Case Study — Making a Successful Transition to Multi-Core Processors

Dr. Robert Craig and Paul N. Leroux
QNX Software Systems
paull@qnx.com

Introduction
Already, multi-core processors are introducing a new level of performance to desktops, laptops, and enterprise servers. Nevertheless, the benefits for embedded systems are, if anything, greater. Network elements, medical test systems, digital media appliances, and even in-car infotainment units are all growing in complexity, with a voracious appetite for computational power. At the same time, many of these systems must also satisfy rigorous requirements for low weight, low power consumption, and low heat dissipation. Multi-core processors directly address these requirements, by providing much greater processing capacity per ounce, per watt, and per square inch than conventional uniprocessors.

At first glance, migrating software to multiple processors on a single chip may seem like a simple way to increase processing capacity. However, this migration can sometimes introduce complications, particularly if a significant amount of the software was designed with the assumption that the underlying hardware wouldn't provide parallel execution.

In a conventional uniprocessor system, the OS automatically serializes the operation of applications. Multiple tasks may appear to run simultaneously, but in fact only one task runs at any point in time. In a multi-core system, multiple tasks really do run in parallel, and this can expose any incorrect assumptions an application makes about access to shared system resources. As a result, an application that runs perfectly in a uniprocessor system may suddenly behave incorrectly when deployed in a multi-core environment.

Multi-core processors are, in effect, multiprocessing systems on a chip. Consequently, embedded developers must graduate from a serial execution model, where software tasks take turns running on a single processor, to a parallel execution model, where multiple software tasks can run simultaneously. The more parallelism developers can achieve, the better their multi-core systems will perform.
Technology choices

To address these challenges, developers must find tools that can analyze the complex system-level behavior that occurs in a multi-core chip. At any instant, threads can be migrating across cores, communicating with threads on other cores, or sharing resources with threads on other cores — complex interactions that conventional debug tools were never designed to analyze.

Fortunately, vendors like QNX Software Systems have introduced system tracing tools that provide a comprehensive view of multi-core behavior, allowing the developer to visualize interactions between cores and eliminate a variety of performance bottlenecks. Using the information that these tools generate, the developer can reduce resource contention, optimize thread migration, increase parallelism, and achieve the highest possible utilization of every processor core.

Developers must also choose the appropriate form of multiprocessing for their application requirements. More than anything else, this choice will determine how easily both new and existing code can achieve maximum concurrency. As Table 1 illustrates, developers have three basic forms to choose from: asymmetric multiprocessing, symmetric multiprocessing, and bound multiprocessing.

<table>
<thead>
<tr>
<th>Model</th>
<th>How it Works</th>
<th>Key Advantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asymmetric multiprocessing (AMP)</td>
<td>A separate OS, or a separate copy of the same OS, manages each core. Typically, each software process is locked to a single core (e.g. process A runs only on core 1, process B runs only on core 2, etc.).</td>
<td>Provides an execution environment similar to that of uniprocessor systems, allowing simple migration of legacy code. Also allows developers to manage each core independently.</td>
</tr>
<tr>
<td>Symmetric multiprocessing (SMP)</td>
<td>A single OS manages all processor cores simultaneously. The OS can dynamically schedule any process on any core, enabling full utilization of all cores.</td>
<td>Provides greater scalability and parallelism than AMP, along with simpler shared resource management.</td>
</tr>
<tr>
<td>Bound multiprocessing (BMP)</td>
<td>A single OS manages all cores simultaneously. As in SMP, the OS can dynamically schedule processes on any core. However, the developer can also lock any process (and all of its associated threads) to a specific core.</td>
<td>Combines the developer control of AMP with the transparent resource management of SMP. The option to lock threads to any core simplifies migration of legacy code and allows designers to dedicate cores to specific operations.</td>
</tr>
</tbody>
</table>

Table 1 — Three approaches to multiprocessing.

Software Scaling on a Multi-Core System

To scale software effectively on a multi-core system, developers must design applications with parallel operation in mind. Typically, applications written for most modern OSs conform to a multithreaded, process-based model. With this model, applications are divided into
processes that act as containers for resources, such as memory, virtual address space, stack, and so on. Within each process, the application is divided into threads. A thread is the entity within the process that the OS schedules for execution. A thread has configurable elements such as thread priority, which determines how the thread executes in relation to other threads in the system.

In AMP mode, a process and all of its threads are locked to a single processor core. While this approach is useful for running legacy code, it can result in underutilization of processor cores. For instance, if one core becomes busy, applications running on that core cannot, in most cases, migrate to a core that has more CPU cycles available. Though such dynamic migration is possible, it typically involves complex checkpointing of the application’s state and can result in a service interruption while the application is stopped on one core and restarted on another. This migration becomes even more difficult, if not impossible, if the cores use different OSs.

In AMP, neither OS “owns” the whole system. Consequently, the application designer, not the OS, must handle the complex task of managing shared hardware resources, including physical memory, peripheral usage, and interrupt handling; see Figure 1. Resource contention can crop up during system initialization, during normal operations, on interrupts, and when errors occur. The application designer must design the system to accommodate all of these scenarios. The complexity of this task increases dramatically as more cores are added, making AMP unsuitable for newer multi-core processors that integrate four or more cores.

**Figure 1** — In an AMP multi-core system, developers must write code to explicitly manage all shared hardware resources. They must also rewrite or redesign this code when migrating to processors with a greater number of cores.
**SMP: Transparent Resource Management**

Allocating resources in a multi-core design can be difficult, especially when multiple software components have no knowledge of how other components use those resources. Symmetric multiprocessing (SMP) addresses the issue by running only one copy of an OS on all of the chip's cores.

Because the OS has insight into all system elements at all times, it can transparently allocate shared resources on the multiple cores, with little or no input from the application designer. Moreover, it can dynamically schedule any thread or application to run on any available processor core, allowing every core to be utilized as fully as possible — threads can float from one core to another, without any need for checkpointing or for stopping and restarting the application. The OS can also provide dynamic memory allocation, allowing all cores to draw on the full pool of available memory, without a performance penalty. See Figure 2.

Because a single OS controls every core, all intercore IPC is considered local. This approach can reduce the memory footprint and improve performance dramatically, as the system no longer needs a networking protocol to implement communications between applications running on different cores. Communications and synchronization can take the form of simple POSIX primitives (such as semaphores) or a native local-transport capability (such as QNX distributed processing), both of which offer higher performance than complex networking protocols.

Once designed, a process can run equally well on a single-core, dual-core, or N-core system, the only potential change being the number of threads that the application needs to create to maximize performance. In full SMP mode, an RTOS like QNX Neutrino will schedule the highest-priority ready thread to execute on the first available CPU core. As a result, application threads can utilize the full extent of available CPU power rather than being restricted to a single CPU.

---

**Figure 2** — In an SMP multi-core system, the OS dynamically manages hardware resources on the developer’s behalf. Software can migrate from dual-core to quad-core processors, without having to be redesigned.
BMP: Transparent Management plus Developer Control

Bound multiprocessing (BMP), a new approach first introduced by QNX Software Systems, combines the transparent resource management of SMP with the developer control of AMP. Like SMP, BMP uses a single copy of the OS to maintain an overall view of all system resources. BMP goes beyond SMP, however, by allowing developers to “lock” any application (and all of its threads) to a specific core. This approach:

- allows legacy applications written for uniprocessor environments to run correctly in a concurrent multi-core environment, without modifications
- eliminates the processor-cache “thrashing” that can sometimes reduce performance in an SMP system
- enables simpler application debugging than traditional SMP by restricting all execution threads within an application to run on a single core
- supports simultaneous BMP and SMP operation, allowing legacy applications to coexist with applications that take full advantage of the parallelism of multi-core hardware

SMP has long had the capability of tying a particular thread to a single processor — an approach known as thread affinity. BMP extends this thread affinity to the process level by providing runmask inheritance. The runmask is a thread-level entity that determines which processors a thread can run on. In conventional SMP mode, threads are created with a runmask that allows them to execute on all processors. In BMP mode, on the other hand, all threads inherit the runmask from the parent thread. This has the effect of “binding” all of the process’s resources and threads to the same processing core (or set of processing cores), giving the designer complete control over how an application uses a particular core.

BMP offers a viable migration strategy for developers who wish to move towards full SMP, but are concerned that their existing code may operate incorrectly in a truly concurrent execution environment. For instance, some QNX customers have locked their legacy processes to one core, while allowing newer, parallelized processes to float across all cores. Using this approach, the customers were able to maintain a stable environment while correcting and optimizing the legacy processes for full multi-core operation.

Multi-Core Case Study: Perform More Tests, Faster

Now that we’ve reviewed the basics, let’s examine how one development team chose the most appropriate multiprocessing model for their first multi-core project.

Healthcare laboratories today face ongoing pressure to cut labor costs and improve operating efficiency. To achieve these goals, they need clinical test equipment that can:

- **Boost overall throughput** — Equipment must allow laboratories to maximize the number of tests performed per hour.
- **Perform a greater breadth of tests** — A single system must be able to perform diagnostics for an array of infectious diseases, cardiovascular problems, blood viruses, and other conditions.

Healthcare laboratories today face ongoing pressure to cut labor costs and improve operating efficiency. To achieve these goals, they need clinical test equipment that can:

- **Boost overall throughput** — Equipment must allow laboratories to maximize the number of tests performed per hour.
- **Perform a greater breadth of tests** — A single system must be able to perform diagnostics for an array of infectious diseases, cardiovascular problems, blood viruses, and other conditions.
To address these demands, a manufacturer of test equipment needed to add more functionality to their product and to increase system throughput, even though the product had already reached the limits of its processing capacity. Consequently, they decided to migrate to an Intel® Core™ Duo processor. As a design goal, the system developers had to preserve the operation of their existing software while introducing new features that would take advantage of the hardware parallelism offered by their chosen processor.

**Dual-Core Means More Headroom**

The existing design used a high-performance 3.2GHz Intel Pentium® 4 processor. The manufacturer could have upgraded to an even faster Pentium processor, but the incremental increase in performance still wouldn't have addressed the design requirements. In comparison, moving to an Intel Core 2 Duo processor provided the desired increase in throughput, along with ample headroom for new software features.

The manufacturer combined this platform with the QNX® Neutrino® realtime operating system (RTOS) to provide a highly scalable software foundation for their multi-core design. The QNX Neutrino RTOS and the QNX Momentics® development suite offered a logical choice since they combine:

- the hard real-time response needed for the instrument’s control loops
- scalable performance through symmetric multiprocessing (SMP)
- guaranteed operation of legacy code through bound multiprocessing (BMP)
- high reliability through a modular microkernel architecture
- system tracing tools for fast development and troubleshooting on multi-core processors
- integrated Intel C/C++ compiler to achieve maximum performance on the Core 2 Duo processor

**Controlled Transition**

The developers migrated their software to the Core 2 Duo processor in a controlled fashion. Since the existing code base was multithreaded, moving to a dual-core processor with symmetric multiprocessing (SMP) yielded immediate performance gains. In some cases, however, timing-related problems occurred. Troubleshooting revealed that programming errors caused race conditions between threads that access a common memory location or I/O port.

This problem never occurred on a single-processor system since the competing threads didn’t execute in a truly parallel fashion. Fortunately, the solution was simple: add proper synchronization primitives (mutexes and semaphores).

Nonetheless, the development team couldn’t find and fix all such cases and still meet the project schedule. As a result, they had to use most of the legacy code without modifications.
To achieve this goal, the developers used bound multiprocessing (BMP), which allowed selected processes and their associated threads to run exclusively on one core. As described earlier, this approach allows legacy applications written for uniprocessor execution to run correctly, without modifications.

The developers used BMP to run all processes on one core and then to selectively distribute processes across both cores. Using this approach, they maintained a stable environment while correcting and optimizing certain processes for full parallel operation on the multi-core processor. In effect, the team achieved the best of both worlds: a short migration path and significantly greater system throughput.

A Matter of Choice

Should a developer choose AMP, SMP, or BMP? As this case study demonstrates, the answer depends on the problem the developer is trying to solve. It’s important, therefore, that an operating system offers robust support for each model, giving developers the flexibility to choose the best form of multiprocessing for the job at hand. AMP works well with legacy applications, but has limited scalability beyond two cores. SMP offers transparent resource management, but may not work with software designed for uniprocessor systems. BMP offers many of the same benefits as SMP, but allows uniprocessor applications to behave correctly, greatly simplifying the migration of legacy software. As Table 2 illustrates, the flexibility to choose from these models enables developers to strike the optimal balance between performance, scalability, and ease of migration.

<table>
<thead>
<tr>
<th></th>
<th>SMP</th>
<th>BMP</th>
<th>AMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seamless resource sharing</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>Scalable beyond dual core</td>
<td>Yes</td>
<td>Yes</td>
<td>Limited</td>
</tr>
<tr>
<td>Mixed OS environment (e.g. QNX Neutrino + Linux)</td>
<td>—</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>Dedicated processor by function</td>
<td>—</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Intercore messaging Fast (OS primitives)</td>
<td>Fast (OS primitives)</td>
<td>Slower (application)</td>
<td></td>
</tr>
<tr>
<td>Thread synchronization between cores</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>Dynamic load balancing</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>System-wide debug &amp; optimization</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 2 — Attributes of three multiprocessing models.