Processor Affinity or Bound Multiprocessing?

Easing the Migration to Embedded Multicore Processing

Shiv Nagarajan, Ph.D.
Nicola Vulpe, Ph.D.
QNX Software Systems
shiv@qnx.com, nvulpe@qnx.com

Abstract

Thanks to higher computing power and system density at lower clock speeds, multicore processing has gained widespread acceptance in embedded systems. Designing systems that make full use of multicore processors remains a challenge, however, as does migrating systems designed for single-core processors. Bound multiprocessing (BMP) can help with these designs and these migrations. It adds a subtle but critical improvement to symmetric multiprocessing (SMP) processor affinity: it implements *runmask inheritance*, a feature that allows developers to bind all threads in a process or even a subsystem to a specific processor without code changes.

Introduction

Effective use of multicore processors can profoundly improve the performance of embedded applications ranging from industrial control systems to multimedia platforms. Designing systems that make effective use of multiprocessors remains a challenge, however.

QNX Neutrino RTOS

The QNX® Neutrino® RTOS has been multiprocessor capable since 1997, and is deployed in hundreds of systems and thousands of multicore processors in embedded environments. It supports AMP, SMP — and BMP.

The problem is not just one of making full use of the new multicore environments to optimize performance. Developers must not only guarantee that the systems they themselves write will run correctly on multicore processors, but they must also ensure that applications written for single-core processors will not fail or cause other applications to fail when they are migrated to a multicore environment. To complicate matters further, these new and legacy applications can contain third-party code, which developers may not be able to change.

The current choices for multicore implementations are asymmetric multiprocessing (AMP), which can by either heterogeneous or homogenous; symmetric multiprocessing (SMP), which may use processor affinity to mitigate the risks of migrating code written for single processors as well as to manage specific performance issues; and bound multiprocessing (BMP), a QNX innovation that resolves the reliability and performance issues which processor affinity only partially addresses. BMP gives developers the means to control where all threads in their system will run — even dynamically created threads.

Asymmetric Multiprocessing

Asymmetric multiprocessing has been around in various forms at least since the 1970s, when master-slave CPU architectures were already being implemented in commercial systems. With AMP, a separate OS, or a separate instantiation of the same OS runs on each processor core.

AMP may still be the solution of choice in some environments; for instance, where intercore messaging speed is not required, or where different OSs are needed. AMP has several characteristics in its favor. The migration path for legacy applications from single-processor systems to AMP systems is relatively straightforward, and AMP offers a direct mechanism for controlling how CPU cores are used. In addition, AMP provides an execution environment similar to that of the single-processor systems familiar to most developers, who in most cases can work with standard debugging tools and techniques.

However, AMP also has a number of disadvantages, which make it unsuitable for many systems. First, AMP has proven difficult to extend beyond dual-core processing, while multicore chips have expanded to eight, 16, or more cores. Second, with AMP, processor and physical memory allocation are static; the OS will always run a process on the same core, even when other cores are idle, so one core can end up being under- or overused. If it can be done at all, any dynamic load balancing must be done by the applications, which must manage coordination between the cores. This coordination requires complex checkpointing of state information, and may even entail service interruptions as an application is stopped on one core and restarted on another. Finally, AMP provides no mechanism for system-wide debugging; processes in a core must be debugged in isolation — even though processes on other cores may depend on them.

Homogenous & heterogeneous AMP

AMP can be either *homogeneous*, where each core runs the same type and version of OS, or *heterogeneous*, where each core runs either a different OS or a different version of the same OS.

In a heterogeneous environment, the developer must either implement a proprietary communications scheme, or choose two OSs that share a common infrastructure (likely IP based) for interprocessor communications. To help avoid resource conflicts, the OSs should also provide standardized mechanisms for accessing shared hardware components.

In a homogeneous AMP environment, developers can make best use of the multiple cores by choosing an OS that offers a distributed programming model. Properly implemented, this model allows applications running on one core to communicate transparently with applications and system services (such as device drivers and protocol stacks) on other cores, but without the high CPU utilization imposed by traditional forms of interprocessor communication.

Thus, except in environments where diverse OSs are expressly called for, where dual-core processors suffice, and where intercore messaging speed can be handled by the application, AMP is of limited value.

Symmetric Multiprocessing

Symmetric multiprocessing dates back several decades, but its advent is intimately linked to the increased availability of multicore chips in the mid 1990s. It has since become the solution of choice for innumerable embedded systems. As opposed to AMP systems, SMP systems run a single copy of the OS that manages all the cores on a chip. Because this single OS

always has insight into all system elements, it can allocate resources on multiple cores with little or no input from the application designer. It makes the entire compute power of a multicore chip available to applications at all times.

If the OS provides appropriate preemption and thread prioritization capabilities, it can also help the application designer ensure that CPU cycles go to the application that needs them the most. And, because it involves only one OS, development and debugging of an SMP system is less onerous than with AMP systems: developers can use standard OS primitives rather than complex IPX (Internetwork Packet Exchange) mechanisms[1].

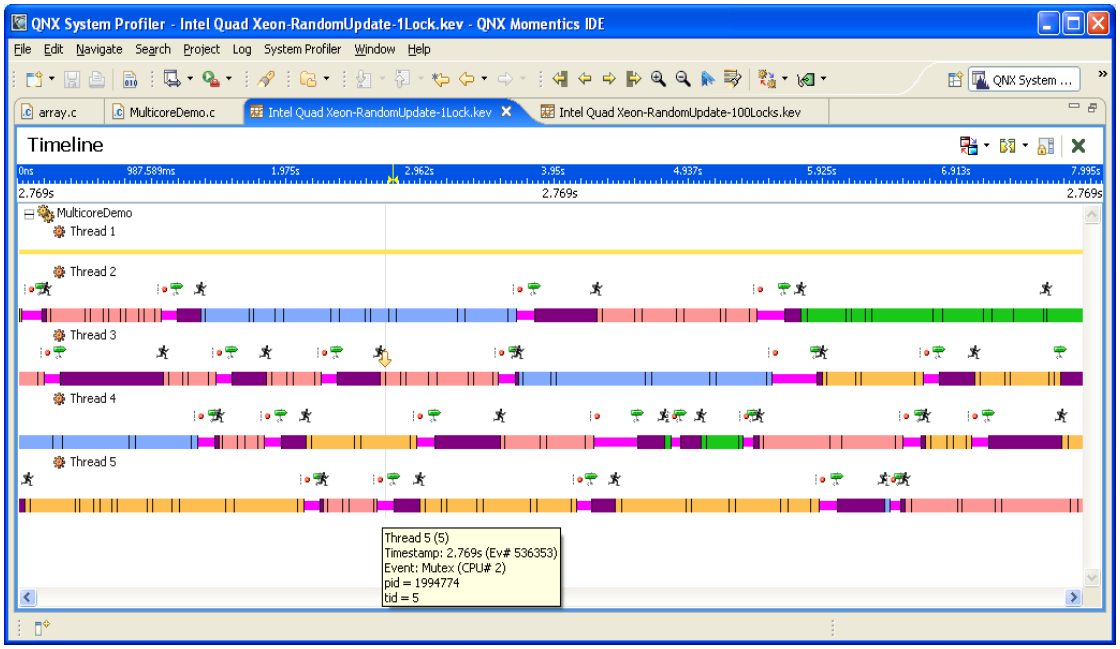

Figure 1 — Tools such as the QNX System Profiler shown in Figure 1 have proven themselves invaluable for debugging and optimizing multicore systems.

Despite its manifold benefits, SMP does present some challenges. Not least significant of these challenges are cache thrashing (when threads migrate between cores, they lose the benefit of data locality in the cache, which often results in cache thrashing); and latent bugs or design features, such as FIFO scheduling, that render code written for single-core processors unreliable on SMP systems.

[1] Craig, Robert and Kerry Johnson (2007). *Software Optimization Techniques for Multi-Core Processors*. QNX Software Systems. http://www.qnx.com.

Processor Affinity

Processor affinity (or thread affinity) solves many of the problems inherent to SMP. It is already well-known and well documented. In brief, processor affinity gives developers the ability to assign threads to specific processors either temporarily or permanently, effectively removing them from the load-balancing equation.

By allowing applications that share the same data set to run exclusively on the same processor, processor affinity eliminates the cache thrashing that can reduce performance in an SMP system. Processor affinity can help legacy applications that use poor techniques for synchronizing shared data run correctly, also by forcing them to run on a single processor. And it offers simpler application debugging than SMP, again by constraining all execution threads in an application to run on a single processor.

Thus, for example, if moving a thread between cores causes cache thrashing and reduces performance, or if a thread handling some graphics rendering endangers completion of critical threads that must meet realtime requirements, the developer can limit the offending threads to a specified processor or processors. The thread that causes cache thrashing when moved between cores can be confined to one processor, while on an eight-core system the graphics thread might be excluded from all but two processors, leaving six processors available for the critical realtime threads.

Limits of Processor Affinity

Though it provides a means for resolving many challenges facing developers working with SMP systems, processor affinity does not solve them all. To use processor affinity to constrain a thread to a specified core or to exclude it from specified cores, a developer sets its *runmask*. A runmask is simply a bitmap that tells an SMP OS where it may or may not run a thread. An important limitation of processor affinity is that a runmask must be *explicitly* set for every thread. This limitation can lead to two problems in particular.

First, if an application uses third-party code which may not be modified, this code can, in effect become a liability in a multicore environment. Assuming that the application, or some processes in the application, must be constrained on an SMP system, either to meet performance targets or simply to ensure that they do not fail, it is essential that the processor affinity constraints apply to all relevant threads, including those in the third-party code.

However, because the third-party code may not be changed, runmasks can not be set for its threads, and whatever the constraints placed on other threads in the application, these threads will remain free to move to any processor in the system — with unknown consequences. For example, third-party code that uses FIFO scheduling and no locking (a technique often used on single processors to guarantee sequential execution) might run perfectly well on a single-processor system, but could not be used on a multicore system.

Second, processor affinity provides no mechanism for setting the runmasks of dynamically spawned threads. Whatever the processor affinity of the parent thread, a child thread enters the world free to move about and execute on any processor in the system. Thus, when migrating applications to a multicore system, or even when developing applications for these systems, developers must find a way to intercept every child thread immediately after it has been created and set its runmask appropriately, a daunting task in anything but the simplest applications — applications which likely wouldn't require a multicore processor in the first place.

In summary, simple processor affinity sets runmasks for individual threads to constrain them to specified processors, but it provides no mechanism for controlling where dynamically created threads or processes initiated by third-party code will execute.

Bound multiprocessing

A QNX innovation, bound multiprocessing (BMP) adds subtle but critical improvements to SMP processor affinity, eliminating with a stroke — or, more accurately, some bitmaps — two problems left unresolved by standard processor affinity: constraining threads in third-party code, and constraining dynamically created threads.

BMP solves the problem of how to constrain both individually created threads, and complete hierarchies of thread creations: a thread, its child threads, any of their child threads, and so on. With BMP, a thread can inherit the processor affinity of the thread that spawned it. Threads spawned by a thread bound to a specific processor can also be automatically bound to that processor, ensuring that the system remains SMP safe, even as it dynamically adds threads.

Runmask Inheritance

BMP uses *runmask inheritance* and *inheritance masks*. Runmask inheritance is simply the process by which a thread inherits the runmask of its parent thread. It allows developers to bind all threads in a process or even a subsystem to a specific processor or processors.

By using runmask inheritance a developer can ensure that third-party code executes on a single processor without changing, or even understanding completely, the inner workings of this code. All he or she needs to do is set the runmask for the thread or threads that will use the third-party code, and the threads in that code will execute only on the specified processor or processors.

The solution for dynamically created threads that may not be SMP safe is the same. The developer only needs to set runmasks for parent threads; child threads will inherit the runmasks and be as movable or as constrained as the parent threads.

```
unsigned    num_elements = 0;
int         *rsizep, masksize_bytes, size;
unsigned    *rmaskp, *imaskp;
void        *my_data;

/* Determine the number of array elements required to hold
 * the runmasks, based on the number of CPUs in the system. */
num_elements = RMSK_SIZE(_syspage_ptr->num_cpu);

/* Determine the size of the runmask, in bytes. */
masksize_bytes = num_elements * sizeof(unsigned);

/* Allocate memory for the data structure that we'll pass
 * to ThreadCtl(). We need space for an integer (the number
 * of elements in each mask array) and the two masks
 * (runmask and inherit mask). */

size = sizeof(int) + 2 * masksize_bytes;
if ((my_data = malloc(size)) == NULL) {
    /* Not enough memory. */
    …
} else {
    memset(my_data, 0x00, size);

    /* Set up pointers to the "members" of the structure. */
    rsizep = (int *)my_data;
    rmaskp = rsizep + 1;
    imaskp = rmaskp + num_elements;

    /* Set the size. */
    *rsizep = num_elements;

    /* Set the runmask. Call this macro once for each processor
       the thread can run on. */
    RMSK_SET(cpu1, rmaskp);

    /* Set the inherit mask. Call this macro once for each
       processor the thread's children can run on. */
    RMSK_SET(cpu1, imaskp);

    if ( ThreadCtl( _NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT,
                    my_data) == -1) {
        /* Something went wrong. */
        …
    }
}
```

Figure 2 *— Setting up a* runmask *and an* inheritance mask *for BMP in the QNX Neutrino RTOS.* [2]

[2] *Multicore Processing User's Guide* (2006-2009). QNX Software Systems. pp. 14-18.

Inheritance Masks

Inheritance masks are another QNX innovation. They enable developers to fully exploit the possibilities offered by runmask inheritance, by performing a little bit of genetic engineering and altering the inheritance characteristics of child threads.

Simply put, developers can use inheritance masks to specify whether a child thread inherits its parent's runmask or begins life with a new runmask, which can in turn be inherited or not inherited by its child threads. This additional capability gives developers designing and implementing multicore systems, or migrating applications to multicore systems full control of where every thread, whatever its origins, may or may not run.

Conclusion

Considering the benefits offered by multicore processing: great computing power at low clock speeds and with low heat generation, the demand for multicore processing support in embedded applications will only increase. Asymmetric multicore processing and symmetric multicore processing each have their roles. Choosing one over the other depends on the problems to be solved. When SMP is used, it is often necessary to limit where specific threads or even applications may run.

Soft affinity

Whenever possible, the QNX Neutrino RTOS ensures that a thread is scheduled to run on the same CPU that it was last running on in order to improve cache performance.

Bound multiprocessing offers developers a significant improvement over simple processor affinity; BMP gives them full control over where even dynamically created threads execute, and is therefore the optimal choice both for migrating and for developing multicore systems.

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® Tool 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 vehicle telematics and infotainment systems, industrial robotics, network routers, medical instruments, security and defense systems, 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