JexLog: A Sonar for the Abyss

Tobias Scheuer Norman May Alexander Böhm Daniel Scheibli
SAP SE
Walldorf, Germany
{tobias.scheuer|norman.may|alexander.boehm|daniel.scheibli}@sap.com

ABSTRACT

Today's hardware architectures provide an ever-increasing number of CPU cores that can be used for running concurrent operations. A big challenge is to ensure that these operations are properly synchronized and make efficient use of the available resources. Fellow database researchers have appropriately described this problem as "staring into the abyss" of complexity [12], where reasoning about the interplay of jobs on a thousand cores becomes extremely challenging. In this demonstration, we show how a new tool, JexLog, can help to visually analyze concurrent jobs in system software and how it is used to optimize for modern hardware.

1. INTRODUCTION

Following Moore's law, hardware systems have evolved to a point where machines with more than one thousand logical CPUs and several terabytes of main memory become available and affordable for enterprise customers. The idea of enterprise operational analytic database systems that run transactional workload and real-time reporting in the context of a single system mandates that today's DMBS can use these hardware resources effectively.

To provide high transactional throughput and real-time reporting, modern databases such as Oracle 12c, Microsoft SQLServer 2016, IBM DB2 and our own SAP HANA [3] keep most data in DRAM and rely on hardware-supported query processing and parallelization. Usually, parallelization is done on multiple levels of query processing, i.e. running multiple queries in parallel (inter-query parallelism), running multiple relational operators of a single query in parallel (inter-operator parallelism), as well as parallelism inside a single, large operator such as a hash-join or long-running scan (intra-operator parallelism).

To system architects and developers, this high amount of different parallelization opportunities and their combination presents significant challenges: Not only do they have to come up with efficient algorithms that allow to parallelize individual operations, but they also have to ensure that – specifically for complex, concurrent workloads – the interplay of multiple queries and operations is solid and efficient [7, 11].

Using state-of-the-art profiling tools like RotateRight Zoom, valgrind/callgrind, or Linux perf, the identification of such problematic patterns turned out to be very tedious or even impossible, because these tools focus mostly on CPU consumption. This CPU-centric analysis may help to stop some patterns of excessive parallelization as the thread creation overhead might show up, but they fail to identify under-parallelization or synchronization efforts that are not CPU intensive (e.g. futex calls). As a result, the profiling tools prominently show those parts of the code that can be considered OK, while they actually miss the interesting, problematic parts. One notable exception is Intel VTune [5, 6] which provides options to analyze the CPU utilization over time, and also includes NUMA-specific statistics, such as thread migration. However for an advanced analysis, profilers like VTune require annotations at source code level or the use of special libraries. In addition, we found that VTune has difficulties handling very large systems with several hundred cores. Tools like HPCToolkit [1] focus on statistical sampling and thus may fail to identify short-running jobs as a root cause of bad scalability. Our approach aims at combining the best of both worlds. We do actual tracing of events and thereby gain a great level of detail. Doing it efficiently and being restrictive on the amount of data to collect, we limit the overhead to a level that is otherwise only available to sampling-based approaches. Additionally, high-level tools for visualizing query workload are available [4, 9, 10], but these tools focus on providing an abstraction where a detailed analysis is required.

This lack of suitable tools, which are also integrated with the database kernel, motivated us to develop our own thread profiler. Besides our focus on finding and fixing scalability issues due to synchronization, we also made sure that the tool can be used to gather traces in production environments. In our demonstration we present JexLog, a tool to collect and visualize the parallel execution of jobs in complex system software running on modern hardware with several hundreds of logical cores. We show how JexLog helps to quickly identify bottlenecks and problematic patterns in the NUMA-aware, parallel execution of mixed workload. Using several real-world examples, we showcase how this analysis helped to improve the performance and scalability of the enterprise-class, commercial DBMS SAP HANA.

Proceedings of the VLDB Endowment, Vol. 9, No. 10
Copyright 2016 VLDB Endowment 2150-8097/16/06.

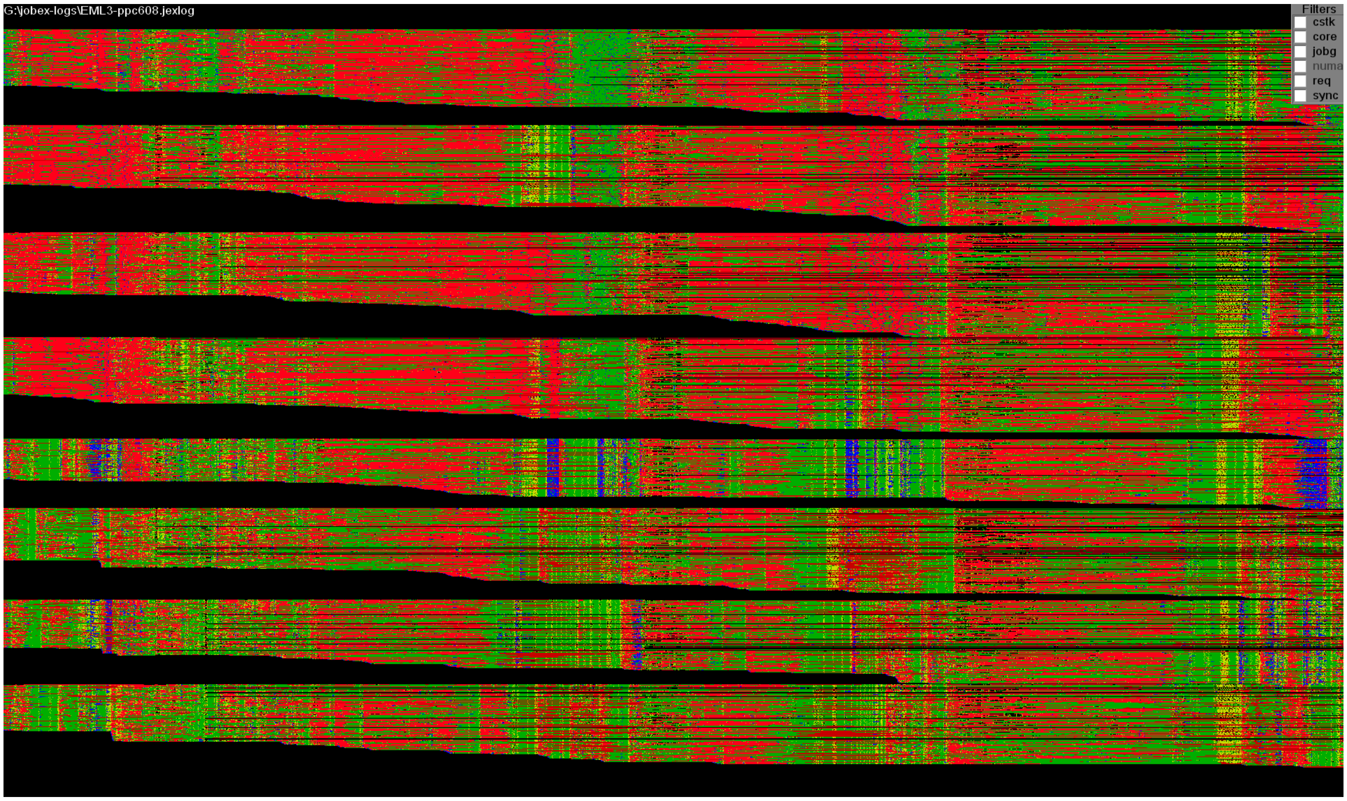


Figure 1: Macro level overview of an 8 socket, 512 logical cores system

2. JEXLOG

The *Job Execution Log (JexLog)* tool consists of two components: The backend part is integrated into the SAP HANA database kernel and collects data about job scheduling, synchronization and the lifecycle of threads. The stand-alone viewer consumes the trace file generated by the backend and provides the interactive analysis of the data collected.

2.1 Data Collection

For the backend we assume that parallelizable workload is mapped to jobs or tasks. These are assigned to priority queues and scheduled to thread pool members by the job scheduler. In SAP HANA we use the implementation described in [7, 8], but the concepts should also apply to other databases as well. Thanks to the job scheduling, it is possible to generate notifications for relevant events like job creation, job scheduling, or begin/end of job execution. When jobs are blocked, e.g. a thread waits on a mutex or waits for the response of a network request, the corresponding events are also generated. Any system that allows to collect these events efficiently could generate job-logs that could be consumed by our viewer component.

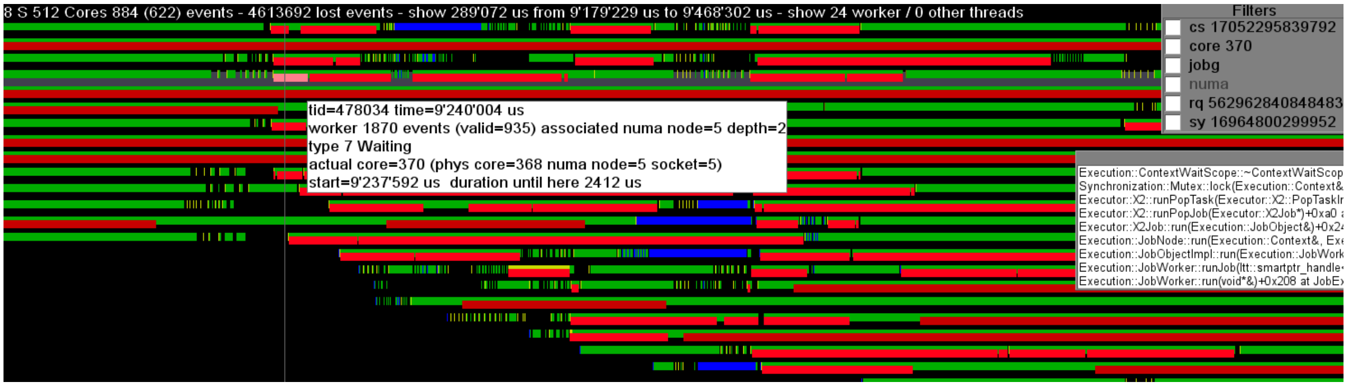
A challenging aspect of collecting the job events is the overhead it may introduce. Especially in highly parallelized programs like the SAP HANA database, this overhead can completely change the way jobs are executed, e.g. lock contention may disappear due to (or thanks to) the effort required to capture events and log them. Therefore an efficient implementation is of paramount importance. We collect events in a pre-allocated per-core data structure in order to keep all data in local caches and to be able to use lock-free algorithms to store the events. We also ensure that an event fits into one cache line, which further reduces overhead. As a result, the data structures scale up well with the number of cores.

In our measurements, we found the overhead induced by the JexLog backend to be below 1%, and hence negligible. However, additionally collecting call stacks incurs a visible overhead of 20μs per event, even with the very fine-tuned implementation for call stacks in SAP HANA. We consider the overhead tolerable especially as it scatters very nicely, so the observed behavior of the jobs, waits, and threads is not changed much. Also, collecting the call stacks is optional, and thus this overhead can be avoided completely.

2.2 Data Analysis

The viewer component of the JexLog is a stand-alone program that consumes the event data stored in a JexLog trace file. It is implemented in C++ using DirectX for fast rendering. An efficient implementation of the viewer is crucial because even for shorter traces the dataset can be hundreds of megabytes in size. Also, the interactive graphical display of thousands of threads demands an efficient implementation of the JexLog viewer.

Figure 1 shows the main screen of the JexLog viewer with example data from an 8-socket machine with 512 logical cores. The x-axis represents the wall-clock time when the events were captured. The y-axis shows the active threads sorted by socket and thread ID. On Linux, where thread IDs increase monotonically for new threads, newly created threads will be at the bottom of each socket group.

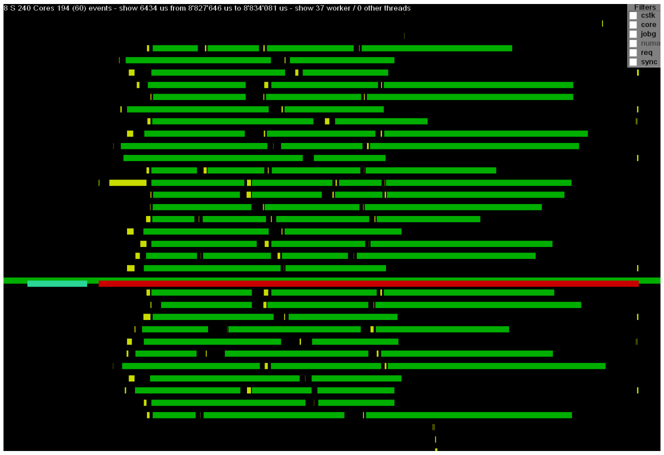


```
8 S 512 Cores 884 (622) events - 4613692 lost events - show 289'072 us from 9'179'229 us to 9'468'302 us - show 24 worker / 0 other threads
```

```
tid=478034 time=9'240'004 us
worker 1870 events (valid=935) associated numa node=5 depth=2
type 7 Waiting
actual core=370 (phys core=368 numa node=5 socket=5)
start=9'237'592 us  duration until here 2412 us
```

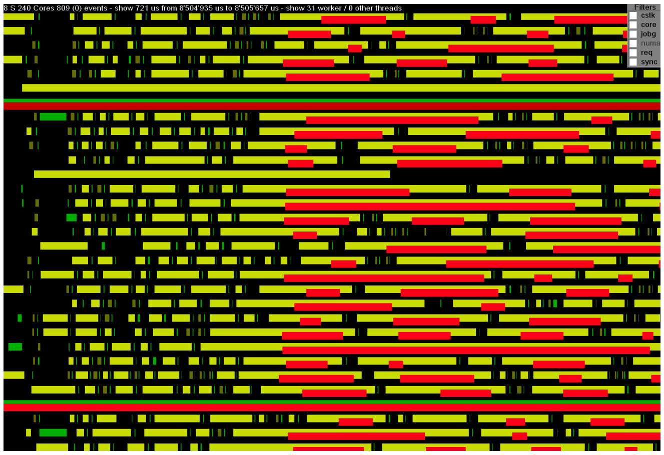
```
Execution::ContextWaitScope::~ContextWaitScop
Synchronization::Mutex::lock(Execution::Context&
Executor::X2::runPopTask(Executor::X2::PopTaskIr
Executor::X2::runPopJob(Executor::X2Job*)+0xa0 a
Executor::X2Job::run(Execution::JobObject&)+0x24
Execution::JobNode::run(Execution::Context&, Exe
Execution::JobObjectImpl::run(Execution::JobWork
Execution::JobWorker::runJob(ltt::smartptr_handle
Execution::JobWorker::run(void*&)+0x208 at JobEx
```

Figure 2: Zooming in on a group of threads



```
8 S 240 Cores 194 (60) events - show 6434 us from 8'827'646 us to 8'834'081 us - show 37 worker / 0 other threads
```

Figure 3: Good parallelization



```
8 S 240 Cores 809 (0) events - show 721 us from 8'504'935 us to 8'505'657 us - show 31 worker / 0 other threads
```

Figure 4: Micro jobs pattern example

A screen almost completely filled with green bars is the desired state; so clearly the red color and the step-patterns indicate that there is something wrong: The red bars point to parts where synchronization blocks the execution of threads. To keep all available cores busy, the SAP HANA job scheduler starts new threads to schedule work on these cores [7], and this leads to the step pattern shown in Figure 1.

To identify the source of the *lock contention* we need to zoom into the dataset. In Figure 2, we have a closer look at the data. However please notice that it is possible to zoom in much further for investigating very short running jobs. Each thread has its own row where jobs and events are shown as color coded bars. A green bar indicates that the `run` method of a job is executed – executing useful code. Yellow bars represent code executed inside the job scheduler to schedule jobs, while blue bars refer to CPU time spent on binding a job to a particular socket. Hence, both indicate overhead of job scheduling, and we aim to minimize the time spent executing this code. Finally, a red bar highlights the fact that a job is waiting on a synchronization primitive. Consequently, this pool thread is not available for processing other jobs. As Amdahl's law teaches us [2], this synchronization limits the speed-up we can achieve with the many cores available to us. Because events can overlap, the bar representing them can too. We decided on partially overlapping the event bars on the y-axis. That way, the user can see the span of each event, but still the currently relevant job phase dominates visually.

The user interacts via mouse or keyboard with the JexLog viewer, e.g. by zoom in and out of the dataset. Figure 2 shows both, the details window and call stack window, which are displayed once the user moves the mouse over a specific thread event. A particular strength of the tool is its filtering capability. Threads can be filtered by combinations of NUMA node, core, job graph, call stack, sync address or request ID. Using the request ID, users can also correlate jobs to their respective SQL statement. Zooming into a particular area of the dataset, there might be time frames where not all threads are active and hence black "voids" might dominate. We resolve these cases by collapsing the empty threads rows, only showing those threads that have events in the visible timeline. These basic interactions are usually sufficient to identify problems related to the multi-threaded execution in the system. In the example shown in Figure 2, the stack trace points us to the source code location where the synchronization happens. With this information, the responsible programmer can be approached and options for the synchronization overhead reduction can be explored based on actual measurements.

The result of successfully reducing synchronization should look similar to Figure 3. There green area indicates that there is no synchronization – only the job in the middle waits for its children to finish. Furthermore, the scheduling overhead is negligible as the yellow bars are very short. However, it seems that before and after the green area, few threads are being executed. This might point to code that is *poorly parallelized* and thus might not utilize the available CPU resources.

A third pattern we demonstrate are *micro-jobs*, i.e. chunks of parallelized work that are too small to justify the overhead of context switching and job scheduling. Figure 4 illustrates this pattern; note that the JexLog viewer shows less than one millisecond of execution time. It is evident that little useful code is executed because there are only very short green bars. This negative effect is exaggerated by the large fraction of time spent for scheduling – as visualized by the yellow bars. In most cases, the root cause is that developers did not expect to have thousands of threads available to parallelize their code and therefore missed to set lower bounds for the work to be done by a job.

3. DEMONSTRATION

Our demonstration is built on a considerable pool of JexLog trace files, including workloads from different industry benchmarks, mixed workloads as well as internal applications. These traces have been collected on NUMA machines with several hundred up to 1152 logical cores.

We will start our demonstration with a high-level overview of complex query workloads running on large NUMA machines, allowing the audience to “stare into the abyss” and get an intuitive feeling for the complexity of the analysis task at hand. We complement this view with a traditional, tree-based visualization of CPU profiling hotspots to differentiate the two and to illustrate the deficiencies of the state-of-the-art tools like perf and valgrind/callgrind.

In a next step, we zoom into the big picture shown by the JexLog viewer. This quickly allows us to identify problematic patterns in the complex workload. We show how to spot problems such as over-parallelization, sequential paths in parallel workloads, and lock-contention by identifying the corresponding pattern in JexLog viewer.

To demonstrate the practical benefits of JexLog viewer for system software development, we will showcase several “war stories” like Figure 2, which we encountered during the development and optimization of the SAP HANA database. By visualizing the system behavior before and after the optimizations, the audience gets an intuitive feeling for the issues encountered and the benefits of the optimizations delivered.

As a last step of our demonstration, we hand over control of JexLog viewer to our audience. Using several examples with ”interesting patterns” we encountered during benchmarking and optimization of SAP HANA, we let the audience explore the characteristics of a complex, multi-threaded system and evaluate the practical usability and benefits of JexLog viewer for system developers.

4. CONCLUSION

The real-world database workloads we present in our demonstration emphasize that getting parallel execution right is still a hard problem today. Database developers need tools to drill down to the lowest level of query execution to identify issues in their code. In our demonstration we share how we identified several typical patterns that limit the effective use of the 1000 cores and more available on modern hardware. JexLog is able to efficiently capture concurrent workload and to visualize it intuitively. The concepts we present can be applied to any database that employs multithreading to parallelize query processing. Thus, we believe that our demonstration is relevant for every database researcher or engineer who aims to make effective use of all available CPU resources on modern hardware.

5. ACKNOWLEDGMENTS

We thank the SAP HANA core development team for their input and feedback on the JexLog tool.

6. REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[2] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *Proc. AFIPS*, pages 483–485, 1967.

[3] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA Database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[4] M. Gawade and M. L. Kersten. Stethoscope: A platform for interactive visual analysis of query execution plans. *PVLDB*, 5(12):1926–1929, 2012.

[5] Intel. Intel VTune Amplifier 2016. https://software.intel.com/en-us/intel-vtune-amplifier-xe. Retrieved February 23, 2016.

[6] A. Marowka. On Performance Analysis of a Multithreaded Application Parallelized by Different Programming Models Using Intel VTune. In *Proc. PaCT*, pages 317–331, 2011.

[7] I. Psaroudakis, T. Scheuer, N. May, and A. Ailamaki. Task Scheduling for Highly Concurrent Analytical and Transactional Main-Memory Workloads. In *Proc. ADMS*, pages 36–45, 2013.

[8] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki. Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-aware Data and Task Placement. *PVLDB*, 8(12):1442–1453, 2015.

[9] D. Scheibli, C. Dinse, and A. Böhm. QE3D: Interactive Visualization and Exploration of Complex, Distributed Query Plans. In *Proc. ACM SIGMOD*, pages 877–881, 2015.

[10] A. Simitsis, K. Wilkinson, J. Blais, and J. Walsh. VQA: vertica query analyzer. In *Proc. ACM SIGMOD*, pages 701–704, 2014.

[11] F. Wolf, I. Psaroudakis, N. May, A. Ailamaki, and K. Sattler. Extending database task schedulers for multi-threaded application code. In *Proc. SSDBM*, pages 25:1–25:12, 2015.

[12] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *PVLDB*, 8(3):209–220, 2014.