, the data points have different distributions across different CPUs. This reinforces that developers must incorporate architecture awareness in their solutions to exploit the underlying system effectively and ensure performance portability across different architectures. Secondly, the workload exhibits a broader range of behaviors as the CPU architecture becomes increasingly complex.

## 4 CONCLUSIONS

Modern processor architectures feature novel NUMA domain configurations, hyper-threaded cores and increasing core counts. In this work, we highlight the relevance of architectural-awareness considerations in modern software stacks, especially in distributed HPC runtime systems. We take GMT as a representative of those systems and dissect its thread *affinity* strategies, introducing two new architectural-aware policies, namely `new_pin` and `singlified`. We evaluate those policies with different GMT/MPI process bindings on three HPC clusters with different CPU architectures. Our findings showcase the inherently multifaced nature of this problem. Indeed, on one side, the affinity policies and the various process bindings have less effect on simpler CPU architectures; on the other, as the topology complexity of the CPUs increases, we observed a variegated range of runtime behaviors. For these systems, depending on the workload, selecting a proper process binding and thread affinity is crucial to achieve superior performance. Therefore, while developing modern HPC runtimes, it becomes essential to incorporate those aspects into their inner designs to improve the utilization of the underlining architecture and provide effective performance portability across different systems. Our experimental study highlights that for only one of the tested systems a configuration that always outperforms the others exists. More in general, the effects of the various affinity policies are not uniform across the systems and the workloads. Therefore, we advocate for providing the user the ability to select the policy that best matches its workload, even if theoretically ideal configurations for specific machines might exist.

## ACKNOWLEDGMENTS

This research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), through the Advanced Graphic Intelligence Logical Computing Environment (AGILE) research program, contract number DE-AC05-76RL01830. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, or the U.S. Government.

## REFERENCES

[1] AMD. 2024. MI200 high-performance computing and tuning guide. https://rocmdocs.amd.com/en/latest/how-to/tuning-guides/mi200.html.

[2] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. 2010. Efficient algorithms for large-scale local triangle counting. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 4, 3 (2010), 1–28.

[3] Francois Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. 2010. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE, 180–186. https://doi.org/10.1109/PDP.2010.67

[4] Vito Giovanni Castellana and Marco Minutoli. 2018. Shad: The scalable high-performance algorithms and data-structures library. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 442–451.

[5] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. 2021. A case against (most) context switches. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 17–25. https://doi.org/10.1145/3458336.3465274

[6] Joshua Hursey and Jeffrey M. Squyres. 2013. Advancing application process affinity experimentation: open MPI's LAMA-based affinity interface. In *Proceedings of the 20th European MPI Users' Group Meeting* (Madrid, Spain) *(EuroMPI '13)*. Association for Computing Machinery, New York, NY, USA, 163–168. https://doi.org/10.1145/2488551.2488603

[7] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[8] James D MacBeth and Larry J Merville. 1979. An empirical examination of the Black-Scholes call option pricing model. *The journal of finance* 34, 5 (1979), 1173–1186.

[9] Patrick Mackey, Katherine Porterfield, Erin Fitzhenry, Sutanay Choudhury, and George Chin. 2018. A chronological edge-driven approach to temporal subgraph isomorphism. In *2018 IEEE international conference on big data (big data)*. IEEE, 3972–3979.

[10] Satoshi Matsuoka. 2021. Fugaku and A64FX: the first exascale supercomputer and its innovative arm CPU. In *2021 Symposium on VLSI Circuits*. IEEE, 1–3.

[11] Alessandro Morari, Antonino Tumeo, Daniel Chavarría-Miranda, Oreste Villa, and Mateo Valero. 2014. Scaling irregular applications through data aggregation and software multithreading. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 1126–1135.

[12] Samuel Naffziger, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H Loh, Mahesh Subramony, and Sean White. 2021. Pioneering chiplet technology and design for the amd epyc™ and ryzen™ processor families: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 57–70.

[13] Nevine Nassif, Ashley O Munch, Carleton L Molnar, Gerald Pasdast, Sitaraman V Lyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, et al. 2022. Sapphire rapids: The next-generation intel xeon scalable processor. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. IEEE, 44–46.

[14] Markus Velten, Robert Schöne, Thomas Ilsche, and Daniel Hackenberg. 2022. Memory Performance of AMD EPYC Rome and Intel Cascade Lake SP Server Processors. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering* (Beijing, China) *(ICPE '22)*. Association for Computing Machinery, New York, NY, USA, 165–175. https://doi.org/10.1145/3489525.3511689