Performance Scaling of Cassandra on High-Thread Count Servers

Industry/Experience Paper

Disha Talreja
Advanced Micro Devices, Inc.
dishaa.talreja@amd.com

Kanishka Lahiri
Advanced Micro Devices, Inc.
kanishka.lahiri@amd.com

Subramaniam Kalambur
Dept of Computer Science, PES University
subramaniamkv@pes.edu

Prakash Raghavendra
Advanced Micro Devices, Inc.
prakash.raghavendra@amd.com

ABSTRACT

NoSQL databases are commonly used today in cloud deployments due to their ability to "scale-out" and effectively use distributed computing resources in a data center. At the same time, cloud servers are also witnessing rapid growth in CPU core counts, memory bandwidth, and memory capacity. Hence, apart from scaling out effectively, it's important to consider how such workloads "scale-up" within a single system, so that they can make the best use of available resources.

In this paper, we describe our experiences studying the performance scaling characteristics of Cassandra, a popular open-source, column-oriented database, on a single high-thread count dual socket server. We demonstrate that using commonly used benchmarking practices, Cassandra does not scale well on such systems. Next, we show how by taking into account specific knowledge of the underlying topology of the server architecture, we can achieve substantial improvements in performance scalability. We report on how, during the course of our work, we uncovered an area for performance improvement in the official open-source implementation of the Java platform with respect to NUMA awareness. We show how optimizing this resulted in 27% throughput gain for Cassandra under studied configurations.

As a result of these optimizations, using standard workload generators, we obtained up to 1.44x and 2.55x improvements in Cassandra throughput over baseline single and dual-socket performance measurements respectively. On wider testing across a variety of workloads, we achieved excellent performance scaling, averaging 98% efficiency within a socket and 90% efficiency at the system-level.

KEYWORDS

Performance benchmarking; Cassandra; NoSQL databases; Performance scalability

ACM Reference Format:
Disha Talreja, Kanishka Lahiri, Subramaniam Kalambur, and Prakash Raghavendra. 2019. Performance Scaling of Cassandra on High-Thread Count Servers: Industry/Experience Paper. In *Tenth ACM/SPEC International Conference on Performance Engineering (ICPE '19), April 7–11, 2019, Mumbai, India.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3297663.3309668

1 INTRODUCTION

In recent years the growth of data and the need for analytics has led to the development and deployment of a variety of distributed platforms for computing and storage. Notably, Apache Hadoop based on the Map/Reduce paradigm [7] and Spark [20] have provided fault-tolerant distributed computing platforms for large data sets. NoSQL databases such as HBase based on the BigTable model [4] and Cassandra [13] have emerged as alternatives to relational databases, providing storage and retrieval mechanisms for analytics applications while using distributed resources. The design of such modern platforms are governed by the need to scale-out across a data center to satisfy fault tolerance, scalability and performance requirements.

Recent years have also seen rapid growth in the number of CPU cores within a single server processor, coupled with commensurate increases in memory capacity and memory bandwidth. This has resulted in the ability to support over a hundred hardware threads (or logical CPUs) within commodity dual-socket servers [14]. In order to make effective use of data center resources, distributed computing and storage platforms need to achieve good scaling performance not only *across*, but also *within* such systems. Therefore, the focus of this paper is in studying the ability of Cassandra, a popular NoSQL database designed for scale-*out*, to scale-*up* within a high-thread count server.

1.1 Paper Contributions and Overview

In this paper, we provide an in-depth analysis of the performance of Cassandra on a modern, high-thread count dual-socket server. We first demonstrate that in spite of an abundance of hardware resources, using default configurations of Cassandra and the YCSB benchmarking tool, the database shows poor performance scaling against thread count. We then analyze what can limit this scalability and show that it is largely due to the software stack being unaware of the underlying NUMA topology of the server [8], and bottlenecks around global locks. We next show that through simple reconfigurations of how Cassandra is deployed, its performance scaling properties can be significantly improved. We show how it is possible to achieve up to 98% scaling efficiency against thread count, which can enable Cassandra to exploit the performance capabilities of the underlying hardware to the fullest.

While performing these studies, we discovered an opportunity to improve the performance of OpenJDK [16], the open-source, official reference implementation of Java Platform Standard Edition (JavaSE) [10]. This paper describes our optimization, and the impact it had on Cassandra. Note that, the impact of this contribution is broader, and has the potential for improving the performance of *any* Java-based workload running on *any* NUMA-based system.

Our results on Cassandra highlight the need to optimize deployments of such applications on processors with high thread counts with due consideration of their NUMA topology, and the importance of optimizing software stack components on which it depends (OpenJDK in our case) to be sufficiently NUMA aware. Such optimizations can help enable distributed schedulers to use such resources in the cloud more efficiently and also ensure that database instances are configured appropriately so as to make the most of available thread count and memory capacity.

The rest of this paper is structured as follows. Section 2 discusses related work. Section 3 describes our experimental setup and the workloads we used to drive Cassandra. Section 4 describes the performance scalability of a baseline configuration and the potential upside from optimization. Section 5 describes the various bottlenecks that we encountered, and Section 6 describes the optimizations made to improve performance scalability. We present our final performance measurements in Section 7 and conclude in Section 8 with recommendations for Cassandra deployments on such servers.

2 RELATED WORK

Recent work has described the risks of prioritizing scale-out properties of distributed systems without paying attention to how well individual computers in the system are utilized [15]. Our work shares the same philosophy in that we focus on the performance scaling properties of Cassandra within a single server. Similar work in the context of Map-Reduce applications [2] running on large thread count systems demonstrated that via network stack optimizations, Hadoop could be made to scale linearly with threads. The Sparkle layer [12] provided similar functionality by optimizing network operations on Apache Spark and demonstrated that it is possible to use the multiple cores for processing data efficiently.

Swaminathan and Elmasri [18] compared the performance of MongoDB, HBase and Cassandra over varying workloads sizes to determine their scalability on a cluster of 14 servers using the Yahoo Cloud Serving Benchmark (YCSB) [5]. However, in their study, each server CPU supported only 4 cores, hence they did not encounter major challenges in scaling up. In fact, the scalability of NoSQL databases on large thread count processors remains largely unstudied by the research community.

Newer processors with very high thread counts are often organized using a NUMA approach, where cores are grouped together into NUMA domains, and memory access time depends on the proximity of the memory to the core that is trying to access it. Such an underlying organization should influence both task scheduling and data placement especially in the case of workloads such as in-memory databases. There are several published examples of the benefits of such optimizations, including a study of how optimizing data partitioning and placement in a column-oriented database can improve performance [17], and how NUMA-aware tuning improved mail and web server performance by 10-20% [19].

Table 1: Experimental setup

Hardware	Server	CPU	AMD EPYC^{TM} 7601
		Frequency	2.5GHz
		No of sockets	2
		Cores Per Socket	32
		Threads per socket	64
		Memory	512GB (32 x 16GB-DDR2400)
	Client	CPU	AMD EPYC^{TM} 7601
		Frequency	2.2GHz
		No of sockets	2
		Cores Per Socket	32
		Threads per socket	64
		Memory	512GB (32 x 16GB-DDR2400)
	NIC		Intel 10-Gigabit X540-AT2
Software	OS		RHEL 7.5; Kernel 4.18
	JVM		OpenJDK/Oracle JDK 8u131
	Cassandra		Apache Cassandra 3.10
	YCSB		0.12

Table 2: YCSB workloads tested

Workload	Description
A	Update heavy workload which has a mix of 50/50 reads and writes
B	Mix of 95/5 reads and writes. Read mostly workload
C	Read-only workload
D	Several new records are inserted, where the most recently inserted records are popular. Read-latest workload
F	A client reads a record, modifies it and writes back the changes.

Previous studies on Cassandra [1] have demonstrated that it scales well with threads. However, unlike us, they don't investigate of bottlenecks in the software stack that limit performance. More importantly, the threads counts they consider are much lower that what are available in high-end modern servers.

3 EXPERIMENTAL SETUP AND BACKGROUND

All our measurements were performed using a pair of dual-socket AMD EPYC^{TM} 7601-based systems. Details of these systems are provided in Figure 1. In our setup, one system runs the Cassandra server (or servers), and the other runs a variable number of YCSB clients [5], a program commonly used for benchmarking NoSQL

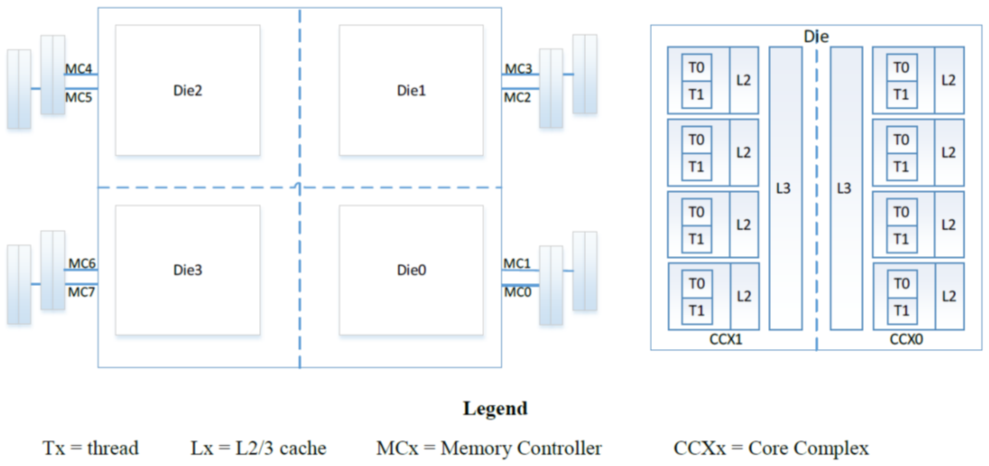

Figure 1: NUMA topology for AMD EPYC^{TM} 7601 CPU

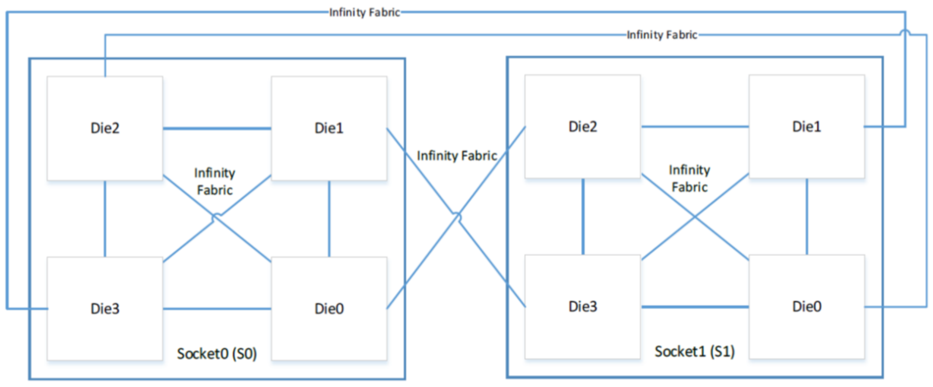

Figure 2: Dual socket topology of AMD EPYC^{TM} 7601 based server

databases. In our studies we tuned the number of client instances to obtain high server utilization.

Since we were concerned with studying scaling behavior, we had to vary the number of active CPU cores in the server system. This would normally lead to higher CPU frequencies at lower core counts (due to increased availability of power headroom). Since the purpose of our work was to analyze performance scaling purely with respect to CPU resources, we disabled dynamic frequency features completely, and ran both client and server systems at fixed CPU frequencies as shown in Figure 1. Our machines communicate over 10 GigE Ethernet and use high-performance NVMe-based SSDs for storage.

Figure 2 lists the YCSB workloads that we used for our analysis. These are default workloads that ship with YCSB, and are widely used by popular benchmarking systems [9]. For our work, all the workloads were configured to generate requests using a zipfian distribution with replication factor set to 1 using default settings for the database itself.

3.1 Terminology

We briefly introduce the NUMA topology for EPYC^{TM} here. As shown in Figure 1, a single AMD EPYC^{TM} 7601 package consists of 4 dies, each die consisting of a pair of core complexes (CCX0 and CCX1). Each complex consists of 4 cores sharing an L3 cache. Each core supports two hardware threads (2-way SMT). A single die has 2 memory channels. Hence, each package supports 32 cores (or 64 hardware threads) and 8 channels of memory. The default NUMA configuration (which we use for our studies) is called *Channel Interleaving*, where each die is configured as a NUMA node, and memory accesses are inter-leaved across the two memory channels on the die. Hence a dual-socket system such as the one used for our studies features a total of 8 dies (or NUMA nodes), as shown in Figure 2. Further details of the NUMA topology for EPYC^{TM} can be found in [14].

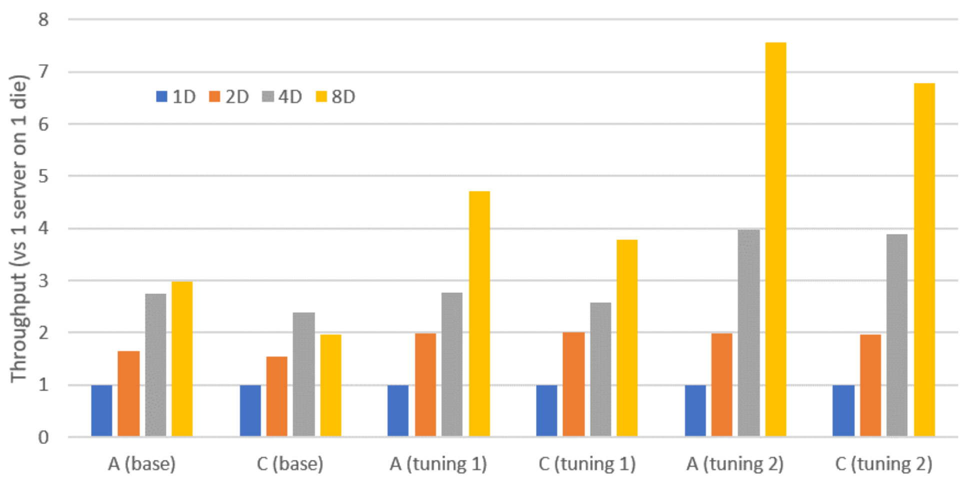

Figure 3: Performance scaling of Cassandra on baseline and tuned configurations for YCSB workloads A and C

4 CASSANDRA PERFORMANCE SCALING

In this section, we present performance scaling data for Cassandra, as measured on our test system, while driving it with a set of YCSB clients with the set up described in the previous section.

We compare the performance scaling of three configurations of the workload. The first configuration, which we use as a baseline, uses workload defaults for various components of the software stack (Cassandra, Java, Linux^{TM} *etc.*). We compare this with the scaling behavior obtained on *tuned* configurations of the workload designed keeping the hardware topology of the server in mind. Details of these techniques and hurdles encountered in doing so are presented in the next section. Figure 3 illustrates how total throughput scales versus increasing hardware threads (expressed in terms of die count) for YCSB workloads A and C (we omit the others for brevity). The throughput is reported in terms of speedups over the throughput obtained when running a single Cassandra server on one die or NUMA node. For each workload, the figure presents performance scaling for the following configurations:

- *Base:* This is an untuned configuration that relies on default values throughout the SW stack. A single Cassandra server is mapped to an increasing number of NUMA nodes.
- *Tuning-1:* In this configuration, except in the 1 die case, we run 2 Cassandra servers on the machine. So '2D' implies each server runs on its own die, in separate sockets. '4D' implies each server gets 2 dies and the pair of dies each server runs on belongs to the same socket. '8D' implies each server gets one entire socket.
- *Tuning-2:* In this configuration, we scale the number of servers along with the available number of dies. So 1D is one server on a single die, 2D is two servers on two dies within a socket, and so on.

It is quite clear from the figure that the tuned configurations can achieve higher scalability than the baseline. The most promising scaling results are obtained with Tuning-2. For example, workload A gains 7.6x (out of a theoretical 8x) from scaling up from 1 to 8 dies, which represents a scaling efficiency of 94%. However, for reasons discussed later in the paper, there may be other factors to consider when selecting a particular configuration. Even so, Tuning-1, a more moderate approach, still achieves 4.7x scaling (vs theoretical 8x). Both of these are far superior to the baseline. For example, for workload A, Tuning-2 achieves 1.44x and 2.55x speed up over the baseline for single socket (4 dies), and dual socket (8 dies).

In summary these findings show that the baseline configuration does not scale well on modern servers offering very large thread counts. However by configuring the *entire software stack* in a manner that is aware of the underlying NUMA topology of the server hardware, it is possible to achieve significant improvements in scaling and throughput performance.

5 BOTTLENECK ANALYSIS

In this section we identify some of the possible reasons behind the lack of good performance scaling in the baseline deployment of Cassandra.

5.1 Hardware Resource Utilization

The first step in this type of analysis is to determine whether limitations on available hardware resources explain the lack of performance scaling. Figure 4 illustrates how the out-of-the-box deployment utilizes primary hardware resources for the YCSB workloads under single and dual socket configurations. Through most of the rest of this paper, we present results for two of the YCSB workloads, A (50% Reads and 50% Updates) and C (100% reads), for reasons of brevity.

The results show that the lack of performance scaling cannot be explained by lack of hardware resources. In the dual-socket case, average CPU utilization is actually *lower* by 23 percentage points compared to the single socket case. Further, the workload does not stress the 8 channels of DDR-2400 memory, utilizing only 12% of theoretically available bandwidth on average. Client-server communications over the 10 GigE network did not pose challenges for the NIC, and disk usage under this configuration is negligible.

	Source 0	Source 1	Source 2	Source 3	Source 4	Source 5	Source 6	Source 7
Dest 0	3%	2%	2%	2%	2%	2%	2%	2%
Dest 1	2%	2%	2%	2%	2%	2%	2%	2%
Dest 2	1%	1%	1%	1%	1%	1%	1%	1%
Dest 3	1%	1%	1%	1%	1%	1%	1%	1%
Dest 4	1%	1%	2%	2%	2%	2%	2%	2%
Dest 5	2%	2%	2%	2%	2%	2%	2%	2%
Dest 6	1%	1%	1%	1%	1%	1%	2%	1%
Dest 7	1%	1%	1%	1%	1%	1%	1%	2%

Figure 5: Distribution of memory reads across 8 NUMA nodes

These results suggest that better tuning of the application and software stack may help obtain higher levels of utilization of the underlying hardware, and hence, higher throughput performance.

5.2 NUMA Awareness

As explained in Section 1, the EPYC^{TM} 7601 consists of 4 dies, where by default, each die and its associated memory appears to software as a NUMA node. The dual-socket server in our studies therefore has 8 NUMA nodes.

The degree of *NUMA awareness* of software refers to the extent to which it recognizes the variability of performance in accessing local versus remote memory and adapts its memory allocation accordingly. A well optimized software stack will allocate memory from the node where it is most frequently accessed.

We measured the NUMA awareness of the workload using hardware performance counters, collected on the server system while Cassandra processes requests generated by the client. Our findings are presented in Figure 5. The figure shows how memory read requests are distributed over the 8 NUMA domains. For example the column labeled "Source 0" shows us how often cores in NUMA node 0 issued memory reads to addresses belonging to the 8 NUMA nodes including its own. The percentage is relative to the total number of memory reads issued by Cassandra during its execution, hence all the values taken together add up to a 100%.

The table indicates that requests that originate on any given NUMA node are *equally likely* to access *any* of the NUMA nodes in the system (including itself). The same study for a NUMA-aware workload should have generated the identity matrix. Clearly, Cassandra is unaware of the importance of optimizing for NUMA locality, and significant performance opportunity is available here.

5.3 Lock Contention

Multi-threaded database workloads often face performance scaling challenges due to locks. However, since Cassandra supports weaker, or tunable consistency levels, we did not expect locks to be a first order performance limiter.

However, profiling data collected using the Java Flight Recorder [6] shows that indeed, locks are a major factor that results in poor scaling as thread count increases. Figure 6 plots the number of collisions for the two hottest locks against increasing number of hardware threads. The plot uncovers very poor scalability, showing a 4X increase going from 16 to 32 threads, and a 9X increase from 16 to 48.

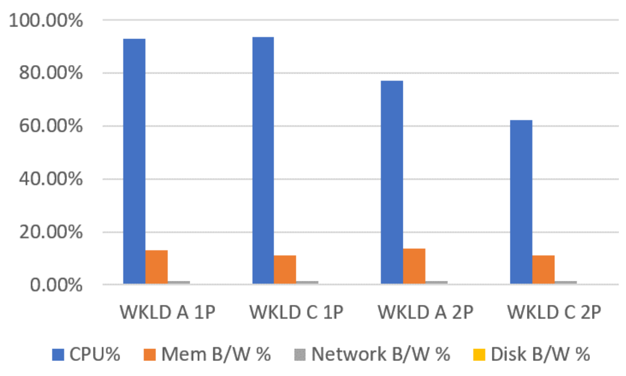

Figure 4: Cassandra resource usage (baseline configuration)

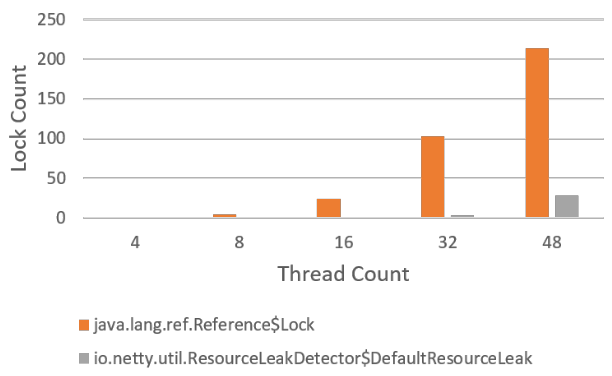

Figure 6: Lock contention in YCSB/Cassandra with increasing hardware threads

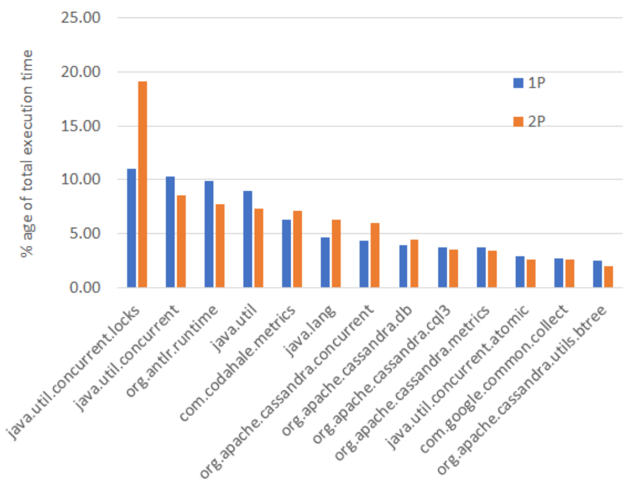

Figure 7: Profiling hot spots indicate significant increase in lock contention with threads

Figure 7 shows how the above behavior results in a skew in the top contributors to profiling hot-spots as we scale up from 64 to 128 hardware threads. In particular, we observe that Java's concurrent lock package nearly doubles its contribution to total execution time. We analyzed stack traces in more detail and determined different sources of these locks. The first is a function that is part of Cassandra's metrics calculations. In fact in a yet unreleased version of Cassandra, this has been recognized as a limitation, and fixes are being been implemented in this area [3]. The other locks we found were traced to the `netty` library, an Java-based networking library used by Cassandra. Till such time these sources of locks are identified and optimized by the community they can manifest as scaling bottlenecks at high thread counts. In our work instead, we focus on how to achieve good scaling on modern server hardware in the presence of such limitations.

6 MULTI-INSTANCE CASSANDRA WITH NUMA BINDING

To help alleviate the issues around poor scaling within a Cassandra server instance, we made two simple changes to our deployment. The first was to deploy multiple server instances on the same physical server instead of one. Further, we deployed these in a NUMA aware manner so as to maximize performance. We look at these decisions in this section and their results, including how it uncovered a performance opportunity in the Java Virtual Machine.

6.1 Multiple Server Instances

In our work, we chose to implement multiple Cassandra servers on our dual-socket system. This can be realized in several ways, based on whether the servers are isolated via virtual machines, containers, or Linux processes. For example, it might be an attractive proposition to create as many servers as there are NUMA nodes on the system. However, this should be done keeping in mind the amount of memory available per server, which depends on how many of the total slots are populated. In our case, a fully populated system would allow for 256GB per server instance, which may be sufficient for a variety of deployments. Also trade offs need to be kept in mind around bottlenecks around networking and storage. In our studies we considered two configurations: (i) one server per socket, and (ii) one server per NUMA node (hence 4 servers per socket, or 8 servers on the system). In each case, we ran the servers on bare metal, *i.e.*, as separate Linux processes.

6.2 NUMA Binding and JVM Bug

NUMA binding helps limit the threads of a computation to a predefined set of cores or NUMA nodes. In addition, the memory accesses they make are also, as far as possible, limited to a prescribed set of memory channels or NUMA nodes. To do this we used both `numactl` based directives in Linux to launch each server on a set of NUMA nodes, and added `-XX:+useNUMA` flag (introduced in Java 6) to our Java flags.

To our surprise, when we studied the performance of single server instance bound to a single NUMA node, we did not achieve the performance levels we expected. We analyzed the memory access patterns using performance counter data (like those presented in Figure 5), and obtained the results reported in the first two columns of Figure 8. We observe that that in spite of the above directives, 75% of accesses still go to remote memory (either remote memory channels within the socket, or remote channels on the other socket). This indicated that the workload, in spite of the NUMA directives, was not using memory in a NUMA-aware manner.

This was root caused to a bug in the implementation of `useNUMA` in OpenJDK8. With the `useNUMA` flag, the JVM allocates as many memory regions as there are NUMA nodes on the platform. These memory regions are called `lgroups` (or Logical Groups) in Java. The idea of these logical groups is to associate every NUMA node with each group and bind those to respective NUMA nodes. This way, the threads running on those NUMA nodes can access memory bound to respective memory regions locally.

Debugging the above observation we determined that irrespective of the number of NUMA nodes that the application uses, Java would allocate the maximum available logical groups on the platform. Hence Java was effectively ignoring the `numactl` directive provided to the application. For example, if Cassandra was bound to one NUMA node (say 0), even then, the JVM during start up, would allocate 8 logical groups. Due to this, only the first `lgroup` would be accessed by the node 0, and the rest would be left unused or would result in remote accesses, depending on the size of the memory heap requested.

After discovering this we implemented a patch to fix this in OpenJDK. With this fix, during initialization, Java now allocates exactly the number of lgroups as the number of NUMA nodes the application is bound to. We tested the patch using OpenJDK8, and it has been accepted for public release in OpenJDK11 [11].

The impact of the patch is shown in in columns three and four in Figure 8: the remote memory accesses of Cassandra were almost *eliminated.*

The resulting performance gains across all the YCSB workloads are presented in Figure 9 for a single server instance bound to a

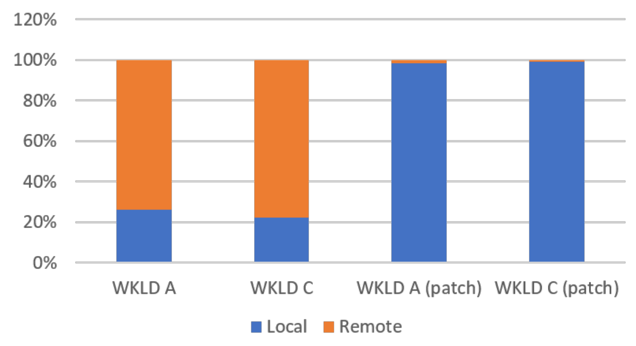

Figure 8: NUMA traffic impact due to Java bug fix

NUMA node. On average, we observed between **26-28%** improvement in throughput across all the workloads underlining (a) the importance of NUMA binding for this workload and (b) the value of the fix to the Java Virtual Machine.

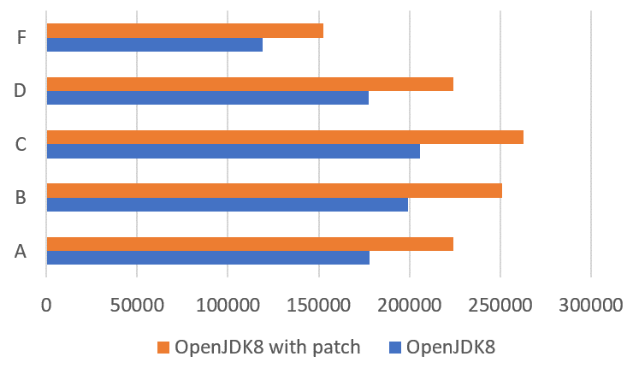

Figure 9: Performance impact of OpenJDK `useNUMA` patch

7 RESULTS

In this section we first review the optimized performance of two configurations: first, running two Cassandra instances (one per socket), and then with as many instances as available NUMA nodes.

7.1 Two Server Instances

In Figure 10, the baseline (4D:1S) is the single, tuned server, with the above JVM fix, mapped to a single socket. This is compared against the performance of two Cassandra instances, one per socket (8D:2S). We observe on average a 1.9X increase in throughput across the 5 YCSB workloads, demonstrating very high socket-level scaling efficiency across a variety of inputs. On measuring hardware resource utilization in this configuration for workload A, we observed 95% CPU utilization and 30% utilization of available memory bandwidth, a dramatic improvement compared to Figure 4, the untuned case.

Figure 11 presents the NUMA traffic distribution for this configuration for the same workload. The higher values along the diagonal imply a larger fraction of memory accesses are now local. The small values in the upper right and lower left triangles imply that cross-socket memory accesses have been minimized. However, the upper left and lower right triangles indicate that within a socket, there is a greater than 50% chance that memory requests will cross intra-socket NUMA boundaries, suggesting that additional performance opportunities remain available.

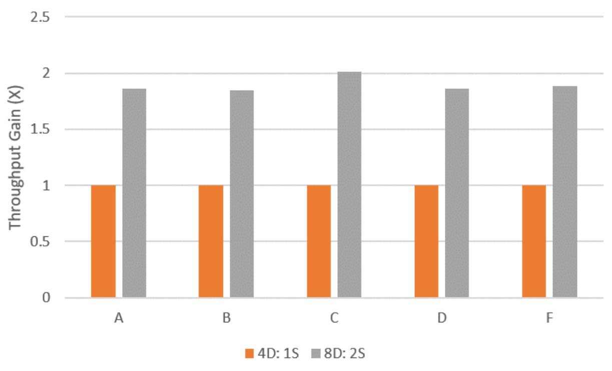

Figure 10: Performance scaling with one Casssandra server per socket

7.2 Variable Server Instances

Figure 12 shows the performance scaling results obtained from running as many Cassandra instances as available NUMA nodes, with the JVM fix, binding each instance to a node. With 4 dies, the average throughput gain across the workloads over the 1 die case within a socket is 3.95x, which represents an in-socket scaling efficiency of 98%. With 8 dies, the throughput gain on average is 7.3X, representing a system-wide scaling efficiency of 90%.

The NUMA traffic distribution for 8 servers processing workload A is presented in Figure 13, demonstrating that with the optimized mapping and bug fixes in the JVM, we were able to almost eliminate all remote memory accesses, and hence achieve the scaling results presented earlier.

8 CONCLUSIONS

In summary, our work has demonstrated methods to extract good "scale-up" characteristics from Cassandra. We have illustrated that a single server instance of Cassandra (i) does not scale well to the large number of hardware threads on offer by modern server systems, and (ii) is unaware of and fails to take advantage of the underlying NUMA topology of modern server hardware that features multiple such NUMA domains. In fact, the second limitation applies to OpenJDK's implementation of Java as well, which has significant implications on a large class of enterprise applications. In separate testing we measured that the fix that we described in this paper results in 11% gain in performance on another enterprise Java workload.

We demonstrated how by mapping Cassandra server instances to NUMA domains and by improving the performance of OpenJDK, Cassandra is able to extract excellent performance scaling characteristics from a single system, achieving over 90% scaling efficiency in both single and dual-socket configurations.

	Source 0	Source 1	Source 2	Source 3	Source 4	Source 5	Source 6	Source 7
Dest 0	5%	2%	2%	2%	0%	0%	0%	0%
Dest 1	2%	5%	2%	2%	1%	1%	1%	1%
Dest 2	2%	2%	5%	2%	0%	0%	0%	0%
Dest 3	2%	2%	2%	5%	0%	0%	0%	0%
Dest 4	0%	0%	0%	0%	5%	2%	2%	2%
Dest 5	1%	1%	1%	1%	2%	5%	2%	2%
Dest 6	1%	1%	1%	1%	2%	2%	5%	2%
Dest 7	0%	0%	0%	0%	2%	1%	2%	5%

Figure 11: NUMA traffic distribution with one Cassandra server per socket

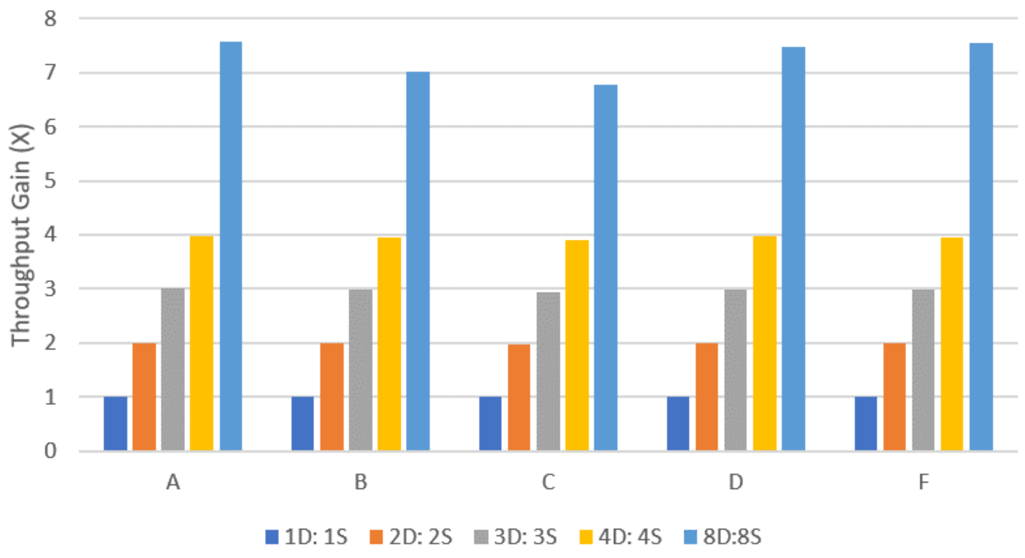

Figure 12: Performance scaling with up to 8 Cassandra servers

	Source 0	Source 1	Source 2	Source 3	Source 4	Source 5	Source 6	Source 7
Dest 0	12.47%	0.17%	0.18%	0.16%	0.15%	0.15%	0.15%	0.15%
Dest 1	0.07%	12.40%	0.08%	0.07%	0.04%	0.08%	0.07%	0.05%
Dest 2	0.03%	0.03%	11.88%	0.04%	0.03%	0.03%	0.01%	0.02%
Dest 3	0.03%	0.03%	0.05%	11.80%	0.03%	0.02%	0.02%	0.01%
Dest 4	0.03%	0.07%	0.03%	0.05%	11.61%	0.06%	0.14%	0.12%
Dest 5	0.01%	0.03%	0.02%	0.04%	0.03%	11.79%	0.03%	0.04%
Dest 6	0.02%	0.03%	0.02%	0.01%	0.04%	0.03%	12.33%	0.02%
Dest 7	0.13%	0.11%	0.16%	0.13%	0.11%	0.16%	0.20%	11.92%

Figure 13: NUMA traffic distribution with one Cassandra server per NUMA node

Instance sizing for Cassandra servers is of course a more complex issue. Database administrators should consider memory and storage requirements per instance, availability, network bandwidth usage and various performance metrics (average and p99 latencies, for example) when choosing an instance size. We hope that our work will provide useful guidance to administrators by highlighting that in addition to these factors, the NUMA architecture of the underlying servers, and NUMA awareness of the software stack needs to be of primary consideration when designing and tuning a deployment.

REFERENCES

[1] Veronika Abramova, Jorge Bernardino, and Pedro Furtado. 2014. Evaluating Cassandra Scalability with YCSB. In *International Conference on Database and Expert Systems Applications*. Springer, 199–207.

[2] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. 2013. Scale-up vs Scale-out for Hadoop: Time to Rethink?. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM.

[3] Cassandra. [n. d.]. Improve latency metrics performance by reducing write path processing. https://issues.apache.org/jira/browse/CASSANDRA-14281

[4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.

[5] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 143–154.

[6] Oracle Corp. [n. d.]. Java Flight Recorder. https://docs.oracle.com/javacomponents

[7] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[8] Advanced Micro Devices. [n. d.]. NUMA Topology for AMD EPYC Naples Family Processors. https://developer.amd.com

[9] Google Inc. [n. d.]. PerfkitBenchmarker. https://opensource.google.com/projects/perfkitbenchmarker

[10] Standard Edition Java Platform. [n. d.]. https://www.oracle.com/technetwork/java/javase/overview/index.html

[11] JDK-8189922. [n. d.]. https://bugs.openjdk.java.net/browse/JDK-8189922

[12] Mijung Kim, Jun Li, Haris Volos, Manish Marwah, Alexander Ulanov, Kimberly Keeton, Joseph Tucek, Lucy Cherkasova, Le Xu, and Pradeep Fernando. 2017. Sparkle: Optimizing Spark for Large Memory Machines and Analytics. *arXiv preprint arXiv:1708.05746* (2017).

[13] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Struc-tured Storage System. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[14] Kevin Lepak, Gerry Talbot, Sean White, Noah Beck, and Sam Naffziger. 2017. The Next Generation AMD Enterprise Server Product Architecture. *Hot Chips: A Symposium on High Performance Chips* 29 (2017).

[15] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at What Cost?. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HOTOS'15)*. USENIX Association, Berkeley, CA, USA, 14–14. http://dl.acm.org/citation.cfm?id=2831090.2831104

[16] OpenJDK. [n. d.]. https://openjdk.java.net/

[17] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anas-tasia Ailamaki. 2015. Scaling up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-aware Data and Task Placement. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1442–1453.

[18] Surya Narayanan Swaminathan and Ramez Elmasri. 2016. Quantitative Anal-ysis of Scalable NoSQL Databases. In *Big Data (BigData Congress), 2016 IEEE International Congress on*. IEEE, 323–326.

[19] Lingjia Tang, Jason Mars, Xiao Zhang, Robert Hagmann, Robert Hundt, and Eric Tune. 2013. Optimizing Google's Warehouse Scale Computers: The NUMA Experience. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA) (HPCA '13)*. IEEE Computer Society, Washington, DC, USA, 188–197. https://doi.org/10.1109/HPCA.2013.6522318

[20] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. *HotCloud* 10, 10-10 (2010), 95.