



QNX SOFTWARE SYSTEMS

Software Optimization Techniques for Multi-Core Processors

Kerry Johnson and Robert Craig
QNX Software Systems
kjohnson@qnx.com

Abstract

Getting your software up and running on a multi-core processor is, in many cases, fairly easy. The real challenge is getting the software to make full use of all the processor's cores. This paper provides examples of multi-core optimization techniques and discusses how developers can use visualization tools to characterize multi-core behavior and measure performance improvements. The paper explores how developers can use threading models to create multiple concurrent tasks and parallel processing; it also discusses how to minimize lock contention by using mutexes to engineer the optimal level of lock granularity.

Introduction

In the past, software developers could rely on faster, more powerful processors to increase the speed of their applications. The industry shift to multi-core processors has eliminated this "free lunch." Now, developers who wish to increase performance must create parallel software that can use multiple processor cores simultaneously.

Parallelism is an important design consideration for image-processing systems, networking control planes, and other performance-hungry, compute-intensive applications. A key metric for such applications is system throughput, and the best way to achieve product differentiation is to either perform operations faster or perform more operations at the same time (i.e. in parallel). Operating systems (OSs) that leverage all available processing cores and tools that provide system-level measurement and visualization are key to implementing parallelism and other multi-core optimization techniques successfully.

Putting all the cores to work: multiprocessing models

OSs vary in how they use multiple processor cores. Some OSs support asymmetric multiprocessing (AMP), some support symmetric multiprocessing (SMP), and some support both. The key difference between these two multiprocessing models is simple: SMP supports parallel software execution, whereas AMP does not.

Asymmetric multiprocessing

In the AMP model, each processor core has a separate OS, or a separate copy of the same OS. Each OS manages a portion of the system memory and a portion of system I/O; see Figure 1. The memory layout and I/O partitioning is typically configured at system bootup.

In AMP, no OS manages the whole system. Consequently, it's up to the system designer to handle the complex task of managing shared hardware resources. For instance, if an application running on one core needs access to an Ethernet port allocated to another core, the application must make the request through an application-level communications protocol. Such protocols must also be used to synchronize tasks running on different cores.

Quad-core AMP system

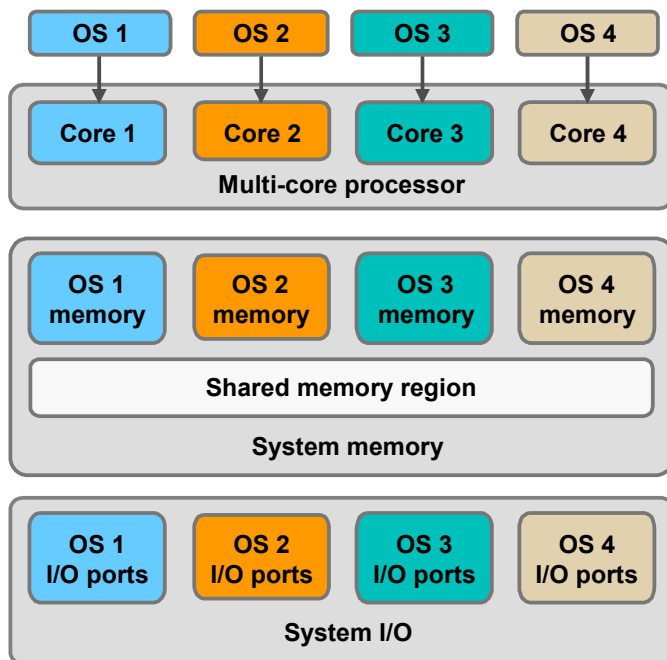


Figure 1 — In AMP, each core has a separate operating system that manages its own memory regions and I/O. The memory regions can overlap to provide a common shared memory region that allows applications on different cores to communicate with one another.

In effect, a quad-core system appears as four separate systems — just like four processor cards in a multi-card system. The main difference is that the quad-core system consumes less power and has a smaller footprint. Also, in a quad-core system, applications running on different cores can communicate with one another via high-speed shared memory; in a system of discrete processors, they would need to use a slower interface such as Ethernet, PCI, or VME.

For compute-intensive applications, a single core of processing capacity is often a limiting factor. With AMP, an application and all of its threads are limited to running on a single core; adding more cores cannot make the application faster.

In AMP, the developer statically assigns a given application to an individual operating system and processing core. This leads to lower overall throughput of the processor complex than SMP since static load balancing leads to scenarios where one or two cores become heavily loaded while other cores are lightly loaded and spend time idling.

Symmetric multiprocessing

In symmetric multiprocessing (SMP), a single instance of the operating system has access to all system memory and I/O. The operating system abstracts the number of CPU cores from

the application software, allowing developers to create parallelized software that can scale as more cores are added. All applications have access to all I/O and all system memory — the OS arbitrates access to these resources. Even though the physical memory is common, the OS can provide memory protection between applications by making use of the processor's memory management unit.

The SMP OS schedules tasks or threads on any available core. Since the OS has all cores at its disposal, tasks or threads can truly execute in parallel. This approach provides the ability to speed up compute-intensive operations by using more than one core simultaneously. With SMP and the appropriate multithreaded programming techniques, adding more processing cores makes compute-intensive operations run faster. Also, developers can employ semaphores, mutexes, barriers, and other thread-level primitives to synchronize applications across cores. The primitives offer synchronization with much lower overhead than the application-level protocols required by AMP.

Quad-core SMP system

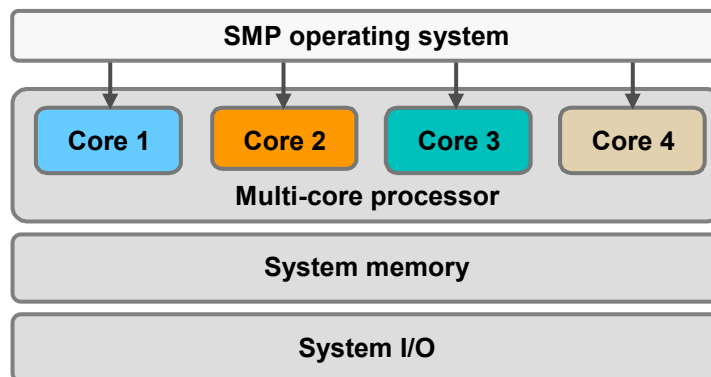


Figure 2 — In the SMP model, a single instance of the operating system manages all processor cores. Applications have uniform access to memory and I/O, and can use the OS to protect memory and arbitrate between various I/O users.

Using visualization tools to optimize the software

By using threading and SMP, developers can introduce software parallelism and thereby achieve greater performance. The following example illustrates how developers can apply these techniques to a quad-core system.

In Figure 3, a single-threaded function called `fill_array()` updates a large, two-dimensional array. The function simply iterates through the array, updating each element. Because the function updates the elements independently (element N value doesn't depend on $N-1$ element), it is easily made parallel.

```

float array[NUM_ROWS][NUM_COLUMNS];

void fill_array()
{
    int i, j;

    for ( i = 0; i < NUM_ROWS; i++ )
    {
        for ( j = 0; j < NUM_COLUMNS; j++ )
        {
            array[i][j] = ((i/2 * j) / 3.2) + 1.0;
        }
    }
}

```

Figure 3 — Single-threaded *fill_array()* function.

To increase the speed of *fill_array()*, the developer can create a worker thread for each CPU and have each worker thread update a portion of the array. Figure 4 shows how the developer can create these threads with the POSIX *pthread_create()* call. To specify the entry point for a thread (where the thread starts running), the developer passes a function pointer to *pthread_create()*. In this case, each worker thread starts with the *fill_array_fragment()* function, shown in Figure 5.

```

void multi_thread_fill_array()
{
    int      thread, rc;
    pthread_t worker_threads[NUM_CPUS];
    int      thread_index[NUM_CPUS];

    // Sync 5 threads, main + threads 2-5
    pthread_barrier_init(&barrier, NULL, NUM_CPUS+1);
    for (thread = 0; thread < NUM_CPUS; ++thread)
    {
        thread_index[thread] = thread;
        rc = pthread_create(&worker_threads[thread],
                           NULL,
                           &fill_array_fragment,
                           (void *)&thread_index[thread]);

        if (rc)
        {
            // handle error
        }
    }
    pthread_barrier_wait(&barrier);
    pthread_barrier_destroy(&barrier);
    return;
}

```

Figure 4 — Main thread creates worker threads to update the array in parallel. The main thread also creates synchronization objects and waits for all worker threads to complete their work. Each worker thread starts executing the *fill_array_fragment()* function.

```
void *fill_array_fragment(int *thread_index)
{
    int col = 0;
    int start_row = 0;
    int end_row = 0;

    // Compute the rows to update by this thread

    start_row = *thread_index * (NUM_ROWS/NUM_CPUS);
    end_row = start_row + (NUM_ROWS/NUM_CPUS) - 1;

    while (start_row <= end_row)
    {
        for (col = 0; col < NUM_COLUMNS; col++)
        {
            array[start_row][col] =
                ((start_row/2 * col) / 3.2) + 1.0;
        }
        ++start_row;
    }
    // Wait at barrier for all threads to complete
    pthread_barrier_wait(&barrier);
    return NULL;
}
```

Figure 5 — Each worker thread updates a portion of the array, then waits at the barrier.

Each worker thread determines which portion of the array it should update, ensuring no overlap with other worker threads. All worker threads can then proceed in parallel, taking advantage of SMP's ability to schedule any thread on any available CPU core.

The main thread must wait for the array to be fully updated before performing additional operations. Thus, synchronization is needed to ensure that all worker threads complete their work before the main thread proceeds. The example uses barrier synchronization, *pthread_barrier_wait()*, where any thread that has finished its work waits at a barrier until all other threads arrive at the barrier.

Now that we've converted the software to a multi-threaded approach, it can scale with the number of CPUs. The example in Figure 6 uses a four-CPU system, but it's easy to adjust the number of worker threads to accommodate more or less processors.

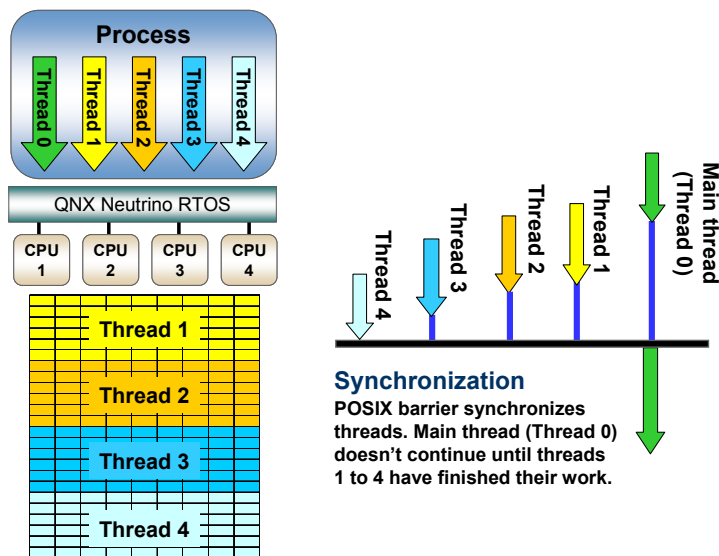


Figure 6 — Multi-threaded approach in which each worker thread updates a portion of the array. Barrier synchronization ensures that all worker threads have finished updating the array before the main thread proceeds.

Visualizing multi-core execution

Although traditional process-level debuggers can help diagnose some problems in a multi-core SMP system, they cannot provide insight into the complex system-level behaviors that arise when multiple threads interact across multiple cores. In fact, a traditional debugger may indicate that a problem exists in one part of the multi-core system when the problem actually resides somewhere else.

To understand these behaviors and to simplify the debugging and optimization of multi-threaded, multi-core systems, developers need a system-tracing tool such as the QNX Momentics[®] system profiler. The system profiler works in conjunction with an instrumented version of the operating system kernel. The instrumented kernel logs all system calls and events, allowing the developer to determine when threads are created, how long they run, and when they complete their work. The logging function provides high-resolution timestamps to ensure accurate timing information.

Figure 7 shows a system profiler trace for the multi-threaded, multiprocessing system. As you can see, the RTOS scheduler has scheduled threads on each available core (indicated by color coding). You can also see how the worker threads wait at the barrier and how the main thread resumes execution after the barrier synchronization. The multi-threaded application is clearly performing as expected.

To measure the performance improvement achieved in this multi-threaded example, we used the QNX Momentics system profiler. First, we measured the timing of a single-threaded process that calls the *fill_array()* function to initialize the array and then proceeds with other work; see Figure 8. The profiler indicates that the total execution time is 5.494739 seconds.

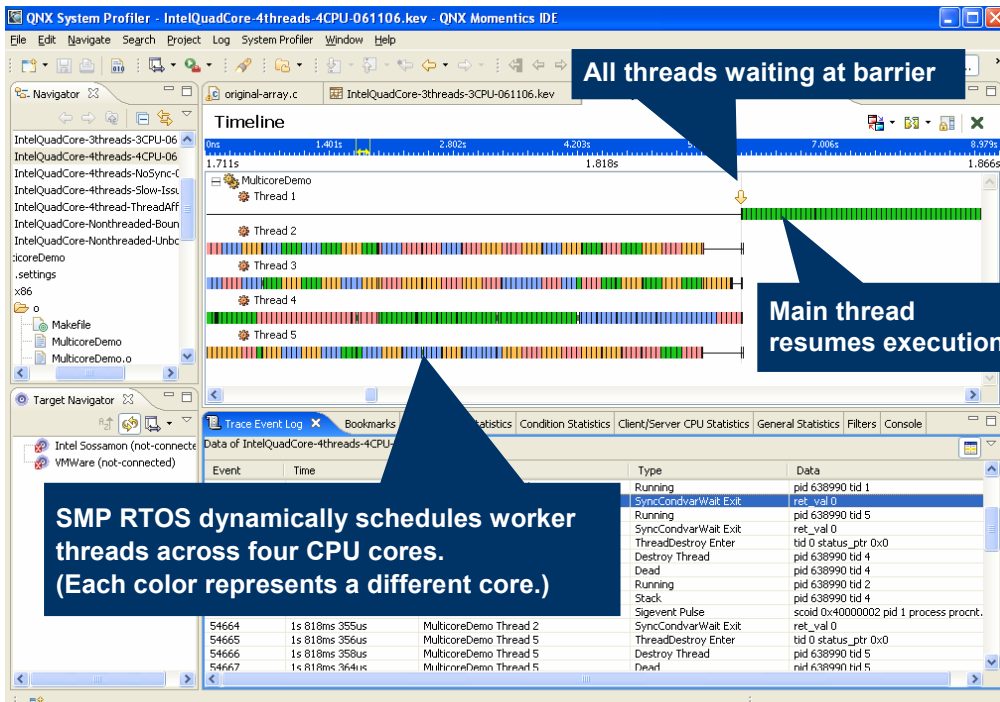


Figure 7 — Using the system profiler to visualize a multi-threaded application on an SMP system.

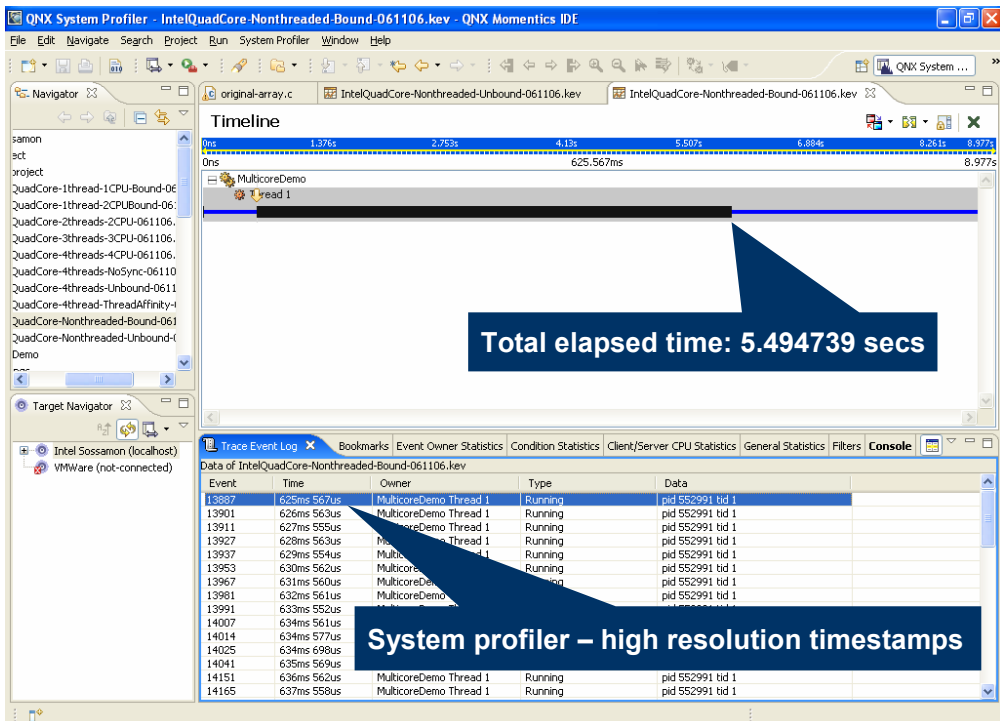


Figure 8 — Single-threaded timing. Trace shows total execution time of 5.494739 seconds.

Figure 9 illustrates the performance improvement achieved by creating four worker threads. Although the four threads are running on a four-CPU system, the profiler shows a 2.7x performance improvement, rather than a 4x improvement. Upon further analysis, the system profiler showed that the additional processing done by the main thread, after *fill_array()* completes, contributes 0.880243 seconds to the overall execution time. This time is consumed in both the single-threaded and multi-threaded implementations. Since this additional processing is single threaded, it doesn't benefit from multiple CPUs. Therefore, it isn't possible to reach a fourfold performance improvement in this case.

This outcome demonstrates Amdahl's Law, which states that the amount of speedup achieved through parallelism is limited by the amount of nonparallel portions of code. When we factored out the linear region, performance increased by 3.95x. The difference between this measured value and a theoretical fourfold improvement can be attributed to the cost of creating and synchronizing four threads.

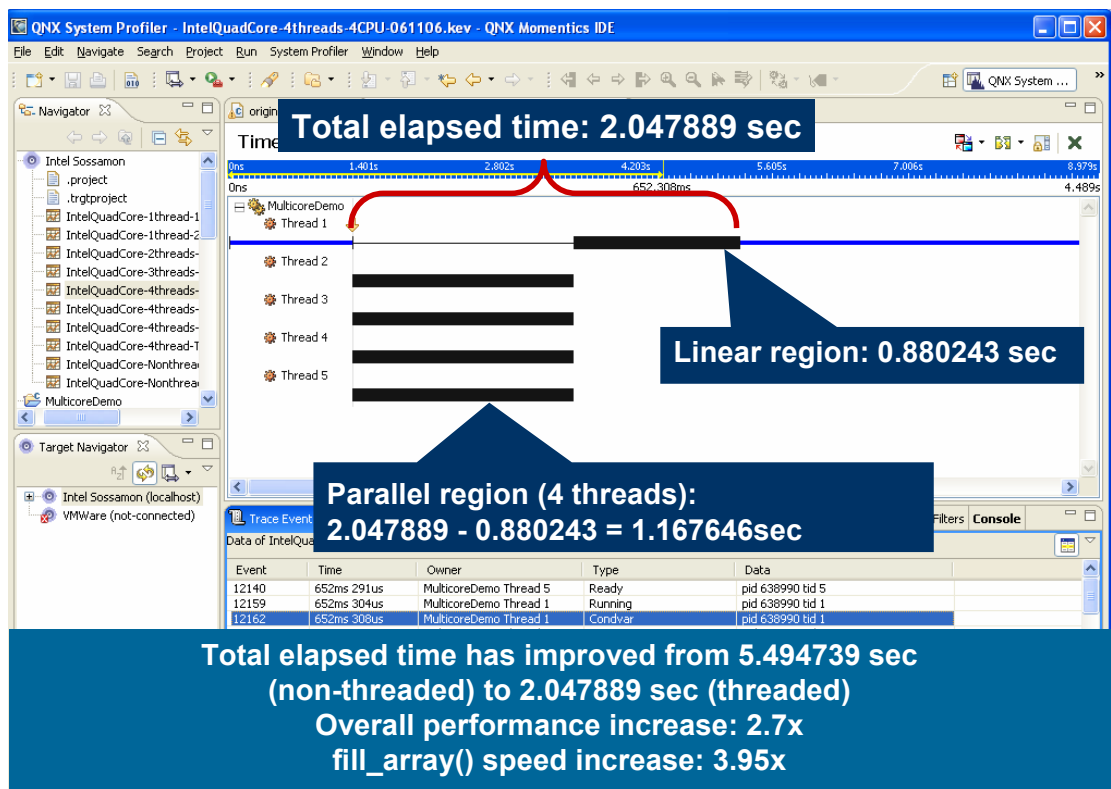


Figure 9 — Multi-threaded implementation with four worker threads and four CPU cores. Total execution time: 2.047889 seconds. System profiler confirms that the parallelized portion of *fill_array()* runs 3.95x faster than the non-parallelized version.

Detecting and reducing resource contention

After introducing parallelism to the design, the developer can implement further optimizations, such as optimizing the locking strategy associated with shared resources.

By allowing threads to run in parallel, a multi-core processor can expose resource contention issues never encountered on a uniprocessor system. For instance, a common symmetric multiprocessing (SMP) bottleneck occurs when two or more threads share a data structure protected by a mutual exclusion lock, or mutex.

A mutex protects a resource by preventing multiple threads from accessing the resource at the same time. Thus, if two threads on separate cores both access resources locked by the mutex, those threads can spend considerable time contending for the lock. Instead of running in parallel with one another, the threads must take turns executing.

For an example of resource contention, consider the array described in this paper. In a uniprocessor environment that has no parallel execution, only one thread will update the array at any given time. Hence the developer needs only a single mutex to prevent multiple threads from updating the array in an invalid way. In a multi-core environment, on the other hand, multiple threads on multiple cores can access the array simultaneously, thereby creating more contention.

To help identify resource contention, a well-designed system-tracing tool can:

- highlight processes that are frequently ready to run but blocked
- generate statistics for threads that are blocked because of resource contention caused by threads on other cores

In Figure 10, you can see the result of running the `fill_array()` program with a single mutex on a four-core system. Four threads run in parallel and complete the required activity in 2.046 seconds. Figure 6 also shows that the threads spend a lot of time contending for the one mutex that protects the array. This contention is indicated by the numerous purple bands, which indicate that a thread is ready to run, but blocked on a mutex. The trace shows significant resource contention (purple bands) in the thread-execution timeline. The contention arises from multiple threads using a single mutex to access a shared data structure.

To reduce the contention, the developer could provide more mutexes for accessing the array. With this approach, each thread can access only a certain region of the array — in essence, each new mutex protects a specific region. Figure 11 shows the reduction in overall processing time when we added 16 mutexes (from 2.046 seconds to 800 milliseconds). It also shows the reduced contention for the mutexes (fewer purple bands).

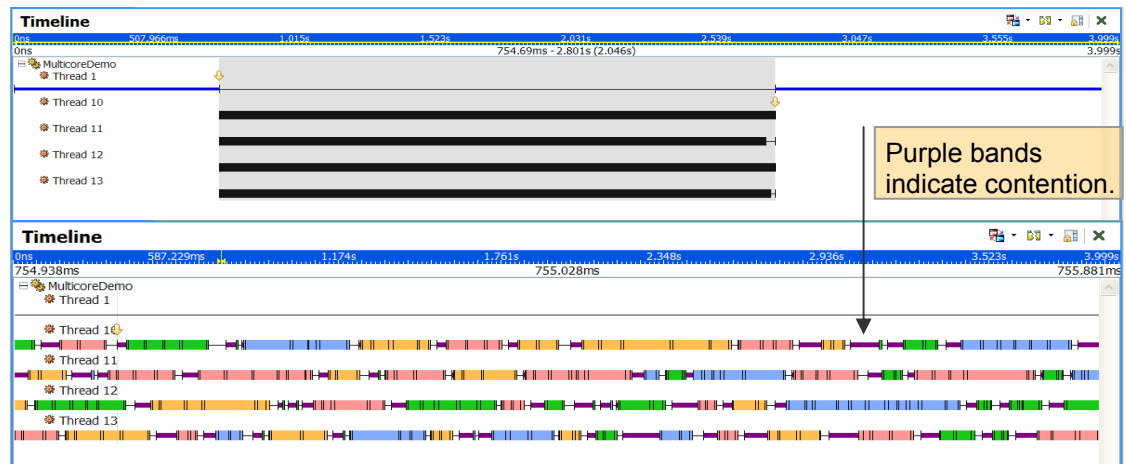


Figure 10 — System trace from four-core system. Total execution for algorithm: 2.046 seconds.

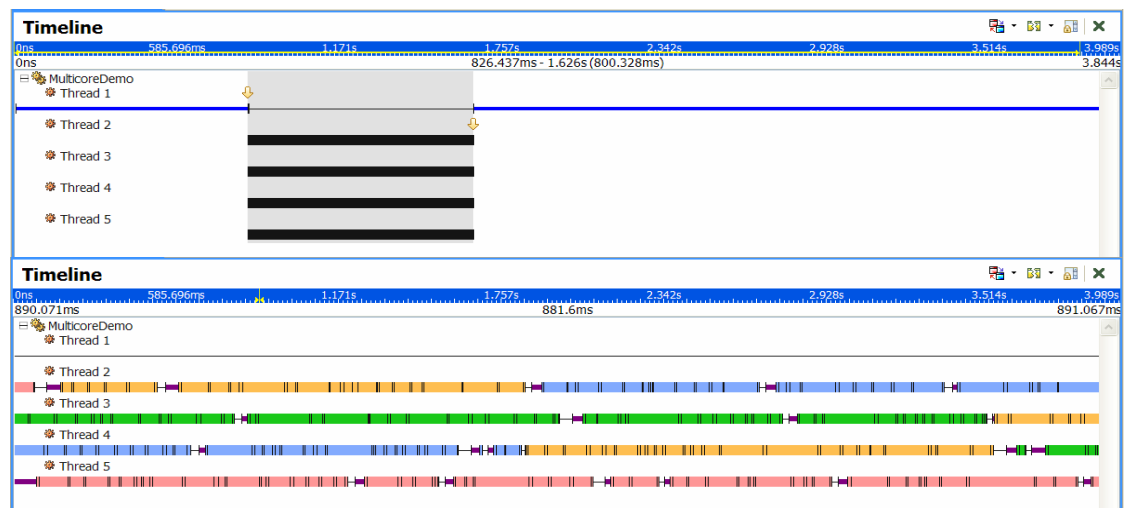


Figure 11 — By adding 16 mutexes to access different data regions, we have shrunk overall execution time from 2.046 seconds to 800.328 milliseconds: a 255% improvement. Moreover, contention for mutexes has visibly decreased.



About QNX Software Systems

QNX Software Systems, a Harman International company, is the leading global provider of innovative embedded technologies, including middleware, development tools, and operating systems. The component-based architectures of the QNX® Neutrino® RTOS, QNX Momentics® development suite, and QNX Aviage® middleware family together provide the industry's most reliable and scalable framework for building high-performance embedded systems. Global leaders such as Cisco, Daimler, General Electric, Lockheed Martin, and Siemens depend on QNX technology for network routers, medical instruments, vehicle telematics units, security and defense systems, industrial robotics, and other mission- or life-critical applications. The company is headquartered in Ottawa, Canada, and distributes products in over 100 countries worldwide.

www.qnx.com