Scalable Debug Architecture for Concurrent Applications

Heiner Litz, Benjamin Braun, David Cheriton
Stanford University
heiner.litz@stanford.edu

In traditional database systems, the application logic and the database itself are loosely coupled entities strictly isolated from each other. This separation of concerns simplifies development and maintenance of database applications as, for example, clients do not need to worry about concurrency. On the other hand, it introduces performance overheads. While of a lesser concern for conventional disk based databases, modern in-memory database application performance suffers from this decoupled design approach and the additional latency it introduces. Furthermore, as database applications are becoming increasingly complex, they demand for more flexible data structures while maintaining ACID properties.

We believe that these challenges need to be addressed with a new programming methodology that provides client applications with direct access to the data substrate while maintaining atomicity, isolation and durability properties of database management system (DBMS). In this paper we sketch the architecture of such a system and describe how it copes with the issues of concurrency, correctness and debugability.

In our proposed system, isolation is ensured by executing application threads as individual processes that map a shared memory segment. The segment contains all shared data structures and enables low latency access using conventional load and store operations. Process isolation ensures that threads only modify their local view of memory and writes are only made visible to other threads by an explicit synchronization operation, usually the commit of a transaction.

While our approach improves performance, it exposes the following well known issues of parallel programming to the application logic. First, arbitrary interleaving of threads leads to a state explosion which exacerbates reasoning about all possible executions. Second, due to in-determinism, in the case a bug is found it can be hard to reproduce. Third, bugs that are not fail fast can lead to memory corruption in which case determining the interaction of threads and the root issue is almost impossible. Fourth, runtime checks including assert statements can have a significant impact on performance if constraints concern globally shared data as the verification routine is required to obtain locks for guaranteeing memory consistency.

Our system addresses these challenges through isolation, replay-ability and immutable snapshots. By operating threads in isolation our system can enforce consistency checks whenever a thread performs an update to the globally shared data segment. To minimize the performance impact of these checks, our system tracks writes to shared data locally and only performs validation at certain synchronization points, e.g. the end of a critical section. Our system provides immutable snapshots to allow threads to verify global memory constraints without interfering with other threads, in particular without locking global data structures. To further reduce overheads, our system enables to offload assertion checking to separate verification threads, which execute in parallel on a free core, continuously checking out snapshots and performing validation. The system, furthermore, maintains a history of the globally ordered per thread writesets to reproduce any previous snapshot and to replay memory accesses to the shared data segment. As a result, in case a non fail fast bug is detected, either through one of the verification threads or due to a failing thread, the system rewinds memory accesses until the root cause can be determined. Finally, our system avoids data races by protecting access to shared data, in particular, by informing the programmer each time shared data is accessed outside of a critical section.

Our proposed mechanism is implemented within the virtual memory system. In particular, each thread maintains its own page table mapping in the shared memory segment at the same virtual address. Isolation is enforced by write protecting the segment and by performing copy-on-write in case the segment is written. By creating a thread local copy on each write, data is rendered immutable and as a result snapshots can be generated by simply checking out a particular version of a data item.

Preliminary results have shown that our system provides low latency data access offering the performance of conventional shared memory multithreaded applications while maintaining most of the ACID properties programmers expect from a database system. In particular, as application software now needs to reason about concurrency issues, we provide the necessary debugging capabilities to facilitate development of correct applications. In conclusion, we believe that our system delivers the best of two worlds. Shared memory application performance with database semantics.