WIND RIVER

Best Practices: Adoption of Symmetric Multiprocessing Using VxWorks and Intel® Multicore Processors

Lori Matassa, Digital Enterprise Group, Intel Corporation
Yatin Patil, VxWorks Product Division, Wind River

Table of Contents

Executive Summary

Three main trends that are shaping the embedded device market today are smaller form factors, standardization, and improved performance per watt. In addition, applications across all device categories, from signal processing in aerospace devices to multimedia processing in consumer devices, continue to require higher performance. However, the traditional method of achieving higher performance, increasing the processor clock frequency, causes the thermal dissipation of the processor to increase as well, and higher thermal dissipation drives up the overall platform energy requirements. In contrast, multicore technology benefits devices and applications by improving performance per watt ratios and by reducing the board real-estate required by multiple-processor designs that place only one core in each package. This allows software developers to leverage symmetric multiprocessing (SMP) capabilities at lower price points than ever before.

The challenge for embedded developers is identifying and extracting parallelism within serially designed applications and designing software so that it scales as the number of cores increases. This paper discusses the combined performance and SMP capabilities of the Intel Core microarchitecture and Wind River VxWorks SMP operating system and walks through SMP software design considerations, methodologies, and Wind River tools that help with each step of the multicore software development cycle.

Introduction

Multicore, also referred to as chip multiprocessing (CMP), is a processor technology that provides parallel processing capabilities by containing multiple independent execution cores and instruction pipelines within one packaged processor assembly. It supports increased platform performance with lower power cores. System software utilizes this technology for load balancing and simultaneously executing the workload across the multiple cores.

Multicore can be used to optimize system performance by using the additional cores to execute multiple streams of code simultaneously. Multicore technology reduces the costs of multiprocessing and provides unprecedented increases in capacity for new uses and functions such as hardware consolidation and robustness of virtualized systems.

Applications can benefit from multicore technology by simultaneously processing multiple concurrent functions within one application, as well as by decomposing data to be equally distributed and simultaneously processed across all cores. Multicore technology can also increase application responsiveness by allowing one or more tasks to execute simultaneously, while another task waits for data or a user command.

Intel® Core Microarchitecture Ingredients

Intel Core microarchitecture includes five new and innovative ingredients enabling new levels of energy-efficient performance.

Intel® Wide Dynamic Execution

First implemented in the P6 microarchitecture, Intel Wide Dynamic Execution is a combination of techniques (data flow analysis, speculative execution, out of order execution, and super scalar) that enables the processor to execute more instructions on parallel, so tasks are completed more quickly. It enables delivery of more instructions per clock cycle to improve execution time and energy efficiency. Every execution core is 33% wider than previous generations, allowing each core to fetch, dispatch, execute, and retire up to four full instructions simultaneously. Further efficiencies include more accurate branch prediction, deeper instruction buffers for greater execution flexibility, and additional features to reduce execution time. Intel Wide Dynamic Execution enables increased instruction execution efficiency, thus boosting performance and energy efficiency.

Intel® Advanced Digital Media Boost

Intel Advanced Digital Media Boost is a feature that significantly improves performance when executing Intel Streaming SIMD Extension (SSE/SSE2/SSE3) instructions. It accelerates a broad range of applications, including video, speech, image, photo processing, encryption, financial, engineering, and scientific applications. Intel Advanced Digital Media Boost enables 128-bit instructions to be completely executed at a throughput rate of one per clock cycle, effectively doubling the speed of execution compared to previous generations.

Intel® Smart Memory Access

Intel Smart Memory Access improves system performance by optimizing the use of the available data bandwidth from the memory subsystem and hiding the latency of memory accesses. Intel Smart Memory Access includes an important new capability called "memory disambiguation," which increases the efficiency of out-of-order processing by providing the execution cores with the built-in intelligence to speculatively load data for instructions that are about to execute before all previous store instructions are executed.

Intel® Advanced Smart Cache

Intel Advanced Smart Cache is a multicore optimized cache that significantly reduces latency to frequently used data, thus improving performance and efficiency by increasing the probability that each execution core of a multicore processor can access data from a high-performance, more efficient cache subsystem. Intel Core microarchitecture shares the level 2 (L2) cache between the cores. This better optimizes cache resources by storing data in one place where each core can access it. By sharing L2 cache between each core, Intel Advanced Smart Cache allows each core to dynamically use up to 100% of available L2 cache.

Intel® Intelligent Power Capability

Intel Intelligent Power Capability is a set of capabilities for reducing power consumption and device design requirements. This feature manages the run-time power consumption of all the processor's execution cores. It includes an advanced power-gating capability that allows for an ultra fine-grained logic control that turns on individual processor logic sub-systems only if and when they are needed. Additionally, many buses and arrays are split so that data required in some modes of operation can be put in a low-power state when not needed.

For more details about Intel Core microarchitecture innovative ingredients visit http://softwarecommunity.intel.com/UserFiles/en-us/sma.pdf.

Intel Multicore Processors

Intel multicore technology provides new levels of energy-efficient performance, enabled by advanced parallel processing and next-generation hafnium-based 45nm technology. These processors contain multiple processor execution cores in a single package, which benefits SMP designs by providing full parallel execution of multiple software threads on cores that run at a lower frequency.

Intel Core Microarchitecture

Intel first introduced Intel Core microarchitecture in 2006 in the Intel Core 2 processor family manufactured with the 65nm silicon process technology. Intel Core microarchitecture extends the energy-efficient philosophy first delivered in Intel's mobile microarchitecture (Pentium M processor) and greatly enhances it with many leading-edge microarchitectural advancements, as well as some improvements on the best of Intel NetBurst microarchitecture. This new microarchitecture also enables a wide range of frequencies and thermal envelopes to satisfy different performance needs.

Intel Core microarchitecture is designed with shared L2 cache between cores. An SMP benefit of shared cache is that it provides improved performance (by reducing cache misses) when data is shared between threads running on cores that share cache. The VxWorks SMP operating system takes advantage

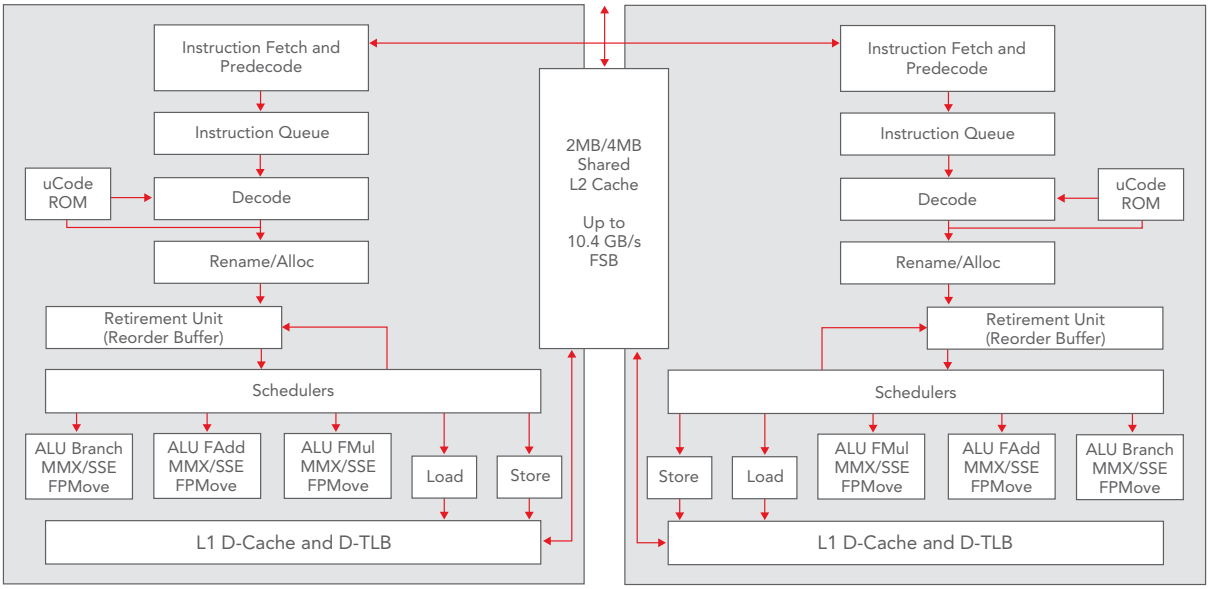

Figure 1: Intel Core microarchitecture, dual-core configuration

of shared cache benefits by providing the CPU affinity function, which allows the developer to map threads to specific cores and can improve performance on multiprocessor systems. Figure 1 depicts a dual-core configuration of the Intel Core microarchitecture.

Intel Core 2 Processor Family, Codenamed 'Penryn'

In the second half of 2007, Intel began production of the Intel Core 2 processor family codenamed "Penryn." The Penryn-based Intel Core 2 processor family is based on Intel's industry-leading 45nm high-k metal gate silicon technology and Intel's latest microarchitecture enhancements. This next evolution in Intel Core microarchitecture builds on the success of Intel's revolutionary microarchitecture (currently used in both the Intel Xeon and Intel Core 2 processor families). With more than 400 million transistors for dual-core processors and more than 800 million for quad-core, the 45nm Penryn family introduces new microarchitecture features for greater performance at a given frequency, up to 50% larger L2 caches (up to 12 MB), and expanded power management capabilities for new levels of energy efficiency. The Penryn family also includes nearly 50 new Intel Streaming SIMD Extensions 4 (Intel SSE4) instructions for speeding up the performance of media and high-performance computing applications.

Next-Generation Intel Core Microarchitecture, Codenamed 'Nehalem'

Coming in 2008 is the latest Intel Core microarchitecture codenamed "Nehalem." Nehalem is designed from the ground up to take advantage of hafnium-based Intel 45nm Hi-k metal gate silicon technology; and with its dynamic and design scalable microarchitecture, Nehalem can deliver both performance on demand and optimal price/performance/energy efficiency for each type of platform.

Nehalem delivers the following:

- Dynamically managed cores, threads, cache, interfaces, and power for energy-efficient performance on demand
- Simultaneous multithreading (SMT) that enables running two simultaneous threads per core
- Extensions to the Intel SSE4 that center on enhancing XML, string, and text processing performance
- Multilevel cache, including an inclusive shared L3 cache
- Memory bandwidth that delivers from two to three times more peak bandwidth and up to four times more realized bandwidth (depending on configuration) as compared to today's Intel Xeon processors
- Performance-enhanced dynamic power management

Intel QuickPath Technology

Nehalem will be the first to introduce Intel QuickPath technology, which is a new system architecture that unleashes performance with an interconnect system architecture that provides point-to-point high-speed links to distributed shared memory. Each processor core will feature an integrated memory controller and high-speed interconnect, linking processors and other components. Intel QuickPath technology delivers the following:

- Dynamically scalable interconnect bandwidth designed to set loose the full performance of Nehalem and future generations of Intel multicore processors
- Outstanding memory performance and flexibility to support leading memory technologies
- Tightly integrated interconnect reliability, availability, and serviceability (RAS) with design-scalable configurations for optimal balance of price, performance, and energy efficiency

Symmetric Multiprocessing Operating Systems

Besides being an application with an ample amount of built-in concurrency, the SMP operating system is probably the most important software factor that affects parallel processing capabilities. The following topics explain multicore software development.

A symmetric multiprocessing configuration (depicted in Figure 2) is where one operating system controls more than one identical processor (or core in a multicore processor). Applications "see" and interact with only one operating system, just as they do in single-processor systems. The fact that there are several processors in the system is a detail that the operating system hides from the user. In this sense, an SMP operating system abstracts the hardware details from the user. An SMP operating system is also symmetric—it needs to work with multiple identical processors, each of which can access all memory and devices in the system in a uniform fashion. Therefore, any processor (or core) in an SMP system is capable of executing any task just as well as any other processor in the system. This symmetry in the hardware allows an SMP operating system to dispatch any executing task to any processor in the system. An SMP operating system tries to keep all processors busy running application threads, in effect load balancing the system's work.

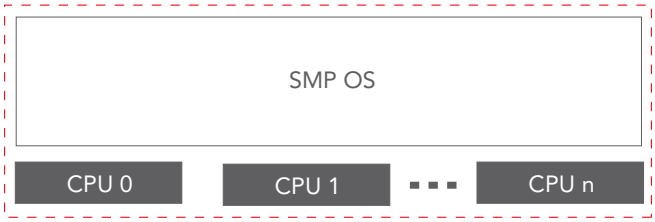

Figure 2: A symmetric multiprocessing OS configuration

As a result of hardware abstraction and load balancing in the system, the SMP operating system simplifies the task of developing software to run on SMP hardware. From the programmer's view, there is only one operating system to write an application for, which will automatically distribute the workload to all available processors. An SMP operating system hence provides the same programming semantics as a uniprocessor system.

All processors in an SMP system share one pool of memory under the control of one operating system. The memory and local caches for each individual processor are kept synchronized by the processor hardware. Therefore, applications in an SMP system can share data with each other in memory very easily and efficiently. This makes SMP systems suitable for applications that need to share large amounts of data with low latency.

Real-Time Capabilities of VxWorks SMP

Real-time is a very relative concept, dependent on the maximum response time that is permissible for the given application. If the application can accept a response time in seconds and the footprint is acceptable, even a desktop operating system might qualify as a real-time system, but only for that application, not every application. At the other extreme, something that needs to be controlled at nano-second speeds may not be suitable for any OS. Such a system may be better served by hardware control logic instead of a microprocessor running an operating system.

Any SMP kernel has to manage and synchronize more than one CPU, which makes its overhead necessarily higher than that for a uniprocessor kernel. Therefore, benchmark-for-benchmark, any SMP kernel will be slower than the corresponding kernel built for only one CPU. However, SMP wins out in its ability to parallelize execution and thus gain performance in ways that a single processor cannot match.

The real-time characteristics of VxWorks are preserved in the SMP case by kernel design choices that emphasize deterministic behavior of kernel operations and lowest possible latencies. This allows VxWorks SMP to be applicable in the same target markets and target applications as uniprocessor VxWorks systems. VxWorks SMP has a deterministic scheduler, which guarantees task execution in priority order. The scheduler dispatches the *N* highest priority tasks that are ready to run, where *N* is the number of processors in the system. Deterministic priority-based scheduling is also one of the critically differentiating features of VxWorks on single-processor systems; this is what differentiates a real-time operating system (RTOS) from a general-purpose operating system.

VxWorks SMP implements interrupt-level parallelism, namely the ability for more than one core in the system to handle different interrupts simultaneously. This increases the responsiveness of the system as a whole. Many critical operations within the OS are protected by spin-locks. These operations complete in deterministic time, which is a function only of the time for which the spin-lock is held. Spin-locks also protect critical data during interrupt processing. These spin-locks are held for a deterministic length of time, which in turn makes the system's interrupt latency deterministic as well.

Multicore by itself does not affect real-time behavior. If anything, multicore technology allows the device as a whole to do more and be more feature-rich and capable. But if the timing tolerances are especially tight, SMP may not be the right choice. An AMP system design is a much more suitable alternative because each core can run a uniprocessor operating system with the lowest possible latencies.

Multicore and SMP Software Considerations

Multicore requires software that exploits the presence of multiple processing units. It's important to understand your multicore system objectives and how your applications will be affected before starting the migration to multicore and developing a migration plan.

Depending on the objectives and design of the current software, the migration might be as easy as just porting your software to the multicore platform without changes, or it could require design changes of the application(s) as well as other changes:

- If the current applications are already designed for a multiprocessor system, they should already be capable of utilizing and benefiting from multicore hardware.
- If the current applications are designed for a uniprocessor platform, they will need to be surveyed and updated as needed to meet the migration plan objectives. This could require changes throughout the applications.

Improve Turnaround, Throughput, or Add Processing

Multicore processors improve software performance by using the additional cores to improve turnaround and/or throughput of data processing. For example, in a factory assembly line, the amount of time it takes to assemble one unit of the product is called turnaround time. The number of products produced per day or per month is called throughput. These two terms are not mutually exclusive; a reduction in turnaround time increases throughput. However, if increased throughput is the end goal, it may be achieved by increasing the turnaround time per unit and/or adding more assembly lines to operate in parallel. When considering how to optimize the performance of software applications, it is up to the developer to decide which of these performance improvements will be used at various points in the code. One area of processing might benefit from improved turnaround (i.e., a "divide and conquer" strategy), while another benefits from improved throughput (bandwidth):

- An imaging application might benefit from improved turnaround and would therefore use a four-core system to divide the image processing for each frame between the four cores. The code would create a minimum of as many threads as there are cores, and each thread would process the pixels for the portion of the image within its assigned region of the screen. Four cores could have four threads operating on a total of four regions (quadrants) of the screen image simultaneously and thus turnaround could theoretically be improved by a factor of four times. Visit http://download.intel.com/technology/advanced_comm/315697.pdf to read a case study about an open source medical imaging application that was threaded for multicore.
- A packet-processing application might benefit from improved throughput and would therefore use the four cores to theoretically process four times as many packets in the same amount of time as on a single core. Visit http://download.intel.com/technology/advanced_comm/31156601.pdf to read a case study about an open source intrusion detection program that achieved supralinear performance across four cores and the parallel programming IP packet processing methodologies that were used.

Regardless of whether an application needs more performance, it can also benefit from multicore by using the additional cores to support more processing, such as adding new features to applications or adding new applications that can run simultaneously with the others. Multicore technology also allows designers to consolidate systems previously built with many boards into one or a few boards, saving space, weight, and power in doing so.

Assess the Readiness of the Applications

As part of the migration plan, the applications should be surveyed for SMP readiness. Any software that is not ready for the requirements called out in the migration plan must be updated. Three application types that should be given particular attention when assessing for SMP include the following:

- **Applications that are serially coded and can multitask well together:** Applications that don't contend for hardware resources could be run on multicore platforms without changes. However, to get the best performance of the application on the multicore platform and to achieve the best scalability to future processors with increasing cores without additional code or configur-ation changes, the application should be redesigned to take advantage of the additional cores by decomposing the work within the hotspots of the code (areas that do the heaviest processing).
- **Libraries and device drivers:** All routines called concurrently from multiple threads must be thread-safe. Two methods are available to ensure thread safety:
  1. Routines can be written to be reentrant.
  2. Routines can use mutual exclusion synchronization to avoid conflicts with other threads. As far as possible, it is better to make a routine reentrant than to add synchronization objects because synchronization prevents parallelism.

  Update the libraries and device drivers that were written in-house, and check for SMP-safe versions from vendors. The operating system must support multiple processors, such as SMP operating systems. VxWorks SMP provides SMP capa-bilities to VxWorks. It leverages multicore processors to achieve true concurrent execution of applications, allowing applications to improve performance through parallelism. VxWorks SMP, when combined with Wind River Workbench, provides users with a complete VxWorks SMP–ready platform comprising the run-time, middleware, and Wind River's market-leading development tools suite.

- **Development tools:** Some software tools (and hardware tools) such as performance analyzers will need the capability to recognize and analyze the software that is running on all of the cores. From hardware and board initialization to device management, Wind River Workbench offers deep capabilities throughout the development process in a single integrated environment, with complete platform integration and tools for debugging, code analysis, advanced visualization, root-cause analysis, and test. The Wind River Workbench suite was upgraded as needed to support debugging and analyzing an SMP environment. Workbench 3.0 has the ability to debug the VxWorks SMP kernel and real-time processes (RTPs) in system or task mode. Workbench and in particular Wind River System Viewer enhancements simplify the diagnosis of race conditions and deadlocks, two common issues that arise when developing programs for multicore environments. All tools work identically with VxWorks SMP as they do with uniprocessor VxWorks.

Concurrency vs. Parallelism

Concurrency is having two or more threads in progress at the same time. Parallelism is having two or more threads actually executing at the same time. On a uniprocessor platform, only one thread can execute at any given time; any other threads in progress must wait. Although concurrency is achieved, parallelism is not possible. On a multi-CPU platform, these threads can execute simultaneously, thus providing parallelism.

The amount of parallelism that can be achieved on a multiple core system is limited by the level of concurrency that is designed in. The maximum theoretical speedup of a serial application that will be redesigned to run on multiple cores is restricted by the amount of code that must remain serial. This is known as Amdahl's law. All jobs include some work that has to be serial. Amdahl's law says serial processing limits speedup: `Parallel speedup = 1/(Serial% + (1-Serial%)/N)`.

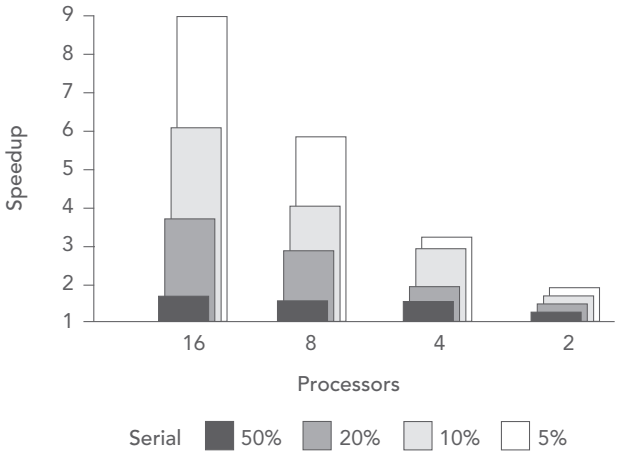

Figure 3: The theoretical maximum performance gains possible for a varying number of CPUs and percentage serialization

Threads have a performance overhead in their creation and termination. So beware that adding too many threads could actually degrade the efficiency of the processing. A good rule of thumb to follow is to have as many simultaneously active threads as the number of available cores. Any less would be considered undersubscribed and any more would be oversubscribed.

Migrating Applications to SMP

During the design stage of the multicore application, the developer will need to identify areas of high processing in the code and then decide how to decompose the work being performed in that code. The work will either be decomposed by data or by function:

- Data decomposition divides large datasets, whose elements can be computed independently, between the available cores. The same task operates on different data. This is typically found in "for" loops.
- Functional decomposition (i.e., task decomposition) divides the computation to the available cores based on the natural set of independent functions (tasks), different functions operating on the same data.

Locating Opportunities for Parallelization

The most important advantage offered by an SMP operating system is the opportunity to run code in parallel. Performance speedups on SMP originate from the ability to run more than one stream of code in parallel. Therefore parallelism is the key to achieving higher performance in SMP environments. System designers must find ways to maximize the system's ability to run as much code in parallel as possible if they are to get the most amount of performance possible in SMP environments.

Three factors limit the amount of parallelization and therefore the amount of performance gain that can possibly be achieved in an SMP system:

1. The amount of parallelization possible given the application algorithm; Amdahl's law illustrates how much speedup is theoretically possible for a given ration of serial-parallel execution
2. The number of OS synchronization calls used by the application
3. Excessive context switching

Parallelism Inherent in the Application

Applications have differing amounts of parallelism that are inherently possible by their nature and the design of the algorithm. More parallelism contributes to better utilization of multiple cores and therefore better performance in SMP operating systems. To better understand the issues involved consider a few real-life situations.

If there is a job to do, it can get done faster with many people working on it than with only one. Cores in a multicore processor are like more workers who can be applied to a job.

Obviously there must be enough work to keep all the workers busy. Then you have to be able to split the work into enough pieces to give every worker something to do. But it can also be similar to having too many cooks in the kitchen: The work can't get done faster just by assigning more workers. The workers have to work well together or the efficiencies that could be gained could be lost. The workers need to be kept busy, not waiting around for something to happen. The more time workers spend waiting for work to do, the less productivity and output from the kitchen.

Dependencies and Resource Contention

In the real world, assembly lines are an example of multi-processing. A lot of work can be accomplished by carefully coordinating multiple activities so that they happen in parallel as much as possible. Yet there are dependencies between the actual stages through which the work goes. If the dependencies are not right or the work is not done correctly, the subsequent steps stall and output suffers.

A delivery warehouse that sorts packages and loads them on to the right delivery truck is an example of an application with almost no dependencies. Each package is processed independently from every other package. This system has a very high level of parallelism. The challenge is that of shared resources. In this example, if there were too few carts available to move the packages around, every worker would be waiting. This situation is called resource contention.

Some tasks simply cannot get done faster by adding more people or resources to it. Reading a book, for example, is a strictly serial operation. Every chapter in the book must be read in order for the story to be understood. Adding more readers to read chapters in parallel would not help. This is an example of an application that cannot benefit from multiprocessing.

Computing applications have many of the same characteristics. There are applications that have inherently high degrees of parallelism possible. There are also applications, though not originally written to exploit parallel execution, that can nonetheless be changed to work in parallel. These applications and algorithms are ideal targets for running in SMP environments. On the other hand, there are applications that by their nature are very serial and gain little or nothing from running in multiprocessor environments.

Maximizing Application Parallelism

The following can maximize parallelism in the application:

1. Decompose the application into multiple threads or tasks that can be scheduled by the operating system in parallel. Although your design might use many more threads than available cores, remember that a good rule of thumb to follow is to have as many simultaneously active threads as the number of available cores. Otherwise the processor utilization is "oversubscribed."
2. Reduce dependencies between threads as much as possible. With many threads and processors in action, dependencies and synchronization points defeat parallelism and ultimately bring down the performance gains that would otherwise have been possible. Dependencies take the form of shared data variables or messages passed between threads that affect their execution.
3. Audit your application for resources contention. The "resources" in this case can be frequently used objects such as semaphores for synchronization, spin-locks, critical sections that must be single-threaded, and so on. The more frequently threads wait for access to these resources, the more time is spent waiting, and correspondingly less time is spent executing the application. Performance ultimately suffers. It is advisable to redesign the application to use such shared resources as sparingly as possible.
4. Operate on local, independent data sets as much as possible. This allows threads to operate with reduced or no synchronization with other threads. Operating on local data also maximizes the chances of finding the needed data in local cache, thus boosting performance.
5. Use lock-free algorithms. These are algorithms designed in such a way as to eliminate the need for access to shared data to be locked with semaphores. Part of this process may involve using dedicated threads for certain kinds of operations, thus precluding the possibility of other threads reentering the same critical section.

Minimize OS Synchronization Calls

OS synchronization calls limit the amount of parallel activities in the system. Synchronization calls need to enforce mutual exclusion, which serializes thread execution. Thus, synchronization calls can impact performance and should be used sparingly. Examples of synchronization calls are semaphore operations, spin-locks, calls to other interprocess communication (IPC) facilities, and context switches.

Excessive Context Switching

An SMP operating system by design will schedule threads on any available core because it tries to keep all cores in the system busy. Often this can lead to threads "migrating" from one core to another as the OS schedules them. Frequent core migration of threads leads to significant cache miss penalties when the thread begins running on a core that does not have its data in cache and reduced performance results. Frequent core migration is typically caused when there are many more threads ready to execute than there are cores available to run them.

VxWorks SMP APIs

The vast majority of VxWorks APIs work identically between VxWorks uniprocessor and SMP environments. Many new APIs have been added and are available in the VxWorks uniprocessor to perform the same as SMP systems. These are the following:

1. Spin-locks, for mutual exclusion, are based on instruction set primitives such as test-and-set instructions. Spin-locks are best suited for short-duration critical sections (about 10 to 20 lines of code is a rule of thumb). A spin-lock protects the locked critical section from being accessed by multiple processors. Other processors spin while they wait to acquire the locks. Hence spin-locks should be used for brief durations. Longer-duration critical sections should be protected with semaphores.
2. Atomic operators are a completely new class of APIs that operate on single variables in memory. They atomically operate on variables to perform arithmetic and logical operations. These operations are atomic with respect to other processors in the system. The short code segment in the following table illustrates a common use of an atomic operator to increment a global counter.

Without Atomic Operators	With Atomic Operators

`level = intLock ();`
`++globalCounter;`
`intUnlock (level);`
	`vxAtomicInc (&globalCounter);`

3. A reader-writer semaphore is a new type of semaphore in VxWorks 6.6 (available in both uniprocessor and SMP). A reader-writer semaphore allows many reader threads to gain fast read access on a shared data structure. Only one writer thread is allowed to modify the data structure at a time. These semaphores are optimized for situations having multiple readers and one writer. Their use is encouraged in producer-consumer-type software algorithms and is especially suited for multiprocessing systems.
4. Thread-local storage is now implemented using a multiprocessor-safe compiler feature and the thread storage class to manage thread-local variable storage. It supersedes the legacy VxWorks taskVarLib facility in kernel mode and the tlsLib facility in user mode, neither of which is available in an SMP environment.

Writing Thread-Safe Code

Thread-safe code is code that is able to work correctly and coincide with other threads running in the same address space. Thread safety becomes especially more important in SMP environments, where it is routine to have several threads running concurrently. For performance reasons, it is advantageous to have multithreaded software share data. But the sharing of data must be regulated with protected mutually exclusive access to that data. In single-processor environments, programmers can often make assumptions about concurrency based on the fact that there is only one processor and on other OS factors such as priority-based scheduling. It is common to find software written for single-processor systems making such assumptions that are never violated. Effectively these are shortcuts that work because there is only one processor in the system. In a multiprocessor system, single-threaded code execution assumptions break down. Shared data between threads must be protected explicitly or else software no longer works correctly. Writing thread-safe code involves the following:

- **Having fewer or ideally no global variables at all:** Every global is a shared data item and access should be mutually exclusive for correct operation. Locks to enforce mutual exclusion increase serialization and reduce performance by making threads wait for locks. Use only as many global variables as you must.
- **Adding mutual exclusion locking around shared data items and critical sections that operate on them:** Such protections could be avoided in several carefully analyzed code paths when only a single processor is involved but become mandatory with SMP.
- **Making no assumptions about execution order or concurrency in the system (e.g., thread priority–based concurrency assumptions):** Often such assumptions are used to increase performance by dispensing with correct protections on shared data items.
- **Having proper synchronizations around shared data access between task and interrupt level:** This is most often found in drivers. VxWorks on a single-processor system always guarantees that interrupt handling and task execution are mutually exclusive. This could be used to avoid protection on variables shared between tasks and interrupt service routines (ISRs). On VxWorks SMP, however, task execution and interrupt handling may execute concurrently. Adding appropriate protections around shared variables (typically they tend to be global variables) is mandatory in an SMP environment.

Tools for SMP Development

Contrary to popular belief, there is nothing inherently different between SMP and uniprocessor systems when it comes to most debugging and tuning workflows. One unique SMP requirement is ensuring that the maximum throughput and CPU utilization is achieved. For this specific problem, the system visualization tools help users understand CPU utilization and concurrency interactions.

The bugs and problems that a developer will see in an SMP system are the same kind of problems that appear in uniprocessor multithreaded systems. Some kinds of bugs are more likely to occur in an SMP system than a uniprocessor system, especially if the software makes certain assumptions about concurrency, but they are not strictly SMP issues. Debugging SMP is not inherently different than uniprocessor systems. That said, here is some guidance to problems that developers are more likely to see in SMP systems and how to find them:

- Problems arising from unprotected or incompletely protected critical sections, such as race conditions and memory corruption, are more likely to occur when threads are running concurrently. This happens because concurrently running software is far more likely to execute such erroneous code than software that runs with little or no concurrency. Memory profiling and System Viewer help catch these errors, same as in uniprocessor systems.
- Priority inversion, for example, a low-priority thread inherits priority over a higher-priority thread waiting on a resource, could be caused when other threads make assumptions about thread priority. System Viewer can help visualize this just as on uniprocessor systems.
- Task-interrupt concurrency issues result from handling interrupts on one core while another core is running a task. Legacy VxWorks applications could assume that task and interrupt processing are mutually exclusive, which is not true in SMP. System Viewer can help visualize this just as on uniprocessor systems.

In general, all of the common issues that the run-time analysis tools let you observe and understand work equally well with uniprocessor and SMP systems. It is just as easy to use the tools in an SMP environment.

Wind River SMP-Capable Tools

The Wind River Workbench suite has been upgraded to support debugging and analyzing an SMP environment. Workbench 3.0 has the ability to debug the VxWorks SMP kernel and RTPs in both system or task mode. Workbench enhancements simplify the diagnosis of race conditions and deadlocks, two common issues that arise when developing programs for multicore environments. The Wind River Debugger, System Viewer, Performance Profiler, Code Coverage Analyzer, Memory Analyzer, Data Monitor, and Function Tracer were updated so they can be used equally well on SMP as on uniprocessor systems.

Wind River System Viewer

Wind River System Viewer contains new analysis capabilities to aid SMP kernel and application developers. For SMP the System Viewer event graph is annotated with information providing core awareness to help developers identify and isolate race conditions, deadlocks, and starvation in the same way they would on a uniprocessor system. This can be useful to minimize performance losses. Using System Viewer, developers can see parallel activity going on in the system: task-level parallelism, task-interrupt parallelism, and interrupt-interrupt parallelism, all synchronized to a common time source for timing accuracy. This can be useful to visualize the execution characteristics of the application and aids in diagnosing software defects arising from concurrent execution. In addition, System Viewer shows the level of CPU loading on every core in the system. This is useful for performance tuning and effective load balancing across all cores in the system.

VxWorks Simulator

VxWorks Simulator can simulate SMP applications on single-CPU or multi-CPU host machines. Using Simulator, users can perform basic to advanced levels of application porting, simulation, and characterization before moving to real multicore hardware. System Viewer is also available on the host machine to enable the user to collect and visualize SMP data even when run using Simulator. Note, however, that while Simulator provides an accurate view of SMP application behavior, it cannot be used to provide accurate estimates of SMP application performance because the underlying host hardware is likely to be significantly different from the desired target hardware.

Wind River Analysis Tools

Wind River Run-Time Analysis Tools (formerly known as ScopeTools) work with VxWorks SMP just as they do on the uniprocessor version of VxWorks. These tools provide capabilities for code profiling, data monitoring, memory usage, and leak detection.

Hardware Debug Tools

Wind River's JTAG-based on-chip debugging solution is useful for hardware bring-up on through OS-aware application debugging. As an OS debugger, Wind River ICE is aware of the VxWorks kernel environment and symbol table. It is able to debug at task level through the JTAG interface. On hitting breakpoints, all cores are stopped, giving the user a view of program execution on all cores. This is useful in debugging concurrency-related issues that depend on the effects of code executing on other cores.

Optimizing Software for SMP Environments

For software to perform optimally in an SMP system, it must exploit parallelism in as many ways as possible, making sure that the CPUs are busy executing the application as much as possible. Parallelism is the key to getting the most performance out of an SMP operating system.

Use every possible approach to promote parallelism in the execution of the system. The CPUs in the system should be kept as busy as possible doing useful work. Applications should be decomposed into threads that can execute in parallel as much as possible. As a note of caution, it is not advantageous to have too many more threads than there are CPUs in the system, or context-switch time will become an increasingly significant factor in reducing performance. Lock-free algorithms are very beneficial whenever appropriate. The absence of a lock promotes parallelism, which enhances performance with SMP. Getting the best performance from any SMP operating system involves the following three approaches.

Use Lightweight Synchronization Primitives

Every OS synchronization call asks the kernel to enforce mutual exclusion, which requires other CPUs in the system to wait until an operation is completed. Since SMP kernels have to manage more than one CPU, their overhead for the same operation is higher than a uniprocessor kernel. When synchronization is required, it is suggested to use the lightest-weight synchronization primitive that is right for the job at hand. Single variable accesses can be atomically performed using atomic operators. Spin-locks are encouraged for short-duration critical sections (typically 10 to 20 lines of code). Longer duration critical sections are best protected using semaphores. Spin-locks are not the best choice in this case, as other CPUs will be forced to spin (i.e., do no useful work) while waiting for the lock. Reader-writer semaphores are ideally suited for producer-consumer type algorithms. Mutex semaphores should be used only when really necessary because mutexes allow no more than one task at a time to access the critical section, which does not promote concurrency in SMP environments.

Use Task-CPU Affinity Judiciously to Tune Performance

Task-CPU affinity is a powerful though double-edged tool to tune performance on SMP systems. Setting affinity on a task locks it to running only on an assigned core. Henceforth, the kernel will only run that task on that core. The following code example illustrates how to set the affinity for a task to CPU1:

```
cpuset_t  affinity;

/* Clear the affinity CPU set and set index
for CPU 1 */

CPUSET_ZERO (affinity);
CPUSET_SET (affinity, 1);

if (taskCpuAffinitySet (taskId, affinity) ==
ERROR)

  {

  /* Oops, looks like we're running on a
  uniprocessor */
  return (ERROR);

  }
```

The following are the benefits of using affinity for performance-critical tasks in the system:

1. Optimized cache locality for the task's data; in other words, the task's data has the highest chance of being located in local cache. This reduces the number of cache-miss penalties it encounters, which contributes to application performance. Without affinity, the kernel may schedule tasks on other CPUs to balance system load. A good rule of thumb is to run threads that share data on cores that share cache.
2. Higher performance due to lower bus contention, again due to the locality of the tasks data.
3. Deterministic CPU bandwidth allocation for performance-critical tasks, which can have a more assured share of CPU time for the core the tasks run on.

Task-CPU affinity can also be used to migrate legacy code that is not yet safe to run in an SMP environment. This is code that still uses locking primitives (such as the intLock or taskLock APIs) that are suitable for and only available on uniprocessor environments. Non-SMP-ready applications can experience single-processor system semantics by confining them to one core in the system. Thus they can continue working on an SMP operating system; though because of affinity they will not be able to run concurrently. Task-CPU affinity when used this way is a migration aid not a performance enhancement.

Minimizing Thread Load Imbalances

Errors and inefficiencies can be brought on by concurrent execution in the system: from task-level parallelism, task-interrupt parallelism, or interrupt-interrupt parallelism. System visualization tools can be used to discover the timing and type of activity and the loads on each core. This is useful for performance tuning and effective load balancing across all cores in the system. Core loading data can be used to set task-CPU affinity for those tasks that are critical or that tend to move from one CPU to another. It is also possible to see in the execution event logs which resources have the most contention (i.e., those that frequently have threads pending on them). Resource contention can be reduced or eliminated by modifying the algorithm to remove over-reliance on frequently used resources such as semaphores and message queues.

Conclusion

As the embedded device market continues to trend toward smaller form factors, standardization, and improved performance per watt, Intel has responded by producing leading-edge microarchitectural advancements and processor innovations in its multicore product lines. The end result is that Intel is increasingly meeting the needs of embedded market segments and has become a pervasive embedded processor.

Wind River's VxWorks SMP operating system harnesses the power of multicore by scheduling the processes to be executed among all of the cores and also provides APIs that are optimized for multicore. Wind River has simplified the migration from VxWorks uniprocessor systems to VxWorks SMP operating systems because the vast majority of VxWorks APIs work identically between both environments. Additionally, the Wind River Workbench suite provides several software development tools that help the developer throughout SMP software development.

For embedded software developers, multicore provides an exciting new era of design possibilities and opportunities to extend the performance and capabilities of their software systems. Multicore processor technology is here. The time to get started with designing your software for multicore is now, and Intel and Wind River solutions guide the way.

WIND RIVER

Wind River is the global leader in Device Software Optimization (DSO). We enable companies to develop, run, and manage device software faster, better, at lower cost, and more reliably. www.windriver.com

intel
Embedded and Communications Alliance