FUNDAMENTALS OF MULTICORE: Operating Systems for Telecom/Networking Applications

Michael Christofferson
Director of Product Marketing, Networking

This white paper provides a “tutorial like” overview of the fundamental issues that apply to multicore software operating systems implementation for the telecom/networking space. It starts with consideration of the different generic application processing classes in telecom/networking (e.g. control plane, data plane). It then considers the mapping of these processing requirements to the two primary software processing models for multicore (e.g. SMP, AMP, etc).

Following this rather abstract treatment, real world implementation issues are addressed including the role of hypervisors and virtualization, as well as a single OS environment for both control and data plane in multicore devices. Lastly some future topics such as the a) future of multicore operating systems, b) advanced load Balancing concepts, and c) advanced IPC (inter-processor communications) in multicore devices are touched upon, and that will be topics for advanced treatment in future Enea white papers. Enea software solutions for all the above are briefly referenced, but are not the main thrust of this white paper. So this paper is ideal for those who wish to understand the real foundational issues that affect effective multicore software design. But this paper also has enough advanced concept ideas in the concluding sections to interest those who are already fairly familiar with multicore software concepts and its challenges.

Why Multicore? There is no question that most of the latest innovations in processor architectures have been focused on multicore processor architectures from virtually all of the major silicon suppliers. Why?

General motivation for Multicore
The new interest in multicore devices is primarily driven by physics – faced with the growing (non-linear) energy consumption and excessive operating temperatures caused by high CPU clock speeds, microprocessor vendors have adopted a new approach to boosting system performance by integrating multiple independent processor cores on a single chip, reducing overall cost and power per delivered MIPS. The immediate benefits of multi-processing on a single integrated chip/device are and could not be obvious: applications that were designed around multiple uniprocessor configurations will be or cannot and can now run on multiple core processors with lower CPU hardware and footprint costs. The computing power of these configurations increases dramatically with no appreciable change in their physical configuration. Multicore processors like these are poised to bring new levels of performance and scalability to networking equipment, control systems, and a host of other embedded applications.

However, this rosy picture of the future benefits of multicore computing devices primarily derives from hardware and system considerations – but in reality, the potential benefits of multicore architectures are wasted unless software, operating system, applications, and debuggers and other tools, can utilize multicore devices properly. These software considerations are the primary thrust of this white paper.

Homogeneous or Heterogeneous Multicore?
There are of course two different implementations of an integrated multicore device – homogeneous cores and heterogeneous cores. Homogeneous cores, featuring identical CPUs on all the cores, are of the most interest in this paper, but heterogeneous cores are usually a mix of general purpose CPU architectures and DSP architectures and are also important, but as a specific case of AMP based software architectures. This paper will focus on the homogeneous multicore cases.
Generic Telecom/Networking Processing Classes
This section will consider the general classes of processing applications in telecom/networking. First we shall establish a set of criteria by which a processing class may be evaluated for applicability to a multicore implementation.

General Criteria
Before getting into actual use cases for multicore as a solution for specific types of applications, it is useful to establish some general criteria for evaluating the effectiveness of any given solution. Here we establish 4 general criteria for such: a) Configurability, b) Portability, c) Scalability, and d) Performance.

There are relationships and couplings between many of these, but these distinctions are still useful.

1. Configurability
When considering a multicore implementation for any system or application it is important to understand the configuration issues. For example, how easy is it to set up the system for a particular device? How easy is it to move to a new configuration (more cores) without requiring a complete overhaul of the system setup – boot, initialization/startup? What are the operating system issues involved in this?

2. Portability
Portability refers primarily to applications portability, but also to some extent involves the operating system. For example, how easily can an application move from uni-processor environment to a multicore environment? Are there OS portability considerations as well?

3. Scalability
Scalability involves consideration of moving to an ever larger number of cores – important as silicon technology progresses to ever higher performance devices by adding more cores. Some issues:
   a. Can more cores be added while maintaining performance? To maintain performance, core interlocking must be kept on a low level, and one must be careful that the various system busses do not congest. Congestion is not just a hardware consideration – most multicore hardware designs feature various mechanisms for helping with this – but the software application itself in most instances probably determines this. So care must be taken to avoid making any erroneous assumptions about any cross-thread or cross-application dependencies if those threads or applications may run on multiple cores simultaneously.
   b. Can more cores be added or removed without causing OS configuration changes? How will OS footprint grow with number of cores?
   c. Can more cores be added/removed without causing application redesign or configuration changes? This is related to portability, but really about portability from n to m cores, not just 1 to many.
   d. Will application performance scale with the number of cores? This is a really interesting issue and one should not just assume that adding more cores boosts performance in a linear or commensurate manner. It also involves OS considerations. But perhaps more often, it involves the application itself. See point “a” above. Here Amdahl’s Law can offer some guidance, Amdahl’s Law is a model for the relationship between the expected speedup of parallelized implementations of an serial algorithm. Reference figure 1 to the left. The parallel portion is the part of the existing application that can be optimized for parallel execution. The conclusion is that not all programs will benefit on parallel execution and that the effort/cost for some programs could be way too high.
   e. How will overall memory consumption grow with the number of cores? Again this is an important issue and it has a strong dependency on the particular processing model used for the application – SMP (Symmetric Multiprocessing) or AMP

![Amdahl's Law](image)
4. Performance
Perhaps at the heart of the whole multicore issue, is performance. Why move to multicore with all the potential headaches if the theoretically available bandwidth provided by the hardware cannot be effectively utilized? The major considerations here are:
   a. How much of the CPU can be utilized?
   b. How many parallel I/O channels can be handled?
   c. How predictable are the operations?

The Two Classes of Processing Characteristics in Telecom/Networking
Let's look at a couple of processing/programming classes that are prevalent in telecom infrastructure, and see how multicore technology maps to the criteria established above as a solution for that class. There are two main classes – CPU/execution bound processing and I/O bound or data transfer processing.

CPU/Execution Bound Processing
This class concerns a processing (CPU utilization) intensive application that could benefit from having parallel execution units as realized by a unified multicore processor. The idea is that if some sort of parallelism is achieved, then the application execution should conclude more quickly – better performance. So let's see how this processing class maps against the established criteria:

1. Configuration
   For CPU bound processing class, no re-configuration of the application environment, often related to the OS, is desirable, but not necessarily required. The idea is that the application environment (OS) should ensure that the application setup does not need to change as a result of finding itself in a multicore device.

2. Portability
   The application should not need to be changed for execution in a multicore environment. HOWEVER, as will be seen later, the issue of "concurrency" – multiple parts of the application that in a multicore implementation run in parallel – interferes with the ability of many CPU bound applications to run effectively in a multicore environment. In other words, not every application may benefit from "parallel execution" without having to be modified or even re-structured with parallelism in mind. But portability is a paramount consideration for the CPU bound processing class.

3. Scalability
   The application should be able to be scaled for execution across any number of cores, with more cores naturally leading to better performance. If performance scalability is not a specific issue, then the issues of configurability and portability become the primary drivers for multicore adoption.

4. Performance
   In this application, the emphasis is on maximizing CPU cycles rather than I/O, and therefore performance is the most important consideration for CPU bound applications. Therefore interrupt and device or I/O handling is not a characteristic of this type of application. But even so, real-time behavior and predictability is absolutely required since in compute bound applications it is not acceptable to have "unbounded" and therefore "non-deterministic" execution times for a given processing instance. Finally, load balancing across the cores, i.e. the ability to make maximum utilization of the available cores is absolutely required, otherwise having additional processing units running in parallel with each other is pointless. Load balancing may sometimes be achieved by static portioning of applications, if the input load may be completely quantified by the designers, but the trend in modern networking implementations is that dynamic load balancing will be required.

Control Plane Characteristics

   - CPU bound processing
   - Operations and Maintenance functions
   - Typically terminates IP traffic

Figure 2. Control Plane Characteristics.

In summary, for the CPU/Execution bound processing class, the most important considerations are performance of the single application and portability of that application. Configuration and scalability are less important. These characteristics map almost entirely to the profile of "Control Plane Characteristics" as depicted above in Figure 2.

I/O Bound or Data Plane Processing
This class is less about application performance and portability than it is about scalability and configuration, in contrast with the CPU/Execution bound class. But as we shall see below, best performance comes from the acceptance that configurability and scalability are addressed properly. The primary idea is not specifically to increase performance of the code processing a single data stream or ‘session’ like for the CPU/Execution bound processing class, but to maximize the number of streams or IP flows that can be processed with a lower number of CPU's and therefore BoM (Bill of Material) costs. In other words for telecom, the goal is a higher number of lines/sessions/subscribers per unit cost and unit power consumption, both of which are important to operators.

1. Configurability
   For I/O bound processing class, configuration is a paramount issue for multicore solutions. The whole idea is to maximize the number of
data streams or sessions for the device. This means that configuration of the number of sessions should NOT depend on number of cores, otherwise the cost effectiveness of such a solution from a "non-recurring engineering" or development cost point of view is negated. Time to market is thus affected adversely.

2. Portability
Portability of the application is less of a consideration for I/O processing flows. Some allowances for modification of the application so that multiple instances of it might run on multiple cores are acceptable, given the desirability to maximize the number of instances that might run, which directly drives the BoM issue.

3. Scalability
Scalability to any number of cores is a primary consideration for I/O bound processing class. Once the I/O bound application has been adapted to a multi-execution environment, then cost effectiveness and adaptability to ever increasing number of cores as dictated by silicon multicore evolution, becomes a paramount consideration. Here as indicated by the established criteria, memory consumption becomes a serious issue, because memory consumption directly affects the BoM costs that make multicore so attractive to begin with for this type of processing class.

4. Performance
In this application, static partitioning of applications bound to specific streams or threads seems to be the best model. An I/O bound application needs to be as deterministic as possible to guarantee maximum performance and is more affected by interrupts and device handling issues. Therefore in this processing class, it has so far become the case that it is better to lock data streams or user "sessions" to specific cores, wherein the ability to distribute these "locked" sessions over a large and easily expandable (see scalability above) number of cores is the key to best overall throughput and therefore maximum "cost per line/session/stream". But the penalty here is that this means that the "user" needs to manually distribute all these sessions across the cores so that the overall processing bandwidth of the multicore device is maximized – meaning that CPU utilization of each core needs to be maximized, but that the user is responsible for this, NOT the operating environment or operating system. However, there is currently much interest in finding methods that can actually "load balance" such applications across multiple cores. Some aspects of this will be addressed in the Conclusion in the end of the whitepaper.

Data Plane Characteristics
Figure 3. Data Plane Characteristics.

- I/O bound processing
- Highly optimized to use as few CPU cycles as possible
- Typically do not terminate IP traffic

So the bottom line for I/O bound or data plane processing applications (Figure 3 above) is that configurability and scalability are more important considerations than portability or performance of the single application. Performance is of course important, but for this class, is addressed more directly by adding more cores (scalability) and by reducing costs for doing so (configurability) than by insisting that the single application itself for a single processing thread, perform better in a multicore environment.

Homogeneous Multicore Software Processing Models
As mentioned above, the main thrust of this paper is homogeneous multicore models from a software perspective. This section will investigate different software models for support of homogeneous multicore devices. And this will then provide direct input into the analysis of the next section, where the software models are mapped for best fit to a) the variety of telecom market applications and b) with consideration of those applications with respect to the general processing models described in above.

One interesting aspect of the following discussion involves the problems presented by a shared memory model used by applications. It will be seen that a "message passing" model will overcome most if not all of these problems.

SMP – Symmetric Multiprocessing
Allocating resources in a multicore design can be difficult, especially when multiple software components are unaware of how other components are employing 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 allocate resources on the multiple cores with little or no input from the application designer.

By running only one copy of the OS, SMP can dynamically allocate resources to specific applications rather than to CPU cores, thereby enabling greater utilization of available hardware. It also lets system tracing tools gather operating statistics and application interactions for the multicore chip as a whole, giving developers valuable insight into how to optimize and debug applications.

Properly implemented, an SMP-enabled OS offers these benefits without forcing the developer to use specialized APIs or programming languages. In fact, developers have successfully used the POSIX standard (specifically the pthreads API) for many years in high-end SMP environments. But in practice, these implementations have only been proven for but a small number of processors or cores (somewhere I 4-8 at most).
A well-designed SMP OS allows the threads of execution within an application to run concurrently on any core. This concurrency makes the entire compute power of the 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.

However, in real practice, SMP works best on purely “CPU/Execution bound” processing models. Whenever there is the slightest I/O bound behavior intermixed within an application, then the real SMP performance breaks down. In other words, I/O delays tend to destroy the “supposed linear boost” of SMP performance across multi cores. So in practice, the SMP model does NOT scale well above 4 cores at best except for “pure processing intensive” applications. This is because SMP is designed to exploit potential parallelism in software by load balancing, but specific I/O operations against streams of incoming data are not parallel in time, they are parallel in “space”, i.e. requiring completely separate instances of an application for each data flow that individually are not subject to “parallel operation”. Even so for potentially parallelized control applications, industry data has shown that the SMP performance boost is not linear with the number of cores. With one OS for the entire device in the SMP model, internal kernel spin locks are required to successfully manage dynamic load balancing of threads, and this overhead increases geometrically with the number of cores. With OS overhead increasing geometrically, then raw performance of applications increases less than linear as the number of cores increases.

Inter-core IPC in SMP

Because a single OS controls every core in an SMP system, all inter-process communication (IPC) between cores is considered “local”. And this improves performance immensely, as the system doesn’t need a special IPC protocol to provide communications between applications running on different cores. So there is no special requirement on the IPC (i.e. message passing, queues, other) for IPC in an SMP environment to work effectively. However, an OS that supports seamless IPC for both intra-OS (on the multicore device) and external communications (to other domains or processors, etc) is a superior solution.

An IPC that has the same transparent APIs both internal to SMP applications and for external communications means that the programming model for communications is simplified – it allows any SMP application to communicate with any other application whether part of the SMP environment or external to it.

Allocation of resources in SMP - the concurrency problem

In an SMP system, all resources, including memory are shared and managed by the OS. However, if applications use a shared memory access model, then by definition there must be protective mechanisms built in to ensure that different parts of the application cannot simultaneously access these shared resources. This is the primary problem that is at the root of what is called the “concurrency” problem in multicore. Normally applications designed for a uni-processor model do not have to be concerned with simultaneous accesses to shared resources, since no two threads execute simultaneously. Even so there are well understood scenarios in uni-processor designs wherein there is the prospect of “collison” between two threads accessing shared data induced by the existence of a multi-threading or multi-tasking operating systems. In these cases designers use OS supported mutexes or semaphores that manage access to shared data or resources to prevent shared resource corruption. But in a multicore SMP scenario, even these protections break down since they only ensure that simultaneous access can occur as a result of the multi-threading OS. In an SMP model parts of the application may simultaneously access shared resources without regard to the OS scheduling scenario, which means that additional semaphores or mutexes must be introduced into the application code to ensure that all threads or tasks are “safe”. There are two conclusions to be drawn from this:

1. The SMP model absolutely dictates that all uni-core applications be reviewed to ensure that proper protections are in place for all threads that use shared data or resources.

2. A shared memory model that is often so efficient in uni-processor designs is NOT necessarily efficient in multicore designs because it may require additional collision protection in the form of mutexes, semaphores, or “spin locks”. And these additional locking mechanisms actually mitigate the whole SMP value proposition of maximum CPU utilization for all cores. Spin Locks inside the OS kernel and/or it’s services that are required to guarantee system data integrity are not “interruptable” and therefore contribute directly to the overall OS CPU overhead that directly reduces the overall CPU bandwith for applications. And it violates the concept of “code portability” that is one of the primary motivations for CPU/Execution bound processing that was mentioned above.

In conclusion, applications that by their design have heavy reliance on shared memory processing models will almost certainly have significant concurrency issues that must be dealt with by instituting various “locking” mechanisms. These locking mechanisms contribute to an overall decrease in maximum CPU utilization among multiple parallel execution cores. And this directly contributes to a lack of scalability of performance of SMP solutions to larger number of cores, for which performance scalability depends on having maximum CPU utilization per core at all times. As will be shown later below, a message passing model for process/thread coordination and synchronization will avoid most of these concurrency problems. To quote an MIT Computer Science and Intelligence Lab study (An Operating System for Multicore and Clouds – Mechanisms and Implementation).
“There are many problems with locks, such as choosing correct lock granularity for performance, reasoning about correctness, and deadlock prevention. Ultimately, programming efficient large-scale lock-based OS code is difficult and error prone”

**AMP – Asymmetric Multiprocessing**

Asymmetric multiprocessing, or AMP, provides an execution environment similar to that of conventional uni-processor systems, which most developers already know and understand. Consequently, it offers a relatively straightforward path for porting legacy code. It also provides a direct mechanism for controlling how each CPU core is used and, in most cases, allows developers to work with standard debugging tools and techniques.

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 (like Linux and an RTOS) or a different version of the same OS. In a homogeneous environment, developers can make best use of the multiple cores by choosing an OS that natively supports a distributed programming model (like Enea’s OSE). In a heterogeneous OS environment the same principle applies – choose a solution wherein a common distributed programming model applies to each of the selected OS’s: Enea LINX for Linux (available as open source) and the Enea OSE RTOS is a very good example of having a common distributed programming model between two OS’s. For a complete description of LINX for Linux go to http://sourceforge.net/projects/linx for a complete description of the cross-platform and portable LINX distributed message passing system.

In general, the AMP model works better for I/O bound processing, because it can scale to more cores better. If I/O rates are fairly uniform and quantifiable, then the designer knows how to partition the cores to achieve maximum CPU utilization of each core. If not, then some sort of load balancing mechanism is needed or at least desired.

Properly implemented, the AMP model will allow applications running on one core to communicate transparently with applications and system services (device drivers, protocol stacks, etc.) on other cores, but without the high CPU utilization imposed by traditional forms of inter-processor communication.

In virtually all cases, OS support for a lean and easy-to-use IPC communications protocol will greatly enhance core-to-core operation. In particular, an OS built with the distributed programming paradigm in mind can take greater advantage of the parallelism provided by the multiple cores.

**Inter-core IPC in AMP**

In contrast to the SMP requirements for IPC, since AMP uses multiple OS instantiations, it typically requires a complete networking infrastructure to support communications between applications running on different cores. To implement the lowest level of buffer transfer, an AMP system may use I/O peripherals (for instance, Ethernet ports) assigned to each processor/core, or better they would use a shared memory/interrupt-based scheme, or special HW assisted inter-core messaging (queues) acceleration engines, depending upon the system’s hardware capabilities.

The best IPC model though is a model that seamlessly supports a) inter-process/thread communication on the same core/OS domain, b) inter-core communications within a multicore device, and c) inter-device communications with other CPUs in the networked system. This means that applications regardless of location may directly communicate with any other application in the larger connected network, not just intra-multicore device. And this further means that the general IPC mechanism should NOT be exclusively bound to a specific interconnect internal to the multicore device. And to achieve this, the communications protocol must be transparent. At the API level, the communications destination and the interconnect for that destination should not be visible.

With the transparent message passing approach, local and remote communications become one and the same: an application can use the same code to communicate with another application, regardless of whether the other application is on the local CPU core or on another core, or even outside the device. Likewise, an application can access a device driver on a non-local processor/core as if that driver were running locally — this location transparency provides applications with easy access to hardware resources “owned” by other cores or processors.

**Allocation of resources in AMP**

With AMP, the application designer has the power to decide how the shared hardware resources (devices and/or shared memory) used by applications are divided up between the cores. Normally resource allocation occurs statically during boot time and includes physical memory allocation, peripheral usage, and interrupt handling. While the system could allocate or access the resources dynamically, doing so entails complex coordination between the cores. And this is a significant problem in a pure multicore AMP scenario. But it is one that can be addressed by the techniques described in the section about Multicore Implementations for Telecom/Networking Applications below.

In the AMP scenario, the problem of “concurrency” that is common to SMP systems is non-existent within a single application. It is only a problem if shared memory or resources situations exist between multiple independent applications on different cores. But in AMP, this is usually well understood by the designers and therefore is easily solved without undue special attention to the applications.

In an AMP system, a process will always be scheduled to run on the same core, even when that core is experiencing maximum CPU usage and other cores are running idle. As a result, one core or more cores can end up being over- or underutilized. To address the problem, the system could allow
applications to migrate dynamically from core to another. But the current “state of the art” operating environments for multicore do not support this “SMP like” behavior well, if at all. More advanced load balancing concepts are needed.

Hybrid Multicore models
After one gets to more than two cores in a homogenous multicore device, an interesting new model appears. What if the cores could be segmented between both an SMP model and an AMP model? The benefits are immediately obvious – applications that are more suitable for SMP can run in conjunction with applications that are more suitable for AMP and in the same single device that effectively could conceivably replace a whole multi-processor system. This will save BoM (cost), power and heat dissipation, and still provide the same or better performance. This is a powerful notion that is just now being explored by the industry because it makes sense. There shall be more discussion on this very topic in the next section, where the multicore processing models are mapped against practical implementations for telecom applications.

Multicore Implementations for Telecom/Networking Applications
From the above rather abstract analysis, we have found that there are two generic classes of processing types for telecom/networking – control plane processing that is CPU intensive and data plane processing that is I/O intensive. And further that these classes map almost directly to two processing models for multicore devices – SMP for control – CPU intensive applications, and AMP for data-I/O intensive applications. But now we must consider the actual implementation issues that result from this analysis. Figure 4 highlights this notion with some real world examples of processing in the IP packet domain – i.e. exactly where SMP and AMP models tend to dominate the solution space. So the choice of model should be clear for many applications. So are SMP solutions and AMP solutions forever bound to be separate solutions on a single multicore device? The good answer is: NO!

The most significant aspect of interest in multicore technology in telecom/networking is that it offers the possibility of consolidation of both the control plane and data plane aspects of a system into one device. This means that all the benefits of cost and power reduction (fewer processor elements in the system) and scalable performance enhancements are achievable. There are other use cases of course – pure performance enhancement via SMP or pure I/O throughput enhancement via AMP. What if a hybrid solution was available that covers pure control plane, pure data plane, or both? This would be the best possible solution and well recognized in the industry. But this hybrid case has several ramifications in a real multicore implementation, the scenario for which is that solutions before multicore most often have consisted of classical multi-CPU, or multi-board systems, each with its own mission requirements and therefore separate processing solutions. The main issues are:

a. That a multicore solution [OS?] must support both SMP and AMP models. This gives maximum flexibility to designers in choosing which functions to consolidate in a multicore device, and it even offers the flexibility.

b. That a multicore solution requires little or no modification of consolidated legacy applications. One does not want to redo the architecture of a working solution when migrating to a multicore device.

c. That a multicore solution should support multiple operating systems. The emphasis on the word “should” means simply that most legacy systems in the real world right now feature a mix of operating environments for control plane and data plane – Linux, various RTOS, and even “bare metal” or “run-to-completion” or “simple loop polling” executives. But this is not always the case for specific implementations.

d. That a multicore solution must support some level of device, resource, and even service (like file systems, consoles, etc) sharing across all cores. Multicore devices have limited hardware resources that are not necessarily commensurate with the number of cores, so as the number of cores grows, then some level of sharing becomes an absolute
requirement. It is also required that single devices/resources be able to be locked to a specific core or application domain.
e. That a multicore solution should support fault localization behavior to a single core or collection of cores. When consolidating processors, it is not desirable to have a fault in one processor domain propagate to other processor domains. The cost of failure that requires complete device reboot dramatically increases with higher core density multicore devices that handle increasingly more traffic.
f. That a multicore solution must support an integrated tools environment for software code debug as well as system performance analysis and visualization.

In addition, this section could also include more advanced use cases such as power management (partial core shut-down etc.), dynamic repurposing of cores in runtime to shift the balance between control and data plane depending on traffic patterns etc.

There are three solution sets that address the above scenarios to varying degrees. These are a) a single OS environment that addresses SMP, AMP, and fault localization, b) hypervisor or virtualization technologies, and c) simple “bare metal” (i.e. no real OS) implementation for multicore devices

Single SMP/AMP OS Solution
What if there were one single OS environment that could address both the requirements of SMP and AMP in a multicore device? Then any application could be configured with its best suited processing characteristics in a multicore device, and run at the same performance profile compared to its previous implementation. This sounds difficult, but there is such an implementation in the market today. It is called

Enea OSE Multicore Edition (OSE MCE) (refer to www.enea.com for more information OSE Multicore Edition and download the white paper on the Enea OSE Multicore Edition). OSE MCE offers both the load balancing and portability (little or no code modification) aspects of SMP processing models, but with support for the performance scalability of AMP models (by locking applications to single cores). Other features of OSE MCE are as follows:

a. Fully integrated high performance, transparent, distributed IPC, for intra-core, inter-core, and inter-device communication – Enea LINX. LINX means one message passing model for applications communications (control messaging) in the system. Refer to www.enea.com for more information.
b. Fully integrated device and resource sharing for applications across all applications and cores, with provision for locking device support or resources to a single application or core.
c. Fully integrated tools for debug and system performance behavior analysis and visualization, including extensive “post mortem” or crash analysis tools – Enea Optima. Refer to www.enea.com for more information.
d. Single core failure/restart and extended fault localization, even down to a single process within a core. This is a hallmark feature of the OSE centralized error handling subsystem.
e. Support for “bare metal” execution environments. Bare Metal means that a single core is either a “run to completion” or “polling processing” environment that doesn’t use multithreading at all. The Enea OSE Multicore Edition can support locking of “bare metal” execution models to a single core within the context of having a full service OS for the overall management of the device. See figure 5 at the previous

![OSE Multicore Executive – Bare Metal Environment](image)

**Reaching Bare Metal Performance**
- Supervisor threads + polling
- Bare-metal performance
- Application decides when to yield
- Same programming model as for regular applications, a single receive point
- But, if sources exist the OSE preemptive scheduling model is maintained

Figure 5. Reaching Bare Metal Performance
And last but not least all applications must run natively on the multicore device without the performance degradation invoked by virtualization techniques, or hypervisors, that are covered in the next section.

However, the Enea OSE Multicore solution in figure 5, or any other single OS solution, does NOT obviously support multiple operating systems. The single OS solution is ideal for application/system migration to multicore where the supported OS is involved, and is also idea for “Greenfield” or completely new designs. Another technique is required to address a multi-execution environment situation.

A Brief Introduction to Hypervisors and/or Virtualization

A single OS environment that can support all necessary processing models offers many advantages, especially in performance. But what about the scenario where heterogeneous operating systems are required due to legacy issues for a particular system implemented on a multicore device? This is where hypervisor or virtualization technologies come into play.

A hypervisor for a multicore device is an implementation of virtualization concepts. Virtualization is a technique used to create an execution environment for software that is similar to the one for which it was originally designed, but on different hardware or operating systems.

Operating System virtualization provides binary compatibility layers that may run on heterogeneous operating system environments, while presenting an interface identical to or at least similar to the original OS environment. For example, the ability to run Linux applications on Microsoft Windows (like VMWare) uses a virtualization technique that simulates the behavior of the Linux operating system on Windows. This is logically, a heterogeneous AMP model. But enabling multiple instances of heterogeneous operating systems on a single machine involves solving technical challenges in virtualization and resource isolation, while retaining complete binary compatibility and at an acceptable level of performance. There are two virtualization strategies: full or partial virtualization. In either case, the virtual machine (VM) virtualizes the hardware to provide the illusion of real hardware for the operating systems executing on this virtual machine.

Full virtualization of the underlying hardware requires virtualization of all the capabilities of the processor and often leads to performance overheads that are much higher than the non-virtualized versions of the OS. The benefit of full virtualization is that it allows operating systems to run unmodified, although at the cost of a significant performance overhead.

In partial, or para-virtualization the underlying hardware is not 100% simulated in software. Some specific OS functions are directly modified in order to map to the specific hardware environment. However, the performance of partially virtualized architectures is much better than the fully virtualized machines, usually falling within a few percent of the non-virtualized versions. The other key requirement for running multiple operating systems in the context of a virtual machine is the ability to isolate the physical resources of a computer – in other words, isolation and protection of each virtualized OS domain from each other. This is achieved by time-space partitioning. Time-space partitioning has its own interesting issues and is not further explored here.

Both full and partial virtualization techniques support 100% application compatibility. But full virtualization supports full binary compatibility of the guest or virtualized OS, but para-virtualization for performance enhancement requires some modifications to the guest OS. Both also retain the benefits of multiple address spaces within a single operating system instance. One significant difference between a stand-alone operating system and a virtualized version is that the virtualized OS runs in a less privileged mode (user mode). This is necessary since...
the virtual machine that provides the virtualized architecture is the sole entity that is running at the highest privileged level (supervisor mode). Figure 6 shows the generic architecture supporting multiple heterogeneous operating systems running on a virtual machine.

Virtualization offers good support for configurability, portability, and scalability (meaning increased performance without software change) for heterogeneous SMP/AMP and/or multi-OS based systems. But it still is true that a homogeneous OS would be better served performance wise by a native, non-virtualized single OS solution, as mentioned in the section about Single SMP/AMP OS Solution.

**Hypervisor Use Cases for Multicore**

After the above abstract discussion on virtualization, let’s now examine the actual use cases for virtualization implementations in multicore devices, as per the discussion at the beginning of this section. There are two major use cases that echo some themes from the above discussion:

- a. Consolidation of multi-OS environments
- b. Segregation and logical separation (or isolation) of multiple execution environments
- c. Performance optimization for multi-application environments

The first and second use cases are obvious, and hypervisors support all the general requirements of the introduction of the section named Multicore Implementations for Telecom/Networking Applications above. The second point, performance optimization, deserves some more treatment though. As noted earlier, virtualization always requires some level of performance degradation compared to a native solution, like the single OS, OSE MICE solution. So how can a hypervisor actually improve performance? The answer lies in the concept of domain partitioning that hypervisors support. A domain is defined as a separate, completely contained, execution environment, be it some OS based application or even a “bare metal” (no OS) execution environment. In a full hypervisor solution, a domain can run in one of several partitioning modes: a) on a single core, b) on multiple cores (as in the case of an SMP OS running on a hypervisor, and c) sharing a single core with another domain – this is sometimes called “core virtualization”. It is this last case that is interesting. Suppose that two application domains do not require the full CPU utilization of a single core in a multicore device. Under such circumstances, then if these two domains could share the same core, then it would free other cores up for other work. The conclusion is obvious – that under some circumstances, hypervisor technology can optimize the performance of multiple application domains (same OS or not, doesn’t matter) by maximizing the CPU utilization of each core. And for designers this has real world implications in vendor selection. Consider the case of a 4 core device from one vendor that has a higher clock speed. If the two core device is cheaper, (or lower power, or less physical size/footprint,) then with virtualization it might be the best performance, power, footprint per cost solution than the 4 core device. In fact, it is this concept that has been the primary driver of virtualization solutions in the established enterprise market domain. But in the embedded software domain the applicability of this core virtualization principle works best when the number of cores is relatively small – in the 2 to 4 core range. The advantages of this technique diminish almost in inverse proportion to the number of cores. Large scale (number of cores) devices are the trend in embedded, and especially in telecom/networking, so one has to be circumspect about employing this principle or technique.

**Conclusion**

So where does this leave us? We have seen that in telecom/networking that the general trend towards adoption of multicore devices points to hybrid SMP/AMP solutions, and that single OS solutions are best for this. But that when there are multi-OS considerations or performance optimization of consolidated application domains in low # core devices, then hypervisor (virtualization) solutions are best. Enea though has a solution that covers all of these cases – namely a solution that...
combines both hypervisor technology and the Enea OSE Multicore Edition native single OS solution. This solution is depicted in figure 7.

The Enea Multicore Platform is a homogeneous OS solution that can support both virtualization of legacy single-core applications and new applications that are developed according to new design principles on the same SoC. It features:

a. HW hypervisor support to provide legacy/Linux virtualization
b. Isolated application domains as either virtualized “guests” or shared memory “verticals”
c. Applications loaded, managed, visualized and debugged via Enea Optima Tools
d. Application and device interconnect via fast OSE IPC (LI) provided by the OSE kernel/hypervisor

However, before concluding, there are some outstanding issues involving the next frontier of multicore processing that bear mentioning. Each of these is virtually a subject worth separate treatment themselves and are outside the scope of this introductory white paper. These are:

a. The future of multicore programming models – is the classic embedded operating system “thread” based scheduling model worth preserving with massive large scale multicore devices?
b. Are there new models for load balancing of applications beyond the age old SMP model?
c. How can recent advances in HW acceleration engines for IPC support change the model for traditional software IPC models?

Look for future white papers from Enea on these timely topics.