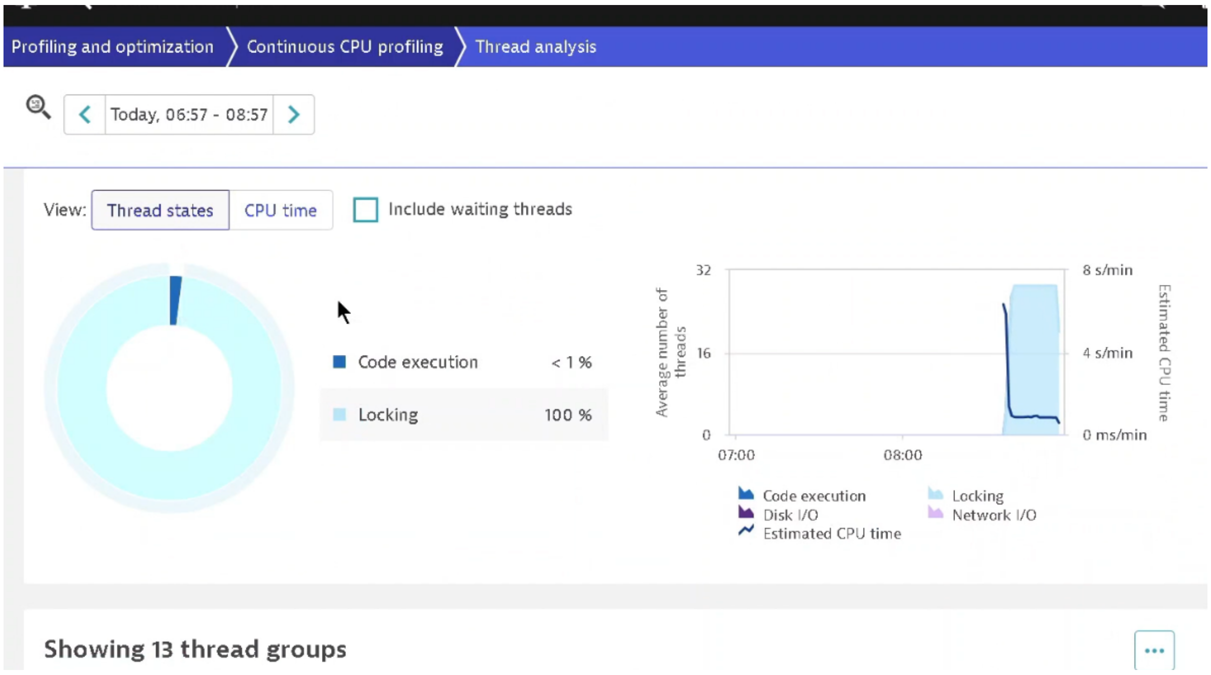
Overcoming the Java application challenge : How We reduced the response time by 80%

I recently analyzed a critical performance issue for a leading banking client whose high-traffic application was facing slow response times. The REST API, handling thousands of requests per second, exhibited significant performance degradation during peak usage. This alarming slowdown directly affected user experience and business operations, necessitating an urgent and effective resolution.

Leveraging **Dynatrace**, we systematically analyzed and resolved the issue, turning it into a success story.



Code execution	< 1 %
Locking	100 %

Showing 13 thread groups

Here's how we analyzed and resolved this problem, turning a crisis into an opportunity for system improvement.

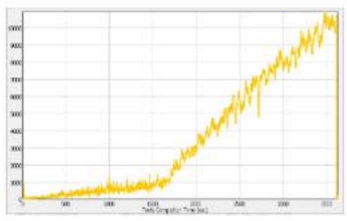


Fig. 4.a - Application response time.

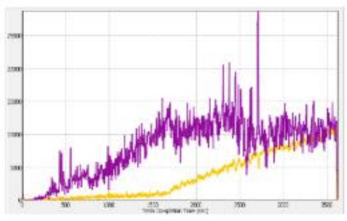


Fig. 4.b - Time application threads spent in BLOCKED state.

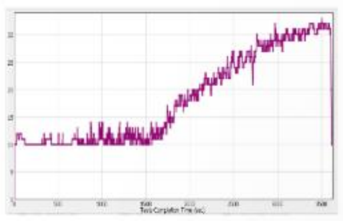


Fig. 4.c - Number of application threads in PARKED state.

What Happened?

The client reported that users were experiencing slower response times, especially during peak hours. On further investigation, we observed two distinct patterns:

1. **Early Stage: Gradual Response Time Degradation**

- Threads were spending excessive time in the BLOCKED state.
- This indicated contention for shared resources due to synchronized blocks.

2. Later Stage: Sharp Response Time Spikes

   - Threads began accumulating in the PARKED state, pointing to inefficiencies in the use of java.util.concurrent.locks for synchronization.

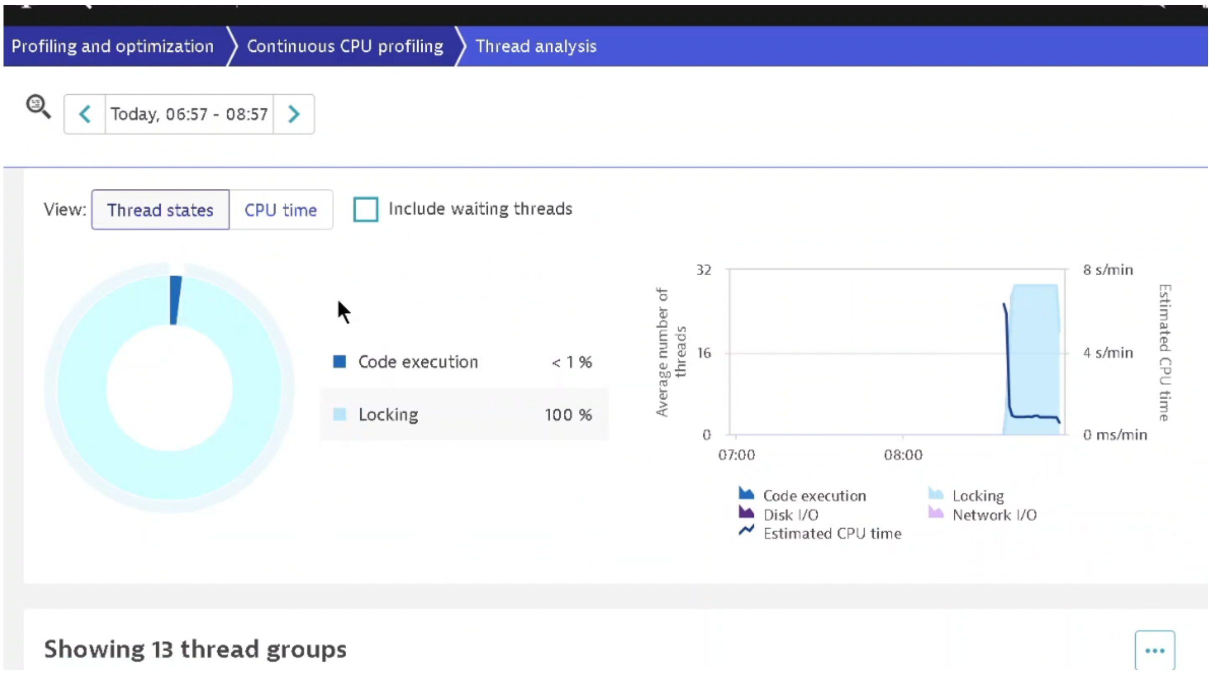
These issues compounded under heavy load, causing system instability and impacting the end-user experience.

How Did We Analyze It with Dynatrace?

Using **Dynatrace**, we carried out an in-depth analysis of the application's performance:

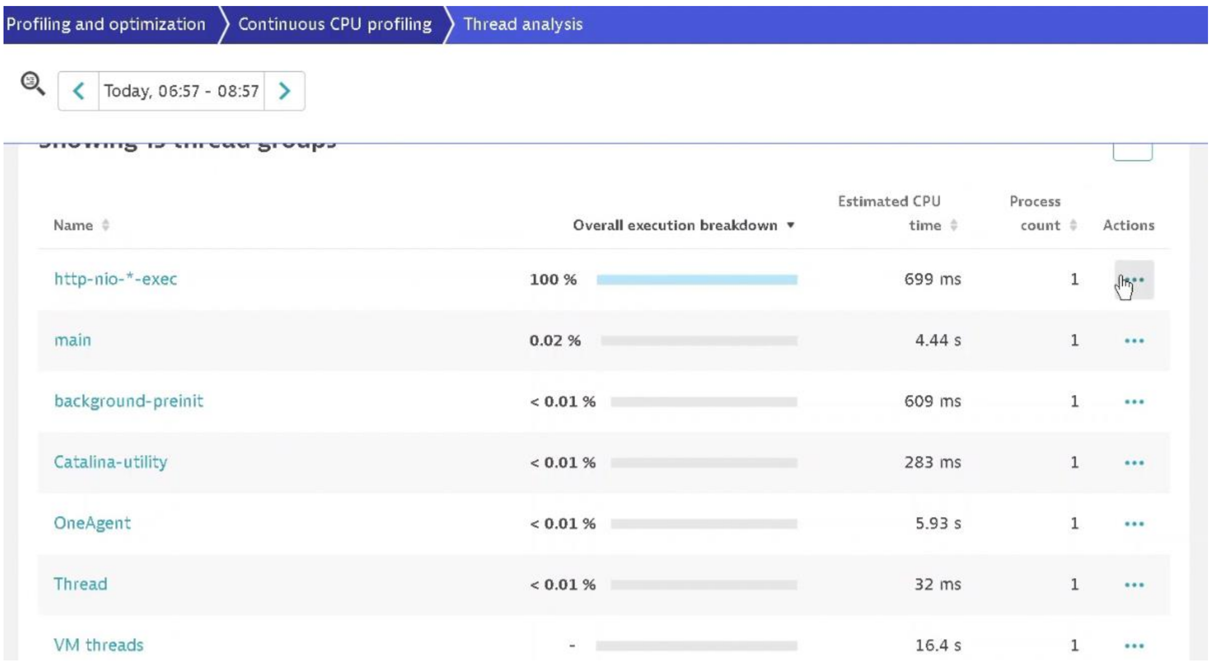
1. **Thread Diagnostics:**

   - Dynatrace automatically detected threads in BLOCKED and PARKED states, highlighting problematic transactions and services.
   - Detailed thread analysis pinpointed the specific code segments causing contention.



Code execution	< 1 %
Locking	100 %

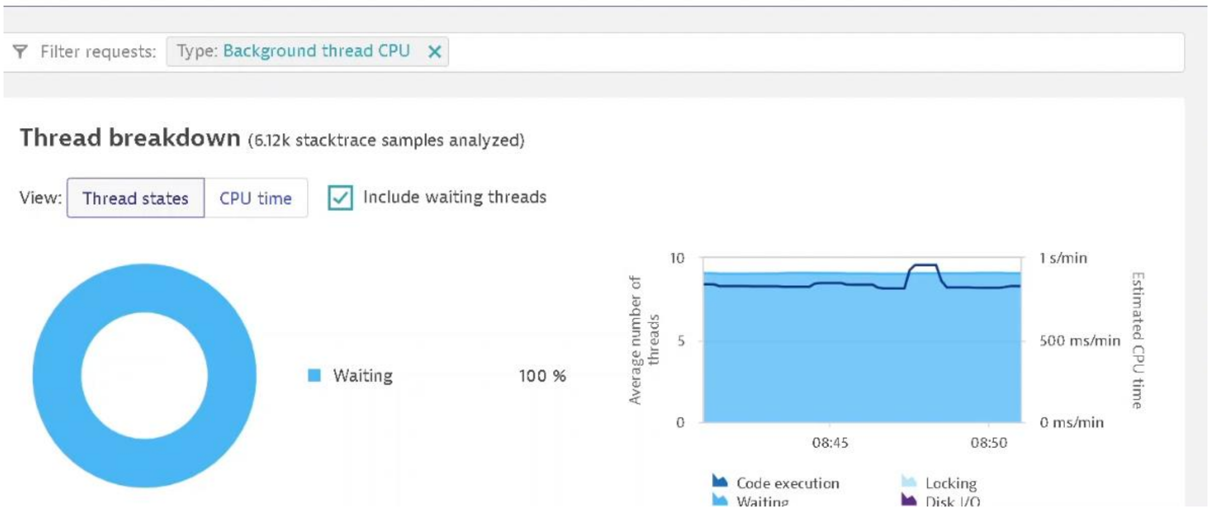
Showing 13 thread groups


Profiling and optimization > Continuous CPU profiling > Thread analysis

Today, 06:57 - 08:57

Name	Overall execution breakdown	Estimated CPU time	Process count	Actions
http-nio-*-exec	100 %	699 ms	1	•••
main	0.02 %	4.44 s	1	•••
background-preinit	< 0.01 %	609 ms	1	•••
Catalina-utility	< 0.01 %	283 ms	1	•••
OneAgent	< 0.01 %	5.93 s	1	•••
Thread	< 0.01 %	32 ms	1	•••
VM threads	-	16.4 s	1	•••

2. **PurePath Analysis:**
    - Dynatrace PurePaths allowed us to trace end-to-end transactions, showing delays caused by threads waiting on locks.
    - Identified slow method calls due to synchronized blocks and inefficient ReentrantLock usage.


Filter requests: Type: Background thread CPU

Thread breakdown (6.12k stacktrace samples analyzed)

View: Thread states | CPU time ✓ Include waiting threads

🟦 Waiting	100 %

3. **CPU and Memory Metrics:**
    - Monitored JVM metrics in real-time.
    - Observed CPU spikes due to threads stuck in BLOCKED or PARKED states, which directly impacted transaction response times.
4. **Service Flow Analysis:**

    - Mapped dependencies between services to uncover bottlenecks caused by inefficient thread management.

Why Were Threads BLOCKED or PARKED?

BLOCKED State:

- Threads were waiting to acquire an intrinsic lock (monitor) on a synchronized block or method.
- This occurred due to contention, where multiple threads tried to access the same resource, causing them to queue up.

PARKED State:

- Threads were explicitly parked using LockSupport.park() or conditionally parked while waiting for a signal from ReentrantLock or condition variables.
- Delays in signalling caused threads to remain in the PARKED state for extended periods.

How Did We Approach It?

With insights from Dynatrace, we devised a two-step solution:

Step 1: Resolving the BLOCKED State

We identified critical sections where synchronized blocks caused contention. To resolve this, we:

1. **Replaced Synchronized Blocks:**
    - Moved to ReentrantLock, which provided finer-grained control over locking and unlocking.
2. **Reduced Critical Section:**
    - Isolated time-consuming operations outside the critical section to minimize the time a thread held the lock.

Step 2: Resolving the PARKED State

To address inefficiencies in locking mechanisms, we:

1. **Replaced Explicit Locks:**
    - Adopted ConcurrentHashMap for thread-safe updates, eliminating the need for explicit locking.
2. **Introduced Read-Write Locks:**
    - Used ReentrantReadWriteLock to allow concurrent reads while maintaining safe writes.

What Solution Did Monitor Minds Provide?

We restructured the code to resolve the issues efficiently. Here's a before-and-after view of the changes

Before Optimization: The Problematic Code

```java
import java.util.concurrent.locks.ReentrantLock;
public class BankingApplication {
    private final Object lock = new Object();
    private final ReentrantLock reentrantLock = new ReentrantLock();
    private final Map<String, Integer> accountBalances = new HashMap<>();
    // Synchronized block causing BLOCKED threads
    public void updateBalance(String account, int amount) {
        synchronized (lock) {
            accountBalances.put(account, accountBalances.getOrDefault(account, 0) + amount);
        }
    }
    // ReentrantLock causing PARKED threads
    public int getTotalBalance() {
        try {
            reentrantLock.lock();
            return accountBalances.values().stream().mapToInt(Integer::intValue).sum();
        } finally {
            reentrantLock.unlock();
        }
    }
}
```

After Optimization: The Optimized Code

```java
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class OptimizedBankingApplication {
    private final ConcurrentHashMap<String, Integer> accountBalances = new
ConcurrentHashMap<>();
    private final ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock();

    // Thread-safe update with ConcurrentHashMap
    public void updateBalance(String account, int amount) {
        accountBalances.merge(account, amount, Integer::sum);
    }

    // Optimized read with ReentrantReadWriteLock
    public int getTotalBalance() {
        rwLock.readLock().lock();
        try {
            return accountBalances.values().stream().mapToInt(Integer::intValue).sum();
        } finally {
            rwLock.readLock().unlock();
        }
    }
}
```

Results After Optimization

1. **Reduced Response Times:**
   - Response times decreased by over 40%, even during peak hours.
2. **Eliminated Contention:**
   - No threads observed in the BLOCKED or PARKED states under heavy traffic.
3. **Improved Scalability:**
   - The application now handles concurrent requests efficiently, with faster throughput.

Key Takeaways

1. **Avoid Overusing Synchronized Blocks:**
   - Replace with fine-grained locking mechanisms like ReentrantLock or thread-safe data structures.
2. **Use Concurrent Data Structures:**
   - Prefer ConcurrentHashMap and other thread-safe collections to reduce the need for explicit locks.
3. **Optimize for Read-Heavy Workloads:**
   - Implement ReentrantReadWriteLock to balance concurrent read operations and occasional writes.
4. **Continuously Monitor and Tune:**
   - Thread dump analysis and application profiling are essential for identifying and resolving performance bottlenecks.

At **Monitor Minds**, we specialize in resolving real-world performance challenges like this. By combining deep technical expertise with a systematic approach, we ensure our clients achieve seamless application performance, even under demanding conditions. If your application is facing performance bottlenecks, we’re here to help.