Embedded Multicore: An Introduction

EMBMCRM
Rev. 0
07/2009

freescale™
semiconductor

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 010 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
+1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: EMBMCRM
Rev. 0, 07/2009

Contents

Contents

Contents

Paragraph Number | **Title** | **Page Number**

Figures

Chapter 1
Embedded Multicore, an Overview

—Jonas Svennebring

The computer industry is driven by pursuit of ever increasing performance. From high-end customized special-purpose computing in networking, telecommunications, and avionics to low-power embedded computing in desktop computing, portable computing and video games, customers expect faster, more efficient, and more powerful products. However, single core products are showing a diminishing ability to increase product performance at pace with consumer desire. Multicore processing is recognized as a key component for continued performance improvements.

The industry is on a clear path towards an increasing number of cores. Dual- and quad-core devices have been established for several years, and they are just the beginning of the explosion in the number of cores per device.

However, this kind of expansion creates a challenge, not only for the semiconductor industry, but also for the system and software designers who put them to work. Writing applications that execute in parallel is seldom easy; sometimes it is not even possible. So why is the industry moving this way? What are the problems, and how can we smoothly work around them?

Those questions are answered in the following sections:

- "Section 1.1, "Why Multicore?" provides an overview of the reasons behind a migration to multicore, the difficulties in raising clock frequency further and, as a result, improving performance.
- "Section 1.2, "Different Types of Multicore" describes the basic topologies across the array of computational environments, including homogenous systems where all cores are identical and heterogeneous multicore systems where these cores differ, including the three predominant approaches to memory designs: distributed, shared, and hybrid.
- "Section 1.3, "Parallelism" outlines the difficulties of managing a system with multiple cores running in parallel and describes the four common forms of parallelism: bit level, instruction, data, and task.
- "Section 1.4, "System and Software Design" compares the advantages and disadvantages of the two approaches to multiprocessing:

asymmetric multiprocessing (AMP), in which each core runs standalone, and symmetric multiprocessing (SMP), in which the many cores act as one through the operating system.

Subsequent chapters focus on hardware, software architecture (such as AMP and SMP systems), changes to operating systems and why system simulation will play a more important role in the development process.

1.1 Why Multicore?

The migration to multicore devices requires complex changes to system and software to obtain optimal performance. It is reasonable to question whether multicore is worth this additional work, or whether it is possible to continue gaining improvements through single-core devices

Before the advent of multicore, most efforts in improving performance increase were straightforward: Crank up the frequency! But it has become all too apparent that pushing the frequency came at a price. Frequency improvements penalize power consumption, which in turn generates heat that requires more advanced cooling, decreases reliability, and shortens the longevity of the device. So, solving the additional problems that come with increasing frequency costs more money.

A rule of thumb is that doubling the frequency causes a fourfold increase in power consumption. Power consumption itself is only proportional to frequency, but higher frequencies need increased voltage because processors with higher speed transistors leak more than slow ones. Equation 1-1 explains the relationship.

$$\textbf{power} = \textbf{\textit{capacitance}} \times \textbf{\textit{voltage}}^2 \times \textbf{\textit{frequency}} \qquad \textbf{\textit{Eqn. 1-1}}$$

Figure 1-1 compares single- and dual-core implementations of the MPC8641. In a single-core configuration, raising the frequency by 50% roughly doubles power consumption; however, dual-core increases power by only 30%.

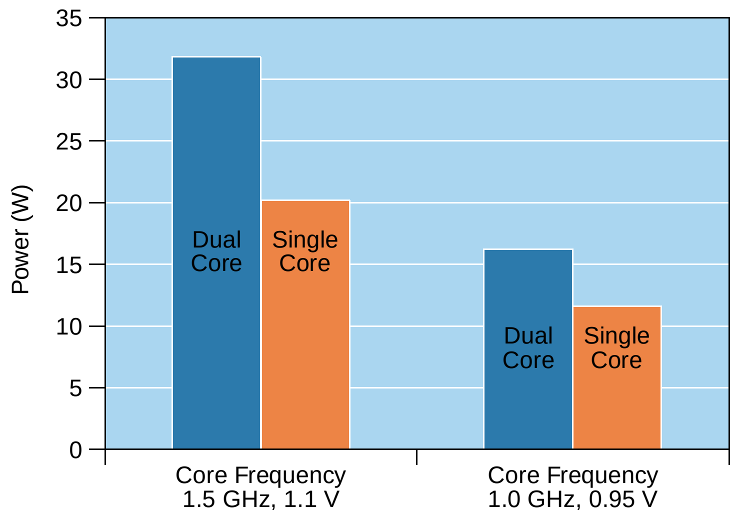


Figure 1-1. Improved Power Consumption as an Incentive for Multicore (MPC8641)

It is well understood that simply doubling the core frequency does not double performance. Techniques such as parallelizing instructions, speculative execution, and pipelining cannot generally scale with the frequency. For example, some stages in an instruction pipeline have internal timing requirements that cannot be met if the processor clock frequency is increased. Therefore, the instruction latency of many instructions cannot scale proportionately and additional pipeline stages are necessary. This naturally increases the number of cycles required for execution and penalizes branches. Although doubling the core frequency may still allow such instructions to execute faster than they would on a core running at a slower-frequency, lengthening the pipeline means that this improvement is less than double.

Of particular significance is the so-called “memory wall” that has materialized as the increase in the on-core speed is not matched by the speed of off-core and off-chip memory and IO subsystems. A high-frequency core matched with a lower-frequency bus will frequently stall as the core waits for data. To some extent such disparities have been compensated by implementing large, fast, on-chip caches, but increasing the size and numbers of on-chip caches subsequently increases both silicon size and power consumption.

Power conservation is especially critical for embedded systems. In a conventional system implementation, the standard upper ceiling of around 20–40 W requires a heat sink and either a fan or substantial air flow for cooling. Ensuring that hot spots are distributed effectively complicates both board layout and the layout of boards within a larger system. This may be acceptable for high-end devices, but not when power requirements drop below approximately 7 W.

When the core runs at about 7 W, the fan can be removed and there is less need for sophisticated management of hot spots. This, in turn, saves money, simplifies board design, and offers more flexibility for placing high-end processors in computing environments where low power is a critical necessity. There are low-power multicore devices that consume only around 2 W, such as the e300-based MPC5121 with integrated graphics and a signal-processing accelerator. For most applications, these devices can be implemented without a heat sink.

Through a great deal of effort and cost, the performance race for desktop and embedded systems has overcome the issues that arise with increasing the frequency. However, innovative workarounds are coming to an end. To continue delivering higher performance with improved power consumption, a new path must be taken. In fact, that trail has been blazed by ultra-high-end systems, such as supercomputers, in which even tens of thousands of CPUs are increasingly common.

1.2 Different Types of Multicore

Given the growing importance of multiprocessing across the computing spectrum and that high-end systems, such as telecom infrastructure, servers, and supercomputers, have long used multiple-core designs as the standard, is natural that the gains of high-end computing be applied to embedded computing systems. These systems have much to offer on how to design and develop software, which is the focus of later chapters.

Multicore devices have been around for many years in different forms. For example, Freescale’s PowerQUICC™ devices implement cores built on Power Architecture™ technology, such as the e500 cores used in PowerQUICC III devices and the single or dual RISC cores in the QUICC Engine™ communication unit. Figure 1-2 shows the different types of multicore environments. A device that contains multiple cores with different types of instruction sets is referred to as *heterogeneous*. In contrast, *homogeneous* multicore devices implement multiple identical cores, as seen in the MPC8641 and P2020. The current trend is to create homogeneous multicore devices, but a significant performance advantage can be obtained by using specialized cores and accelerators to offload the main cores.

homogenous	Describes a multicore environment in which cores are identical and execute the same instruction set.
heterogeneous	Describes a multicore environment in which cores are not identical and implement different instruction sets

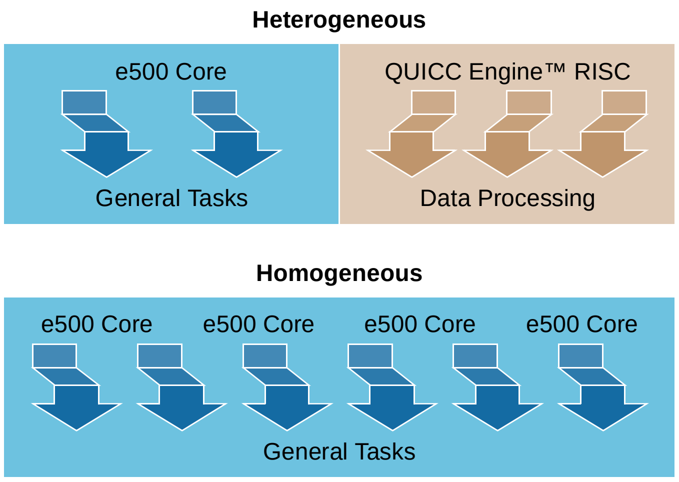


Figure 1-2. Heterogeneous and Homogeneous

Figure 1-3 shows basic core memory topologies.

- In distributed memory designs, each CPU typically has a private memory and communication between CPUs is performed over a high-speed network
- In a shared memory design, there is a public memory that is shared by multiple cores.
- In a hybrid design, there is a shared memory resource, but each core has private memory as well. This allows each CPU/core to have a private memory that can smoothly be shared on a public memory.

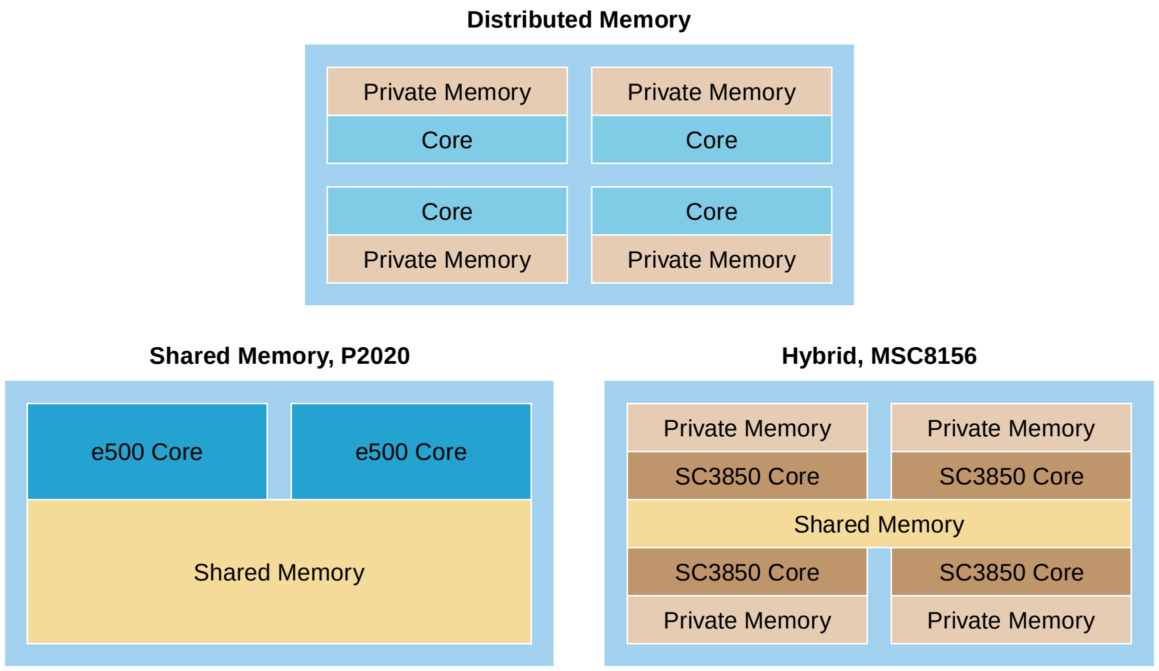


Figure 1-3. Memory Designs in Multiple CPU Systems

As process technology shrinks below 45 nm, devices can be implemented with not just two and four cores, but many tens of cores, an approach commonly referred to as manycore rather than multicore. It is there that the biggest challenges for system and software design lie. A dual-core device can typically provide performance increase without any changes because the operating system can dedicate one core for the main application and the other for special tasks such as interrupt handling. However, in a manycore device applications must be redesigned to make use of all cores to take optimal advantage of the processing power available.

1.3 Parallelism

Parallelization is the central challenge of developing a multicore environment. Of course, parallel execution is nothing new. However, implementing a system in which work can be done in parallel in a computing environment in which order must be maintained at all costs poses problems. Why can’t this be solved in the hardware, or by the compiler or operating system? The answer is that parallelization has already been implemented in these areas.

Parallelism can be thought of as taking four basic forms—bit level, instruction, data, and task. These forms are discussed in the subsequent sections.

1.3.1 Bit-Level Parallelism

Bit-level parallelism extends the hardware architecture to operate simultaneously on larger data. For example, on an 8-bit core, performing computation on a 16-bit data object requires two instructions. However, by extending the word length (the native data length that a core works with) from 8 to 16, the operation can now be executed by a single instruction. Thus as the computer industry has matured, word length has doubled from 4-bit cores through 8-, 16-, 32-, and 64-bit cores.

1.3.2 Instruction-Level Parallelism

Instruction-level parallelism (ILP) is the technique for identifying instructions that do not depend on each other, such as working with different variables and executing them at the same time. Because programs are typically sequential in structure, this is not an easy task. Certain applications, such as signal processing for voice and video, can function efficiently. A DSP, for example, and the Freescale StarCore™ architecture can execute 6 instructions per cycle per core or double this rate when working with video processing. ILP is commonly implemented in the compiler. Other common techniques in this area are speculative and out-of-order execution, features supported by the RISC-based Power ISA, and are implemented in the e500 and the legacy PowerPC cores, such as the e300 and e600.

1.3.3 Data Parallelism

Data parallelism allows multiple units to process the data concurrently. One such technique implemented in hardware is SIMD (single instruction/multiple data), which is implemented in the 128-bit vector instructions defined by the AltiVec instruction set and the 64-bit vector instructions defined by the signal-processing engine (SPE) instruction set.

Data parallelism is also where multicore plays a significant role. Performance improvement depends on many cores being able to work on the data at the same time. When the algorithm is sequential in nature, difficulties arise. Crypto protocols, such as 3DES (triple data encryption standard) and AES (advanced encryption standard), are often sequential and therefore difficult to parallelize whereas matrix operations are generally easier to parallelize because the data is interlinked to a lesser degree. In general, it is not possible to automate data parallelism in hardware or through a compiler because a reliable, robust algorithm is difficult to assemble. Another difficulty is identifying which parts of the software should be parallelized as not all functions benefit from parallel execution. Both of these are problems that you will face when doing it by hand.

1.3.4 Task Parallelism

Task parallelism distributes different applications, processes, or *threads* to different units. This can be done either manually or with the help of the operating system. The difficulty with task parallelism is not with how to distribute the threads, but with how to divide the application into multiple threads. For systems with many small units, such as a computer game, this can be easy. However, when

thread

A flow of instructions that runs on a CPU independently from other flows there is only one heavy and well-integrated task, this division can be very difficult and often faces the same problems associated with data parallelism.

1.4 System and Software Design

Of the four types of parallelism, multicore focuses most on data and task parallelism. Accordingly, this is where the system and software design matters. This section provides a brief overview, different design approaches for handling tasks, and data management in parallel.

The simplest way to progress from single-core to multicore computing is to run each core independently. This approach is called *asymmetric multiprocessing* (AMP or ASMP) in contrast to *symmetric multiprocessing* (SMP), in which all of the cores act as one through the operating system.

asymmetric processing (AMP or ASMP)

An approach to multicore design in which cores operate independently and perform dedicated tasks.

symmetric processing (SMP)

An approach to multicore design in which all cores share the same memory, operating systems, and other resources

In an AMP design, each core runs by itself and often is dedicated to a single task, such as decoding incoming data or handling a specific step in data processing. This can be done in a general-purpose core or in a custom-designed core that has a dedicated security unit for performing encryption and decryption, such as Freescale’s P2020.

An AMP system can be designed in which a set of cores can perform all of the tasks required for the complete processing of a particular task so that the same process can be performed on multiple cores running in parallel. Alternatively, a system can be defined in such a way that each core specializes on a single step in a multiple-step process where results are passed like serial stages in a pipeline.

With either AMP approach, it is important that the hardware distributes the work among the cores. In the case of Ethernet traffic, for example, this can be done by filtering MAC or IP addresses to specific cores. However, with an SMP design, the operating system distributes the work. SMP requires homogeneous cores that share memory such that any thread or process can be assigned to any core at any time. Assuming that an application is divided into multiple threads, this is a very convenient approach because the operating system does most of the work. However, there are performance losses because all cores compete for the same memory with SMP. Currently, this memory bottleneck sets a practical upper limit of about eight cores, although there are ideas for how this can be extended further. These ideas are discussed in subsequent chapters.

Combinations of SMP and AMP yield good results in scenarios in which the main system runs on a few cores that use SMP and are helped by cores running AMP modes as software accelerators. For example, in applications such as telecom 3G/LTE, one or more such accelerators process layer 1 and hand off the processed data to layer 2, which is running with SMP. One core may run a real-time OS and the other Linux (see Figure 1-5).

When multiple operating systems run on the same device, they need to share common resources. For memory, the MMU can easily do this, but for interfaces it is more complicated. The general solution to this problem lies a level below the operating system and is called a *hypervisor*. The hypervisor provides system-level resources that allow operating systems to interface. It

hypervisor

System-level software that allows multiple operating systems to access common peripherals and memory resources and provides a communication mechanism among the cores.

is through the hypervisor that operating systems communicate with each other and with the shared hardware.

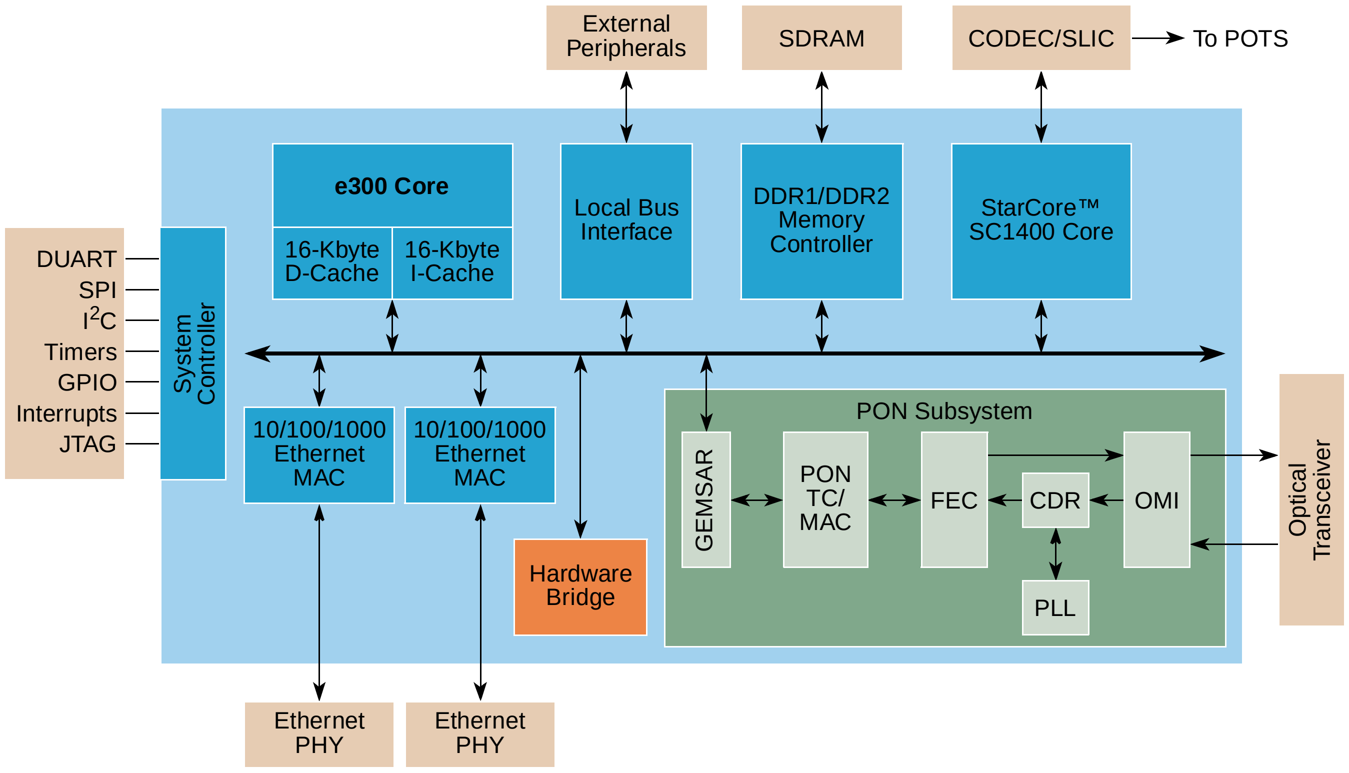


Figure 1-4. MPC7120 GPON Block Diagram

Multilayer systems can benefit from a heterogeneous device with cores dedicated to the specific tasks. Figure 1-4 shows the MSC7120, one such example for GPON (gigabit parallel optical networks, i.e., fiber to the home). It features an accelerator block for the physical layer, a Starcore SC1400 core for signal processing, and an e300 core built on Power Architecture technology for higher layers of processing.

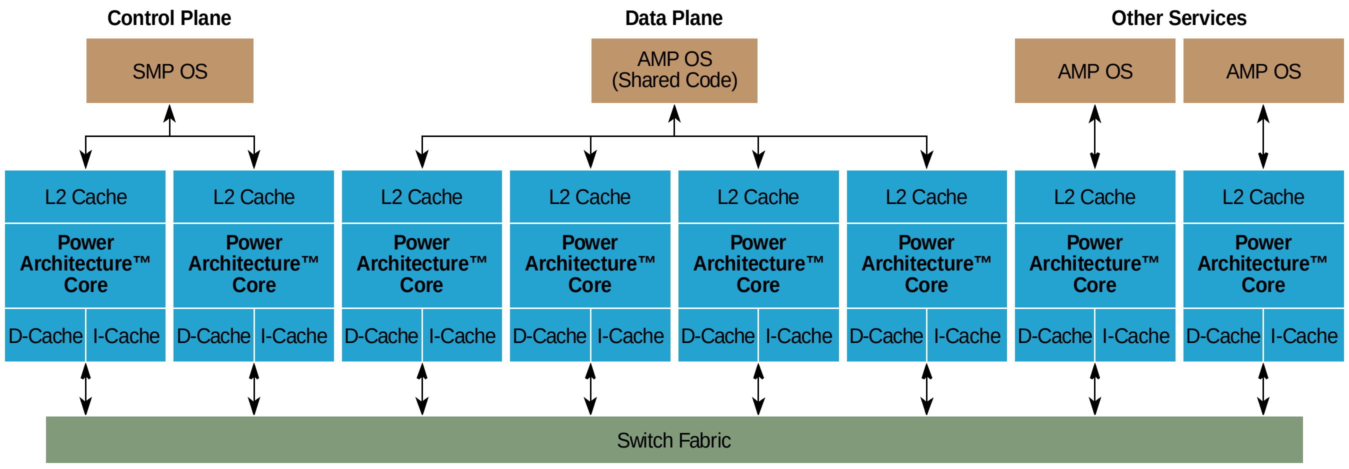


Figure 1-5. Mix and Match

Virtualization is a technique that allows one unit to act as multiple units or vice versa. For the embedded market, virtualization can be used to move a legacy system into a device, such as merging multiple single-core systems into one multicore device.

virtualization
A computing concept in which an OS runs on a software implementation of a machine, that is, a virtual machine (VM).

paravirtualization
A virtualization technique that presents a software interface to virtual machines that is similar, but not identical, to that of the underlying hardware.

Full virtualization features a complete simulation of the underlying hardware so that any software that can run on the real hardware can also run on the virtual machine. The drawback to this approach is the performance overhead. *Paravirtualization* can reduce the overhead. In this scenario, the software needs to be aware of the virtualization and therefore has to be ported. For more information about virtualization, see Chapter 6, “Virtualization and the Hypervisor.”

1.5 Conclusion

Multicore devices provide a path forward for increased performance. This path requires comprehensive and pervasive system and software changes as well as new, innovative hardware designs to ensure that the software can take advantage of the increased computational power. Freescale has years of experience with many types of embedded multicore devices and thus can ensure that all necessary components are present to ease the software burden and to avoid having an inefficient core. This balance is key for multicore applications.

Chapter 2
Embedded Multicore from a Hardware Perspective

—Jonas Svennebring

As the computer industry transitions into multicore computing, the hardware must change shape accordingly. The change must happen not only in the number of cores and how the software uses them, but also in the supporting functionality. Memory and communication interfaces (Ethernet/PCI Express®/Serial RapidIO) and accelerators for crypto, deep packet inspection, and communication stacks that have traditionally resided outside the chip are moved onboard for higher integration purposes and to optimize and balance the loads shared by the cores.

This chapter examines the hardware aspects of multicore computing more deeply by looking at two Freescale homogenous multicore device solutions: general-purpose processors (GPP) and digital signal processors (DSP). It contains the following main sections:

- Section 2.1, “Multicore Devices,” discusses a representative GPP device and a representative DSP device. It also discusses power savings, system-level stability, and security.
- Section 2.2, “From Coprocessors to Multiple Cores,” discusses the evolution of devices to multicore and the attendant technological issues.

2.1 Multicore Devices

We begin by focusing on homogeneous cores: devices that require more re-architecting than a simple addition of cores to the silicon. As examples, we use two Freescale multicore solutions from the two main device groups: general-purpose processors (GPP) and digital signal processors (DSP).

Figure 2-1 shows an example of a GPP: the QorIQ™ (pronounced like “core IQ”) communication processor P4080 based on Power Architecture technology.

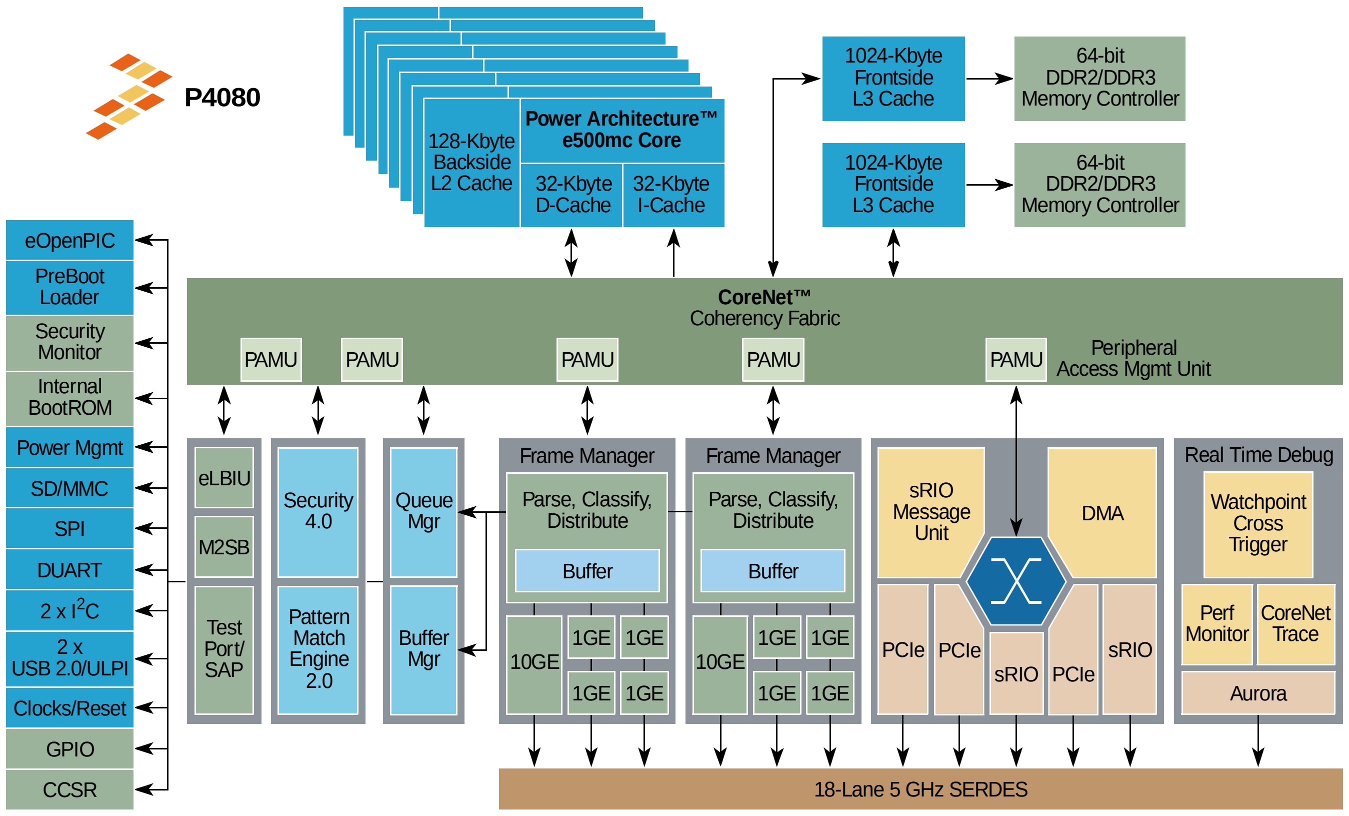


Figure 2-1. P4080 Block Diagram

Figure 2-2 shows the MSC8144 Starcore® DSP.

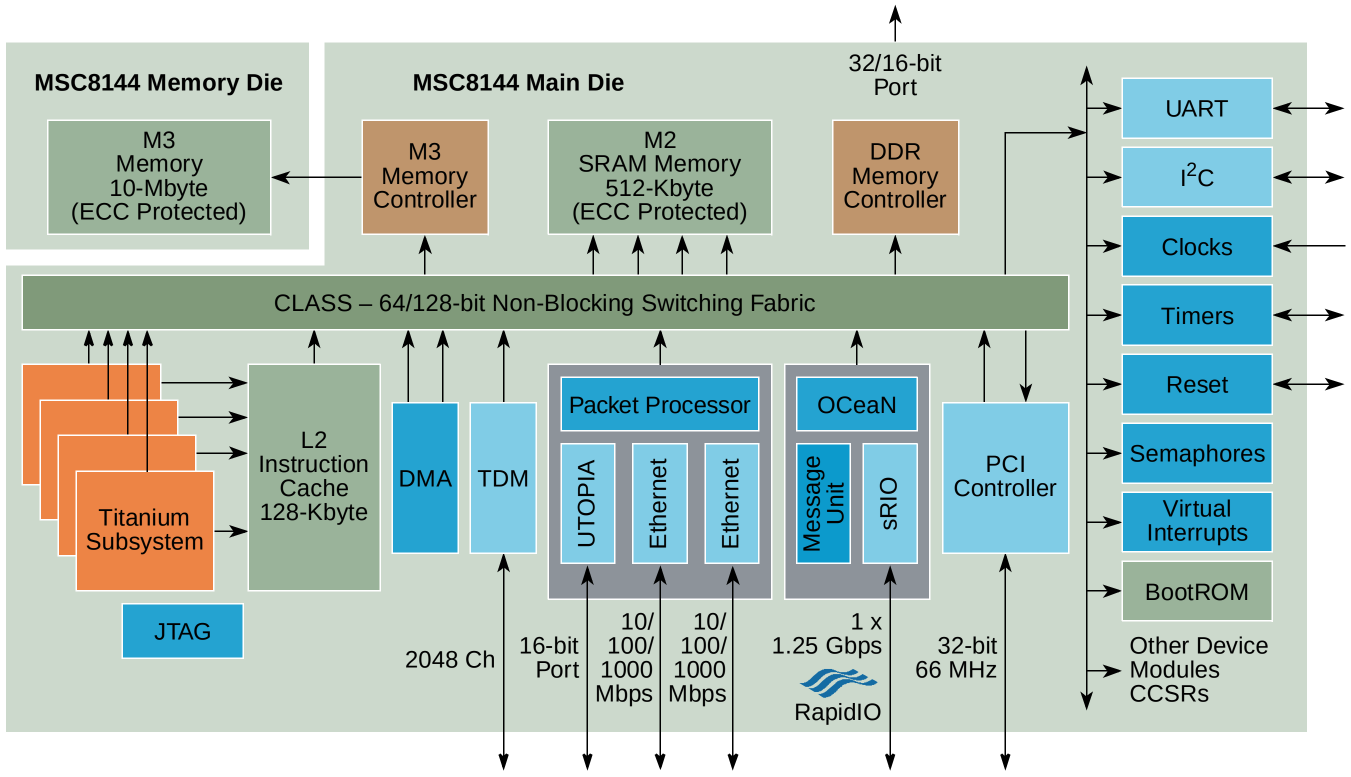


Figure 2-2. MSC8144 Block Diagram

In a homogeneous environment, relatively little in the basic core functionality of instruction execution directly changes. As a reference, Figure 2-3 shows the Freescale e500mc core. Like other cores in the e500 family, the e500mc uses superscalar dispatch, a seven-stage pipeline, and an ability to dispatch and retire two instructions per cycle. The e500mc's five execution units, the branch, floating-point, load/store, and two integer units, allow out-of-order execution to minimize resource and memory stalls and features a completion queue that ensures in-order completion.

Each of the P4080's eight cores run standalone, with the principal goal of having each core run as independently of each other as possible, thus avoiding stalls due to core collisions from attempts to access the same peripherals or memory. This, in combination with high performance and a relatively small die size, makes the e500mc core a good base for multicore devices.

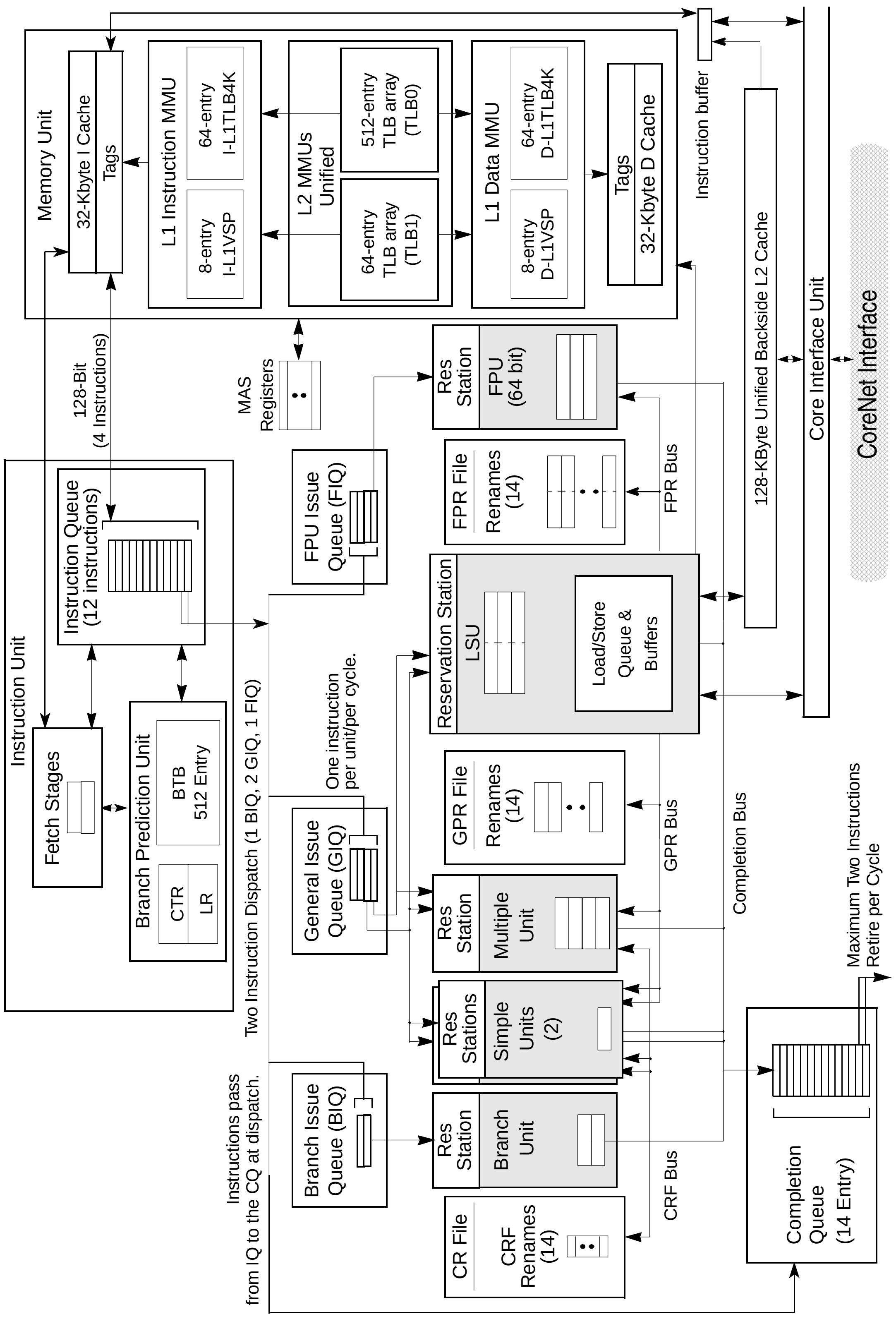


Figure 2-3. e500mc Block Diagram

Coupling the need to synchronize software on the different cores with the need to minimize the number of wasted cycles makes core-to-core communication a critical priority. Starcore devices employ *virtual interrupts* so that each core can get another's attention quickly. With the e500mc, a similar approach is adopted by the Message Send and Message Clear instructions, **msgsnd** and **msgclr**. These two new instructions, now part of the Power Architecture, are used to allow one core to signal a doorbell interrupt to another.

virtual interrupt
A software-triggered interrupt from one core to another.

2.1.1 Power Savings

Another goal of hardware design is to maximize the power consumption made possible by migrating to multicore. Devices typically implement modes that halt execution and power down the device to different degrees—for example, nap, doze, and sleep. However these modes can be difficult to enable in the software, and the wake-up can be time consuming, especially if the PLLs (phase-lock loops) require resynchronization. To simplify this, the e500mc core introduces a **wait** [for interrupt] instruction that halts execution on a specific core until an interrupt occurs. While the processor waits, instruction fetching stops, and the execution pipeline idles.

To further reduce power, the P4080 has separate power rails with different voltages, including complete shutdown (static and dynamic) of all or a subset of cores and multiple PLLs to allow some cores to run at lower, less power-consuming clock frequencies.

2.1.2 System-Level Stability and Security

In a traditional single-processor, single-operating system environment, there is a need for only two privilege levels, one for the operating system (supervisor) and one for the user applications (user or problem-state). But in a multiple-core, multiple-operating system system, it is necessary to add a layer of privilege to coordinate all of the competing domains within the system. Freescale's instruction set architects have extended the Power ISA (instruction set architecture) to include instructions, registers, interrupts, and memory management resources that this additional executive-level software uses to protect memory resources and to provide a virtual interface to peripheral resources that can be shared across all of the computing domains in multicore devices such as the P4080.

This new layer of architecture creates a new privilege level, the hypervisor level. Hypervisor operation is discussed in detail in Chapter 6, "Virtualization and the Hypervisor."

When comparing devices, one should be careful not to look just at the raw core performance, but at how efficiently the surrounding parts can feed the execution units with data, how system bottlenecks are managed and minimized, and how the load can be distributed among the cores. In the Freescale Power Architecture and StarCore devices, the programmable interrupt controller (PIC) can be used to configure how hardware interrupts are prioritized and how they are directed towards specific cores. For example, Tx (transmit) of a device can go to one core and Rx (receive) can go toward another. Another system design approach is to use a fully symmetric interrupt scheme that will ensure that all cores get triggered by an interrupt.

2.2 From Coprocessors to Multiple Cores

1989 First Multicore

Freescale's first multicore device, M68302, was launched 1989. It was a heterogeneous devices pairing a 68000 core with the CPM.

The roots of multicore can be traced to the earliest days of microelectronics and the evolutionary trend for more and more logic to move off the board and onto the same chip as the core. Transistors were combined to form small integrated devices and those devices evolved into processors that were given on-chip caches. The continuous improvements in process technology made it practical to integrate special-purpose functionality, such as interrupt controllers and DMA into the processors. The next step was to move high-speed communications, video controllers, and peripheral controllers, such as PCI and Ethernet into the devices, this to lower cost and increase performance. It was only natural to call them Systems on a Chip, or SoCs.

With SoCs evolving into multicore devices the ability to process data increases significantly. Data must be communicated to and from the device at a much higher rate. This in turn raises a need for specific hardware acceleration. The Freescale PowerQUICC processor family handles communication processing, for example routing and prioritizing incoming packages, by microcode executed in the QUICC Engine communication unit. With higher data rates, such as the dual 10-Gigabyte Ethernet interfaces on the P4080, this processing has to be implemented directly in the hardware but with flexible configurations.

Other accelerators commonly seen are for encryption and decryption of various protocols, table lookup, and deep packet inspection. At high data rates, these things are difficult to do in software and can offload the cores for other operations; that is, the cores are offloaded to the accelerators so they can do other tasks.

2.2.1 Internal Access

Devices typically use a bus-based approach for internal communication. Buses are simple to design, and they give high throughput with low latency as long as there are few masters that initiate data transfers. This is the case with single-core devices, where typically only the core and some advanced peripherals can function as bus masters.

However, the use of buses in multicore devices faces two considerable obstacles: As shown in Figure 2-4, as the number of units increases, so must the physical length of the bus chain. The fixed-signal speed (electron mobility related to the physical properties of the silicon) within the device necessitates an increase in handshake time which in turn limits the clock frequency, reducing bandwidth and increasing latency.

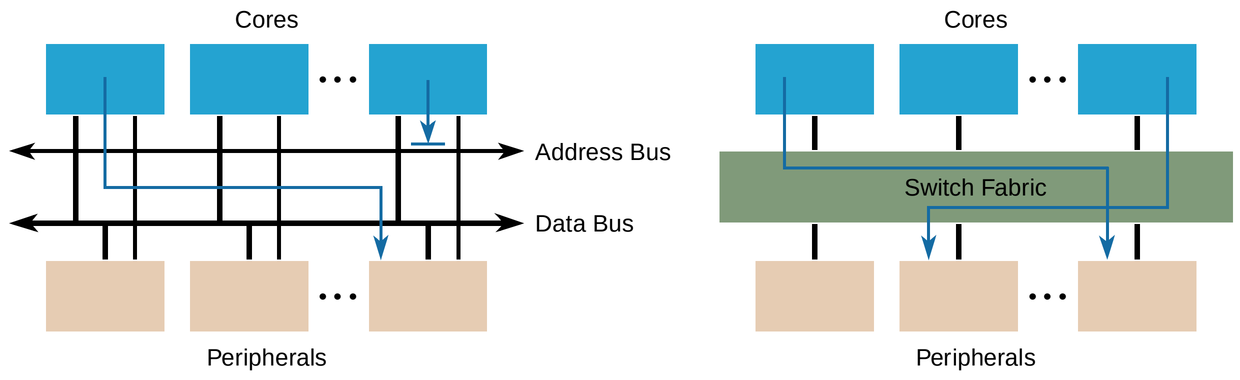


Figure 2-4. Single Bus vs. Switch Fabric

Ironically, although microprocessors can perform almost instantly the sorts of complex complications that decades ago institutions spent millions on and built rooms for, the step into multicore processing has brought a simple problem to light: Because the total bandwidth must be divided among the bus masters, more cores means less bandwidth per core.

Also, with increased bus traffic, the risk of collisions increases and this lowers bandwidth even further. In short, a bus does not scale well above four cores.

The solution, shown in Figure 2-4, is a *switch fabric,* which allows for multiple simultaneous accesses. With such an approach, as one core communicates with the Serial RapidIO interface, another can access memory, a third can use the Ethernet interface, and so on. The advantages of having dual DDR interfaces can now be fully realized because two sets of cores can work with separate interfaces. To reduce collisions, in addition to on-chip caches, the cores can be spread over the two interfaces. This approach of having multiple access points in the memory can also be seen with the M2 memory on the MSC8144. Because the cores are expected to work directly with the M2 memory, it has four interfaces, one for each core. The general drawback with switch fabrics is increased latency. Freescale has minimized this, not only to make the fabric itself efficient, but also to pair the cores with nearby cache memory.

switch fabric

Interconnect architecture that allows data coming in on one of its ports to be redirected out to another of its ports. All inputs are connected to all possible outputs.

memory layer 2 (M2)

A second-level internal memory, similar to an L2 cache.

Because it is more complex than a bus, there is a desire to optimize a switch fabric, both for the cores that use it and for the applications running on the cores. Freescale uses two different switch fabrics, the CLASS in the Starcore DSPs and CoreNet™ technology in the QorIQ communication processors. The software complexity of general-purpose processors increases greatly with multiple cores placed into highly integrated devices, but providing a more sophisticated communication fabric, such as CoreNet, reduces that additional demand on software. To accomplish this, the CoreNet fabric implements advanced functionalities such as cache coherency across all cache layers. CoreNet fabric also supports software semaphores by extending the bit-test to guarantee atomic access between cores. The CLASS is better suited for DSPs as they tend to use less complex operating systems and the application software is more in control. For example, after using a software cache coherency, the switch fabric complexity can be reduced and silicon area can be saved.

In a multitasking system, in addition to translating the effective addresses used in software to the physical addresses used by the memory subsystem, a memory management unit (MMU) must protect applications from interfering with each other. Although the MMU provides protection for each core in a multicore system, other masters, such as peripheral DMAs, can corrupt the memory. Unlike with single-core devices, a multicore system often uses many operating systems which opens the risk of incorrectly configuring other masters so that memory accesses interfere with one another. To prevent this, a new concept of peripheral access management units (PAMU) is introduced into QorIQ devices. Much like an MMU, the PAMU is located at the connection of non-core masters and the CoreNet fabric, as seen in Figure 2-1. The PAMU can be configured to map memory and to limit access windows thereby increasing system stability.

peripheral access management unit (PAMU)

Similar to an MMU, a PAMU is located at the connection of non-core masters and the CoreNet fabric

2.2.2 Memory Hierarchy

The advancement of multicore implementations has been facilitated greatly by continued improvements in process technology. With the introduction of 45-nm technology, core logic forms a relatively small part of the device. In addition, caches are very costly, both in terms of power consumption and size. As the core-to-memory interfaces speed differential increases, it is necessary to increase cache size. As core frequency stabilizes and as caches are shared among cores more efficiently, the demand for cache resources has been reduced.

Looking again at Figure 2-1, we can see that each core has its own Harvard L1 caches, one for data and one for instructions. The caches are very fast and the core can work directly with them by using the core pipeline for prefetch and write-back queues. The unified L2 caches are private to the core but are shared between data and instructions. Medium-sized private caches reduce the risk of resource competition, reducing wasteful cache thrashing between the cores, and give a minimum guaranteed storage area for each core. In other instances, where the core is configured as a software accelerator, the L1 and L2 caches can accommodate all code with plenty of room for data. One can also configure, on a per way basis, the cache as SRAM and address it as normal, store variables, etc.

Backside caches, as in this implementation, are considerably faster than front-side caches and fit well as fast private caches. To maximize usage and minimize core stalls, one can use a feature called cache stashing. Data received from the interfaces are placed in memory and the core is then informed through an interrupt. As the core retrieves the data from memory it instantly suffers from memory stalls since the data has to be transferred from external memory which can be on the order of hundreds of core cycles. However, by using stashing, as seen in Figure 2-5, the data is placed in L1/L2 cache at the same time as it is sent to memory. When the interrupt is triggered the data is conveniently available and the core is fully utilized.

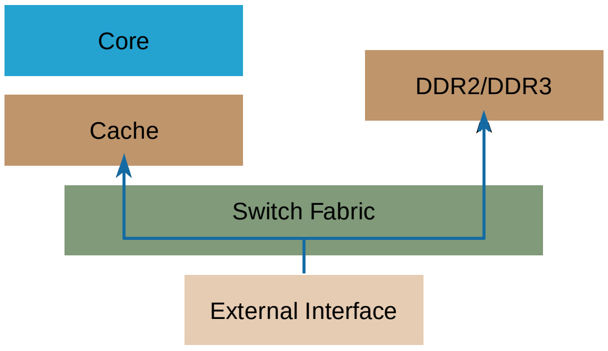


Figure 2-5. Cache Stashing

For most applications, one large code base is either shared by all cores or, if some cores are specialized and running their code in the L1 and L2 caches, is used by a controlling subset of these cores to run this code. Commonly, this type of code is executed randomly with areas of more intense execution. For example, a complex computation or frequently occurring code such as the operating system kernel. The intense and frequent parts will end up in the L2 caches. Having a large, shared L3 cache also captures the less-used parts of the code, which also comprises the largest part of the code footprint. On the P4080 this cache is in line, with the two DDR2/3 memory interfaces.

2.2.3 Interfaces

With architectures built for strong computational performance and data throughput, only one piece of the solution remains—external interfaces to pump the huge volumes of incoming and outgoing data. Freescale's approach has been to go with a high degree of integration of common devices. Both the Power Architecture and Starcore devices can be seen with high-speed interfaces, such as Gigabit Ethernet, Serial RapidIO, PCI Express and general buses. Common low- and medium-speed interfaces, such as UART, SPI, I^{2}C and USB, are handled easily by any core, but for the enormous flow of data associated with 10-Gb Ethernet interfaces, the work must be divided up between the cores.

With the QorIQ family, Freescale introduces the concept of hardware off-loading through the frame, queue, and buffer managers, shown in Figure 2-1. The frame manager is the central part that connects directly with the Ethernet interfaces. Packets are then brought into a parser and classifier unit that inspects the packet headers, including higher layer protocols up to L4, both standard and user defined as well as user tunneled. Based on pre-configured settings the packets are then sent to different queues, forwarded for decryption, sent out on a different interface, thrown away etc. This is all done at line rate even with a load above 10 Gbps per frame manager. For example, we may decide to assign each core a unique IP address but have all TCP packets to port 80 (HTTP) and port 22 (secure shell) sent to core 0. All UDP packets spread evenly between cores 4-7 and ARP traffic to core 1. As the data gets classified and divided up, the buffer and queue managers (Bman and Qman) take over. Because the buffers are already handled by the hardware, this concept changes and simplifies the way that the drivers in an operating system work. The focus can instead shift to data-path configuration.

2.2.4 Debugging and Profiling

With ever-increasing system complexity, the demand for deeper insight into what happens in the chip is needed to find bugs and optimize performance through profiling. Freescale high-end DSPs and GPPs have long had JTAG-based interfaces to support run-time control and debugging as well as for reading out trace buffers and profiling counters to see how the program executed, what took time, how many cycles were wasted on stalls, and what the reason was. Although this is slightly intrusive, this data can also be read out by running software and operating systems, thereby allowing for transmission over standard high-speed interfaces.

The on-chip debug functionality is now expanded to allow for insight into the switch fabric, data managers, and core interaction. The P4080 has, along with the JTAG connector, also a Nexus port directly to memory or over an Aurora interface. This interface has Gigabit bandwidth and can be used to send not only information on program execution but also information on what data is processed. With a powerful external acquisition unit this data can be recorded and as bugs appear, the user one can go through the execution history and identify the root cause. Figure 2-6 depicts the possibilities and shows both the JTAG emulator as well as the trace probe. These in turn interface with the CodeWarrior® development tools on an ordinary PC.

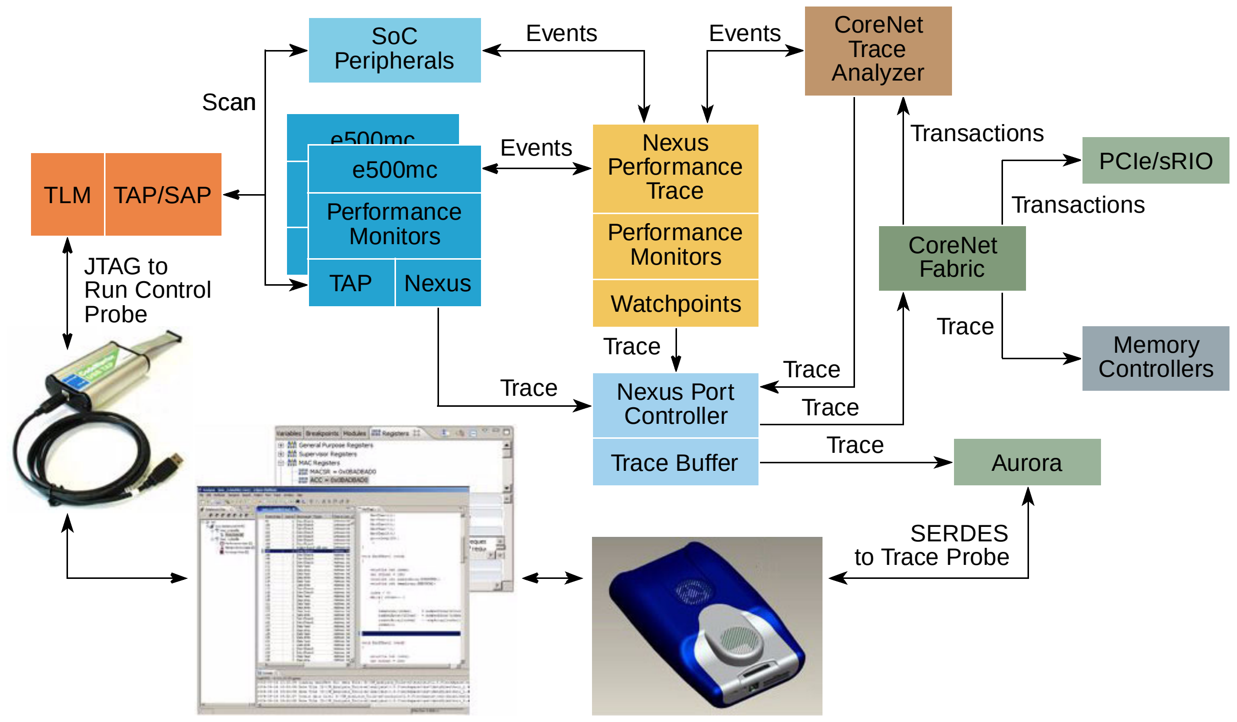


Figure 2-6. Debug Interface

2.3 Conclusion

The transition to multicore devices involves much more than simply adding cores. The glue between the cores has to change from a bus architecture to switch fabrics that allow many to many parallel connections. New approaches to core designs, such as Freescale’s cores based on SC3400 Starcore and e500 cores, are needed that adjust for those changes, and those adjustments are facilitated by modern and flexible architectures the provide new functionality that simplifies core-to-core communications. One key problems is how to divide the incoming data among the cores, especially with the high-bandwidth interfaces. One possible solution is the introduction of hardware managers. Hardware and software are being codeveloped to address the new challenges and to leverage the many new possibilities, that will advance overall computational performance.

Chapter 3
Embedded Multicore: Software Design

—John Logan and Jonas Svennebring

Designing software for embedded multicore devices raises new questions compared to designing for a single-core processor. How do I partition the tasks in my application to achieve the most from the hardware? Should I choose an SMP or AMP software architecture? Which communication and synchronization issues should I consider between tasks?

This chapter explores software design and asymmetric multiprocessing. It includes the following sections:

- Section 3.1, "Amdahl's Law," and Section 3.2, "Gustafson's Law," examine the two concepts that are useful for evaluating parallel algorithms—Amdahl's Law and Gustafson's Law.
- Section 3.3, "Parallelism," examines task and data parallelism.
- Section 3.4, "Symmetric and Asymmetric Multiprocessing," introduces the concepts of symmetric and asymmetric multiprocessing, which are discussed in greater detail in Chapter 4, "Embedded Multicore: SMP and Multithreading."
- Section 3.5, "Processes and Threads," discusses the use of processes, threads, and locks in embedded multicore systems.

3.1 Amdahl's Law

The basic aim of a multicore processor is to increase application performance by allowing multiple tasks to run in parallel. This can involve running multiple independent tasks in parallel, multithreading one application so that it runs across multiple cores, or some mixture of both.

For a typical application, there is a portion that cannot be parallelized (called the serial portion) and a portion that can. Ideally, the serial portion is very small. In the 1960s, a computer architect at IBM named Gene Amdahl formulated the equation shown in Equation 3-1, which is referred to as Amdahl's Law. This equation is used to predict the maximum speedup that can be expected in a typical application.

$$\text{Speedup} = 1/(S + (1 - S)/N)$$

Where: *S* is the portion of algorithm running serialized code
N is the number of processors running parallelized code

Amdahl's Law ***Eqn. 3-1***

For example, consider an image-processing algorithm running on a four-core device ($N = 4$). Sixty percent of the application can be parallelized across all four cores, and 40% ($S = 4$) cannot be. With $N = 4$ and $S = 0.4$, the expected speedup is as shown in Equation 3-2

$$\text{Speedup} = (1/(0.4 + (1 - 0.4)/(4))) = 1.82$$

Example Speedup for a 60% Parallelizable Application in a Four-Core System ***Eqn. 3-2***

An algorithm that took 10 seconds, now completes in 5.49 seconds.

According to Amdahl's Law, the maximum possible speedup is limited by the proportion of the serial portion (*S*) of the application. As more processors are added to the parallelized portion (that is, as *N* increases), the rate of speedup decreases. See Figure 3-1, which shows curves for Amdahl's Law with varying number of cores and varying sizes of serial portion. As more cores are added, the speedup tends towards 1/*S*, as shown in Figure 3-1.

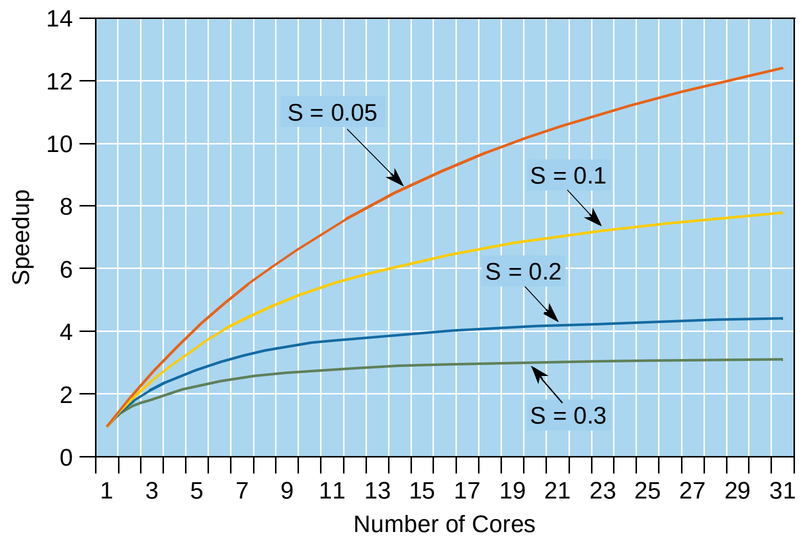


Figure 3-1. Amdahl's Law: Speedup as a Function of Number of Cores

Amdahl’s law seems to impose a fundamental limit on the performance boost achievable with multicore processing; however it makes a fundamental assumption about the application. Amdahl’s Law assumes the problem size is fixed, that is, the ratio between the serial and parallel portions does not change. For example, in our image-processing example, this would mean you process only a fixed size or number of images. Otherwise, the complete algorithm (serial portion + parallelized portion) must be rerun on each run. In many applications, a fixed problem size model is not appropriate.

3.2 Gustafson’s Law

There are many examples of applications that do not have a fixed problem size. For example, in a network routing application, there may be an initial configuration phase that cannot be parallelized, followed by the main task of routing and processing data packets. The number of packets is usually unknown; indeed the design goal may be to handle as many packets as possible. In such a system, it’s easy to see how adding more cores could boost performance—each core can receive a new packet to process when it has completed processing the last packet. Adding more cores means more packets processed in parallel.

In such a system, the relative size of the serial portion decreases over time and the parallelized portion grows. Gustafson’s Law, named after John L. Gustafson, states that the speedup for such a system—known as *scaled speedup*—is as follows:

$$\text{Scaledspeedup} = N + (1 - N) \times S$$

Where: *S* is the serial portion of algorithm running parallelized
N is the number of processors

Gustafson’s Law ***Eqn. 3-3***

Gustafson’s Law shows that for a system where the problem size is not fixed, performance increases can continue to grow by adding more processors. Figure 3-2 shows curves for Gustafson’s Law with different values for the serial portion and number of processors. Notice how speed continues to increase with more cores.

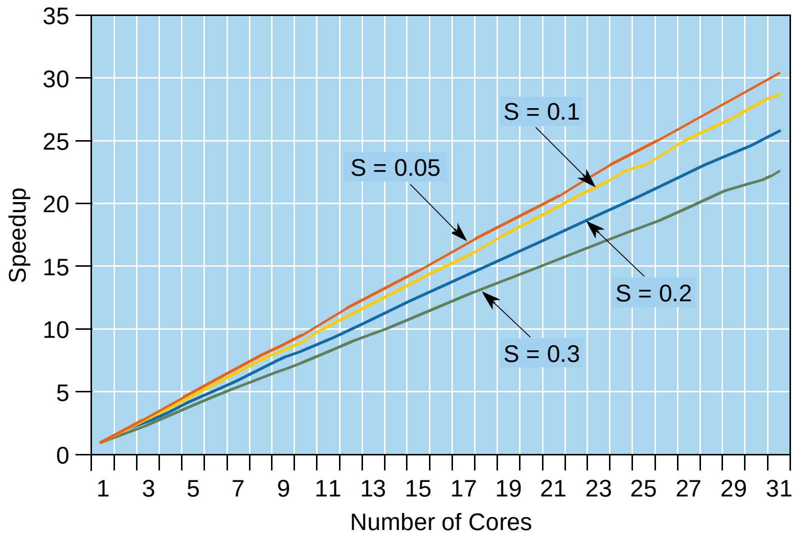


Figure 3-2. Gustafson’s Law

3.3 Parallelism

We now have some simple formulae for evaluating the effects of running an application on a multicore processor. Let's look at different types of parallelism and how an application can be spread across multiple cores.

- Task parallelism occurs when each core executes a different task. For example, imagine a word processor application. It can run multiple tasks in parallel on the same data file, such as updating the display, spooling information to a printer, or performing a word count.
- Data parallelism occurs when multiple cores execute the same task on different data sets, such as running the same algorithm on different sections of an array or running the same algorithm on different data packets.

For example, consider an IP-network router application that receives data packets on a number of network interfaces and that must route the data from the correct ingress port to the correct egress port. The router receives IP packets containing different types of data flows. It needs to identify each data flow type, apply the appropriate processing and route the flow to the correct destination. An algorithm could be written to perform this task. An instance of this algorithm could be run on multiple cores of a multicore device to allow it to handle multiple data flows in parallel. This would be an example of data parallelism.

In addition to handling packets for the data flows, which compose the data plane of the router, the router application must also handle control and configuration tasks, which compose the control plane. The control plane contains a diverse range of tasks, such as routing table updates, making statistical measurements, and handling error conditions. Therefore, it needs to run different tasks in parallel—task parallelism.

The high-level block diagram in Figure 3-3 illustrates how the router application can be implemented on an 8-core device, such as the Freescale QorIQ™ P4080 family of communication platforms, with data path and control path spread across the cores.

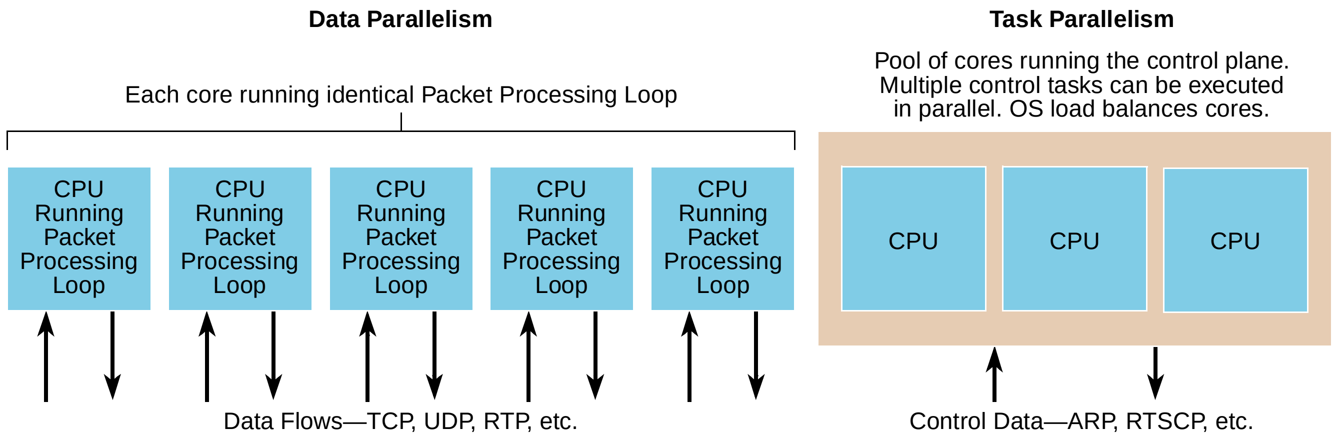

Data Parallelism
Each core running identical Packet Processing Loop
CPU Running Packet Processing Loop
CPU Running Packet Processing Loop
CPU Running Packet Processing Loop
CPU Running Packet Processing Loop
CPU Running Packet Processing Loop
Data Flows—TCP, UDP, RTP, etc.
Task Parallelism
Pool of cores running the control plane. Multiple control tasks can be executed in parallel. OS load balances cores.
CPU
CPU
CPU
Control Data—ARP, RTSCP, etc.

Figure 3-3. Block Diagram of Router Application

3.4 Symmetric and Asymmetric Multiprocessing

Multiprocessing has historically been designed for heterogeneous devices such as math, audio, graphics, or communication co-processors. The latter has been very successful for Freescale due to the devices based on QUICC Engine™ technology. This type of multiprocessing is called asymmetric (AMP or ASMP) because the cores are different or have different system views and hence cannot share the burden of one task between them.

Due to the inability to increase core frequency, multiprocessing has recently been used to increase the computational performance of the device itself, which requires devices with multiple identical cores. By having identical, equal powered cores with full access to the memory we get a symmetric hardware. Symmetric multiprocessing (SMP) has many advantages for simplifying software design. We will explore these advantages in Chapter 4, "Embedded Multicore: SMP and Multithreading." Although symmetric hardware allows for SMP software, it does not have to be used that way. Each core could be dedicated for a specific task just as easily. For example, one core could have industrial control with real-time functionality, and the other could have user interaction.

The next chapter will focus on SMP software design and operating system functionality.

3.5 Processes and Threads

Processes and threads are the two mechanisms that allow an operating system (OS) to provide parallel processing. The term 'process' describes an instance of a program being executed. It has an associated address space and a process control block, which contains attributes and state information about the process.

thread

A flow of instructions running on a CPU independently from other flows

Each process contains one or more threads of execution, or threads. A thread is a basic unit of program execution consisting of a flow of instructions that run on a CPU independently from other flows. Threads within a process share the address space and resources. An SMP operating system schedules threads for execution on the available cores on a device. On a single-core device, threads are time-sliced to give the illusion of multiple tasks running simultaneously. On a multicore device, threads can be truly run in parallel.

For an operating system, switching between processes is a relatively heavyweight task. Typically, context and state information has to be saved for the old process and loaded for the new one, and changes must be made to memory mapping. Swapping between threads in a process is a more lightweight task because the address space and resources are common. Figure 3-4 shows the relationship between processes and threads.

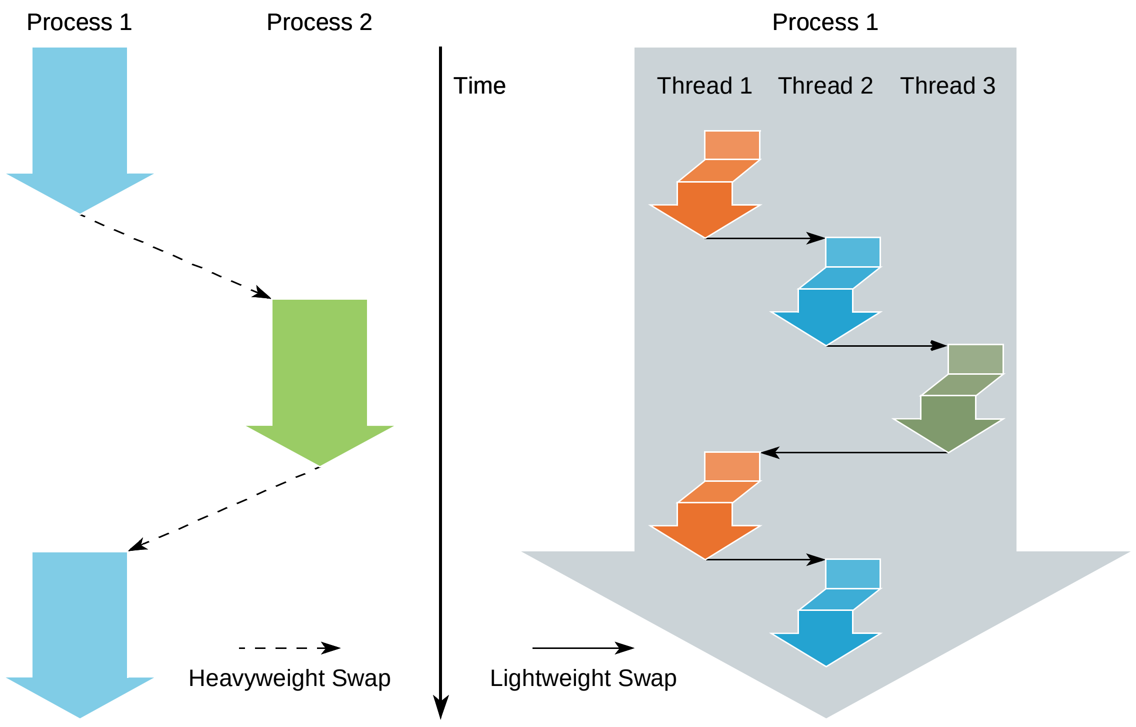


Figure 3-4. Processes and Threads

To illustrate the difference between processes and threads, we conducted a test on a Freescale MPC8548 running Linux 2.6.23 in which we looped a create and destroy cycle 500,000 times. As a process, this took 115.82 seconds, but as a thread, it took only 40.81 seconds. These results are architecture dependent, but the comparison roughly characterizes the complexity differences between a process and a thread.

3.5.1 Task and Process Mapping

In a desktop PC, the operating system takes care of scheduling and running processes and threads. Most desktops run using an SMP operating system—one OS running across all cores—where processes or threads can be mapped to any core. In most cases, the operating system attempts to run threads on the same cores each time they execute. This boosts performance when caches are already loaded with the required data from previous runs.

This linking of threads to specific cores is called processor affinity. Processor affinity has two forms: soft affinity and hard affinity. Soft affinity exists when the OS prefers to link a process/thread to a specific core, but can choose another if needed. Hard affinity exists when the user/programmer specifies exactly where the process/thread should run. The OS also performs load balancing—spreading the required tasks across cores to minimize waiting.

processor affinity

Modification of the native central queue scheduling algorithm. Each queued task has a tag indicating its preferred/kin processor. At allocation time, each task is allocated to its kin processor in preference to others.

soft (or natural) affinity

The tendency of a scheduler to keep processes on the same CPU as long as possible

hard affinity

Provided by a system call. Processes must adhere to a specified hard affinity. A processor bound to a particular CPU can run only on that CPU.

In an embedded application, it may not be desirable for an OS to completely control mapping of the tasks using soft affinity. Returning to the IP router application example, there are two classes of tasks to be done: data plane and control plane. Data plane tasks handle packets and data flow, such as VOIP, video streams, and network gaming. These typically require low latency, and the system designer must guarantee some Quality of Service or throughput figures for the design. If the data path can be mapped to a specific subset of cores using hard affinity, it is much easier to design and test for these requirements. The control plane is less sensitive to latency and has a large mix of different tasks. It is desirable to map these tasks to another subset of cores and allow the OS to schedule them as appropriate.

Within these two subsets, users may wish to further constrain tasks to particular cores. For example, on the data plane, it may be possible to have one task per core running a very fast algorithm to process data flows. If each task has a single thread, the OS scheduling overhead would be removed. Indeed, if this were possible, it could be possible to run the parallel data plane algorithms without a full OS.

3.5.2 Run to Completion

In the early days of computing, users used polling to check for events such as received data on the serial port and key presses on the keyboard. However, for applications with many such possible events, checks with nothing to report wasted time. Moreover, if the event was a burst of data, the first data could have been overwritten by data arriving later by the time polling detected the event.

To address this, interrupts were introduced. Just as it sounds, interrupts permit the event to interrupt the core, which responds by handling the condition associated with the event. When finished handling the interrupt, the core returns to its previous work. The advantages to interrupts are many, but disadvantages also arise when there are large numbers of incoming interrupts. Interrupts have an overhead latency required of jumping to and from the event with registers, stack, privilege level etc., to be shifted out when the interrupt is called for and back in again when it returns. Hence the interrupts are taking core cycles away from the regular applications.

Multiple cores have the flexibility to allocate interrupts among themselves. As described in Chapter 5, “Embedded Multicore: SMP Operating Systems,” users can clear out a core and use it only to batch-process data. This principle of run to completion is similar to polling, but in this case, other cores use interrupts to handle events. Polling for more data is only required after a set is processed.

Such an approach yields higher core utilization and also less complex software as it can run with a very limited OS or even on bare metal. Freescale refers to this as an LWE (lightweight executive) and supports it with a rich set of library functions. The LWE contains standard functionality, such as memory management, as well as device drivers and protocols to interact with operating systems running on the other cores.

bare metal

Bare metal (or bare board) is application software running directly on the hardware, i.e. running without an underlying operating system.

3.5.3 Interprocess Communication and Synchronization

In a system with multiple threads and processes, some communication between processes is needed to pass information or status. Most operating systems provide a set of functions to allow inter process communications (IPC) and to enforce synchronization. Pipes, sockets, message queues, and signals are common constructs used to provide communication. This document does not look at communication mechanisms in detail; OS documentation should contain full details of the resources it provides.

pipes

Software connections between programs. e.g., in Linux command line “ls *.c > grep main”

Synchronization is required between threads to prevent them from working on the same data or on shared resources at the same time, which causes data corruption. This can be achieved by adding functions to protect the application’s critical sections, which are the sections of code that manipulate shared data. Most operating systems allow a range of synchronization functions based on two types—semaphores and mutex (mutually exclusive) locks.

3.5.4 Semaphores and Locks

A semaphore allows or blocks access to a section of code. It consists of two operations—a test function and an increment function—which use an integer variable. As shown in Example 3-1, there are two functions: P (*proeberen*, test) and V (*verhogen*, increment). Both operate on the semaphore variable *s*.

proeberen and verhogen

Proeberen is Dutch for ‘Try’, Verhogen is Dutch for ‘Increment’—so named by Edsger Dijkstra, the Dutch computer scientist who defined the functions.

Example 3-1. Semaphores

```
P(s)
{
s = s-1; /*This must be an atomic operation*/
wait until s >= 0 {}
}
V(s)
{
s = s+1; /* This must be an atomic operation */
}
```

Imagine a scenario where A and B are two threads that engage in the following sequence of actions.

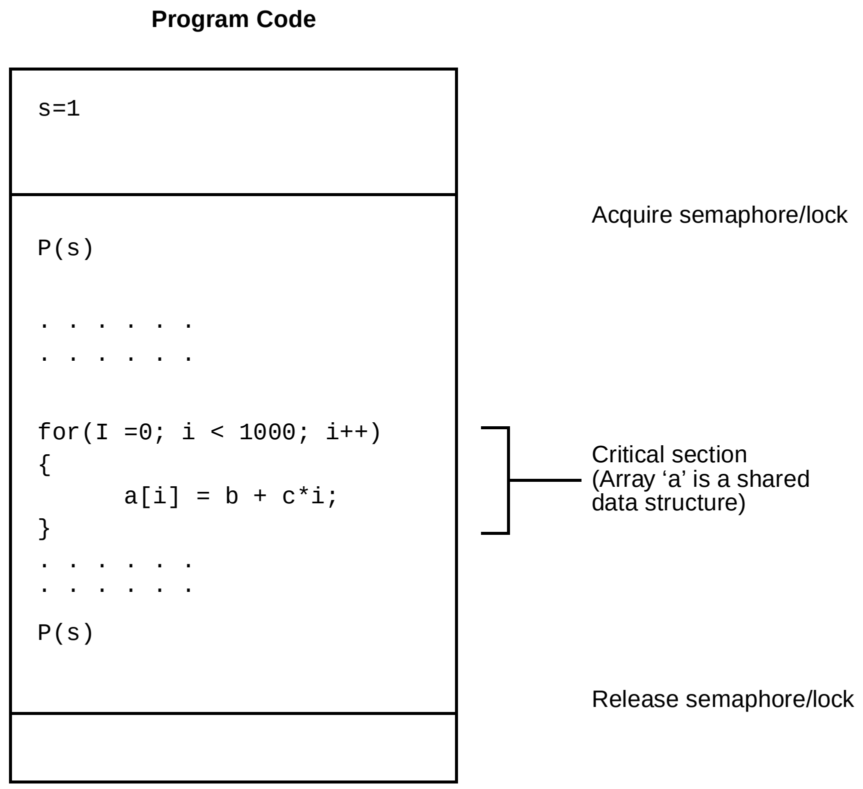
1. Thread A and Thread B try to access the same critical section of code; **s** is initially set to 1.
2. Thread A enters P function and decrements **s** (**s** = 0).
3. Thread A can continue to execute the critical section. Meanwhile, Thread B also enters P and decrements **s** (**s** now = –1). Thread B must wait while **s** is less than 0.
4. Thread A runs the critical section and enters the V function, incrementing **s** to 0.

5. Thread B can stop waiting and execute the critical section.
6. Thread B finishes and runs the V function, incrementing **s** (**s** = 1).

The increment and decrement functions can be implemented in software, but are commonly implemented with a special bit test and set atomic instruction to ensure that there is no task switch between test and set. In Freescale's QorIQ family, the CoreNet switch fabric also has functionality that ensures exclusive access between the cores such that only one core has access to the lock-bit at any given cycle. The Starcore family can do similar operations with cache configurations; they also have special hardware semaphores. Such techniques simplify software design.

In the above example, a thread waiting for the semaphore prevents a core from executing any other code; the core sits in a loop waiting. Most operating systems add wait queues to prevent this from happening. When a thread is waiting for a semaphore, it is put on a wait queue. The OS can then schedule another thread to run on the core. When the semaphore become available (**s** > 0), the thread is taken off the wait queue and can resume operation. Semaphores can have values greater than 1, allowing more than one thread to share the resource at the same time. This is also useful for sharing resources that can support multiple clients.

Figure 3-5 depicts a critical section of code containing a shared data structure, in this case array `a[]`.



```
s=1

P(s)

. . . . . .
. . . . . .

for(I =0; i < 1000; i++)
{
      a[i] = b + c*i;
}
. . . . . .
. . . . . .

P(s)
```

Figure 3-5. Using Semaphores to Protect a Critical Section

Mutex locks, or locks, are similar to semaphores, but a lock can only allow one thread access to a critical section. Typically, a lock has acquire and release functions. Chapter 4, “Embedded Multicore: SMP and Multithreading,” addresses this topic further, explaining common issues and how to optimize lock usage.

Conclusion

At this stage, we have some ideas about how to partition a system. Figure 3-6 shows a block diagram of the application partitioning, providing an example of implementation on a real machine. The control plane processing is done in a single SMP configuration running, for example, Linux. The data plane is implemented using cores running AMP mode with an identical packet processing loop. Each of these could run an RTOS (real-time operating system) such as Enea's OSE® or as an LWE. Within each of the two planes—data and control—there is room to further parallelize specific parts of each task's algorithm. Tools such as Amdahl's and Gustafson's laws can be used to estimate the performance with particular algorithms, allowing one to make a judgement on the best solution.

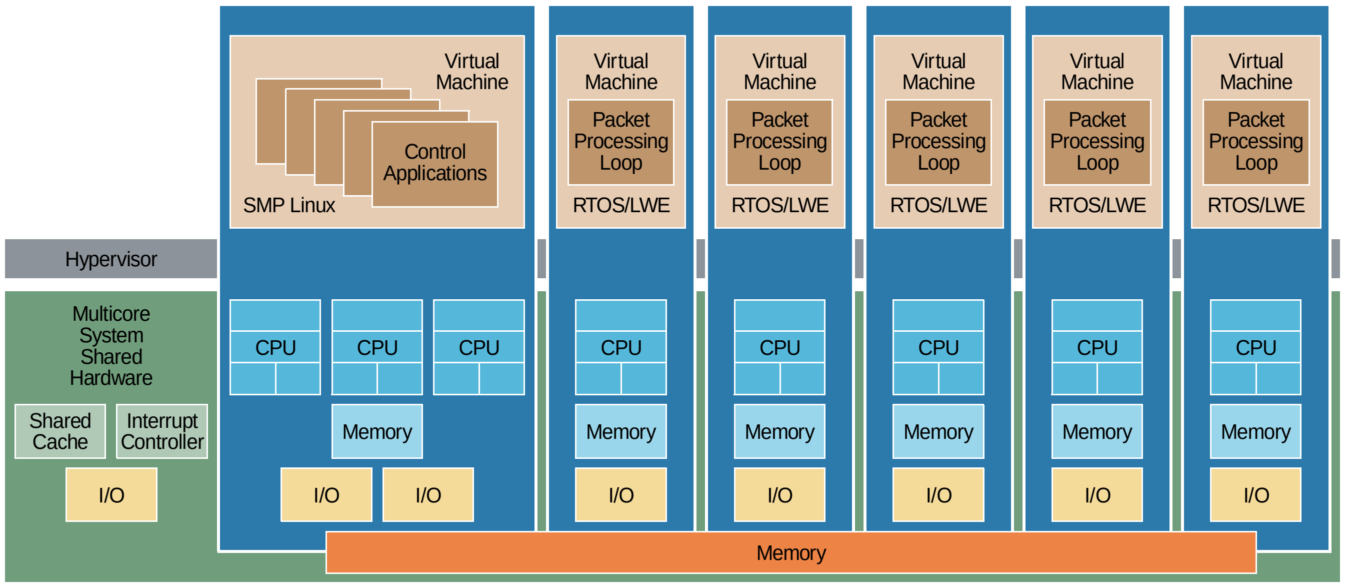


Figure 3-6. Network Routing Application

Chapter 4 Embedded Multicore: SMP and Multithreading

—Jonas Svennebring

As discussed in the previous chapters, multicore processing overcomes many barriers found in single-core computing, particularly in terms of performance and power management. Symmetric multiprocessing (SMP) simplifies the changes required to reap the full benefits of migrating to multicore. However, in order for the operating system to balance an application over the cores, software must be retooled to take advantage of parallelization.

This chapter describes different techniques for parallelization and explains how to implement them. It includes the following sections:

- Section 4.1, "Introduction to Symmetric Multiprocessing," summarizes advantages and disadvantages of symmetric multiprocessing.
- Section 4.2, "Parallelized Application Designs," describes the three primary design approaches to parallelization: master/worker, peer, and pipeline.
- Section 4.3, "Macro- and Microparallelization," describes different levels of parallelization and the support provided by the POSIX and OpenMP multiprocessing standards.
- Section 4.4, "Performance Constraints and Common Pitfalls," describes special concerns for working in a multiprocessor environment, and how to address them through the use of various locking strategies.

4.1 Introduction to Symmetric Multiprocessing

Symmetric multiprocessing (SMP) is a system with multiple processors or a device with multiple integrated cores in which all computational units share the same memory. This chapter focuses on the latter. With the support of an SMP-aware operating system, each core can be load-balanced to ensure that the workload is evenly distributed across the system for maximum overall performance. Because the memory is shared, any core can handle any task at any time. The operating system scheduler assigns a task for each core rather than selecting one task at a time to run system wide.

An advantage of SMP systems is their relative ease of implementation; they work similarly to a single-core system, but with maximum performance proportionate to the number of cores, as described by Section 3.2, "Gustafson's Law." The focus shifts to the application and how to partition it into different tasks, as is described in the following sections.

The disadvantage of SMP is scalability. Commonly, 8 to 16 computational units are believed to be the maximum number of cores that improve performance. This is due in part to Section 3.1, "Amdahl's Law," which shows that even a small amount of sequential code reduces scalability. It is therefore important that the operating system functionality allows a high degree of parallelization and that system calls from one task do not stall another. On the hardware side, the memory structure also limits scaling of SMP. Because each core needs to access the shared memory, increasing the number of cores increases the number of access stalls.

Both of these limitations can be addressed as the number of cores on devices approaches the scaling limit. With respect to software, the kernel and libraries can be reworked to better allow multiple simultaneous calls, limit semaphore locking, or other needs. The hardware architectures can replace the bus connection with more scalable switch fabrics, multiple memory interfaces, and advanced hardware coherent caches, as described in Chapter 2, "Embedded Multicore from a Hardware Perspective."

Another solution is to break the SMP guideline of attaching all cores to a common shared memory. A nonuniform memory architecture (NUMA) provides separate memory resources that are only available to a single core or subset of cores. Then, through a slower connection, one group of cores can access the other group's memory and the scaling can continue. However, NUMA complicates task scheduling for the operating system because a particular task cannot be scheduled freely to any other core at any time without incurring an initial penalty in transferring the data to a different memory block. To some degree this extra complexity is already there, inasmuch as cache usage gives a penalty if tasks are transferred between cores. Most SMP systems have processor affinity awareness and take the hardware design into account when scheduling. Chapter 5, "Embedded Multicore: SMP Operating Systems," discusses this in the context of Linux.

For future SMP development, device performance is not simply a multiplication of the number of cores by their individual raw performance. True performance with multicore devices is not equal to number of cores times their performance. Freescale is pushing for fewer but stronger cores, rather than taking the approach of competitors who have many weak cores, which in the end, results in poorer performance because of the inability to scale well. If the surrounding software support and hardware glue is not good enough, the outcome could be very low system performance.

4.2 Parallelized Application Designs

A parallelized application typically has one of the following designs:

Master/worker	One master thread executes the code in sequence until it reaches an area that can be parallelized. It then triggers a number of worker threads to perform the computational intensive work. Once finished, the worker threads turn the result back to the master and become dormant.
Peer	Like the master/worker design except that the master also functions as a peer (worker) sharing the intensive computational work, which saves a thread. Both approaches target applications that have a sequential portion that is difficult to remove, requiring a combination of concurrent and sequential execution.
Pipelined	A pipelining approach can be be applied to application design. By dividing the applications into a series of smaller, independent stages—where the output of one stage is the input to the next—each stage can be placed on a different core, forming a series of decoupled stages in a pipeline. These parts might be such things as different protocol stack layers or specific functions such as encryption/decryption. Pipelining can be very powerful if the degree of parallelization is high, but it may take some effort to generate a constant throughput: pipeline stages should have the same execution latency and they must be tuned such that one stage does not become a bottleneck or, worse, that failure at one stage crashes the system.

4.3 Macro- and Microparallelization

The concept of multitasking was introduced to account for one task wasting a disproportionate amount of time waiting for IO, with the actual computational load composing a small fraction of the overall execution time. Through multitasking, one core can handle multiple tasks, each having almost the same performance as running by itself. However, the driving factor for multicore multitasking is different: to maximize the load on each core. The portions that should be parallelized and distributed among the cores are those portions that are computationally most intensive.

The traditional way of parallelizing tasks is referred to as macroparallelization, where a user or application assigns a larger portion of work to one task, which is then implemented by a process or a thread. This can be a user or a group of users in a client-based application such as a database or web server. It can also be a specific type of work such as the user interface to an application. In such cases, the parallelized task "lives its own life" in that it makes complex decisions, such as error handling, and also executes for a longer time. Because the tasks spend a lot of time waiting for IO, to maximize core utilization, the number of macrotasks should be greater than the number of cores.

macroparallelization

A task, process, or thread is assigned a large portion of work. The code is largely autonomous and capable of making complex decisions.

Macrotasks should outnumber cores.

To maximize performance in a multicore device, and thereby keep each core busy with work, computationally intense parts of an individual macrotask can be parallelized into microtasks—small segments such as a loop or a independent function calls that the microtasks divide up among themselves. Each microtask performs a narrowly defined amount of computational work

microparallelization

Larger tasks are broken into smaller code segments that are performed under the control of the main task.

and then provides the results to the main task. Unlike macrotasks, the number of microtasks should not exceed the number of cores, or they compete individually and add switch overhead.

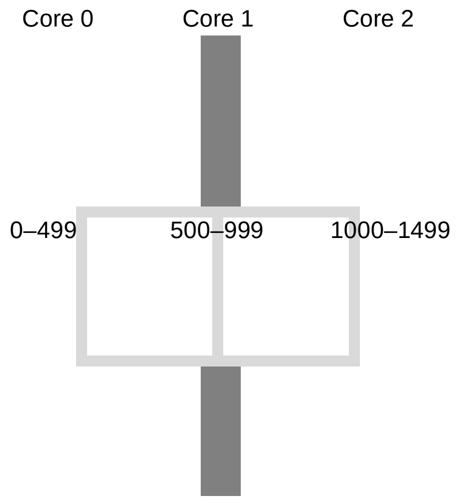
Example 1 shows how a loop is divided into three parts using OpenMP, which is discussed in the Section 4.3.2, “OpenMP” section.

Example 1. Microtask Example

```
void Main()
{
 int bufferSize;
 byte buffer[];

 while(LifeIsGood() == true){
        bufferSize = GetNewData(buffer);

        #pragma omp parallel for
         for (i=0; i < bufferSize; i++){
```



Microtasks and macrotasks can be combined, in which case the operating system must handle how threads should be scheduled and balanced among the cores.

4.3.1 POSIX Threads

POSIX threads, or Pthreads, is a thread API that is part of the POSIX standard for portable operating systems. Initially, the POSIX standard targeted UNIX-like systems, but since its release in 1998, it has grown. Today, it is a standard commodity that provides a dependable foundation for multicore applications.

The API is limited in its complexity, containing roughly 60 functions that are grouped into three function classes. Thread management contains basic functionality for creating and terminating threads, handling change status, and configuring general attributes. There is also a class for mutex locks, used for synchronizing threads and resource usage. Lastly there is a class for conditional variables, which allows communication among threads that share a mutex.

POSIX (portable operating system interface)
A family of related standards specified by the IEEE to define the API.

POSIX threads (Pthreads)
A thread API for portable operating systems.

mutex (mutual exclusion)
algorithms used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by sections of critical code.

Because threads run on a shared memory, they do not need a specific memory-sharing functionality. A global variable is visible to all threads. However, it is important to use the mutexes to ensure that no other thread is updating a variable at the same time. There are also other parts of the POSIX standard, such as message passing, that deal with surrounding functionality.

Example 2 shows a small program that creates two new threads while the main thread continues to execute. Support for Pthread is widely available; for example, the CodeWarrior debugger spawns a new debug window for each additional thread and allows separate control. Compilers also commonly support Pthreads, and in this example we have used the GCC compiler: `gcc -lpthread test.c`

Example 2. Pthreads

```
#include <stdio.h>
#include <pthread.h>

void Thread_Main(void *threadid)
{
  int *tid=(int *)threadid;
  printf("Worker Thread %d\n",tid[0]);
  for(;;);
}

int main(int argc, char *argv[])
{
  pthread_t threads[2];
  int id[2]={1,2};

  pthread_create(&threads[0], NULL, (void *)Thread_Main,(void *)&id[0]);
  pthread_create(&threads[1], NULL, (void *)Thread_Main,(void *)&id[1]);

  printf("Main Thread\n");

  for(;;);
}
```

4.3.2 OpenMP

OpenMP (open multiprocessing)

An API that supports multiplatform shared memory multiprocessing programming in C/C++ and Fortran on many architectures.

Mainly targets microparallelization

pragma (pragmatic information)

General compiler-specific compiler directive in C/C++; pragmas are generally not standardized. For example, a pragma can instruct the compiler to align instructions or data in memory. Another might set optimization level.

OpenMP is a multiprocessing standard that was first released in 1997. OpenMP facilitates adding thread functionality to C/C++ programs usingpragmas, which are an instruction to the compiler. In OpenMP, pragmas, which start with `omp…`, are translated into a function or library call. OpenMP is built on top of Pthreads, but requires a special compiler and libraries for support, as well as a multithreaded operating system. OpenMP has been growing rapidly and has been included with GCC since Version 4.2. You can compile your program easily by using `gcc -fopenmp test.c`

Whereas Pthreads can be used to create both macro- and microtasks (although the latter can be tedious), OpenMP mainly targets microparallelization, as shown in Example 3. After a master thread is started, work-sharing commands such as FOR and SECTION cause it to fork and execute in multiple threads. A loop can be divided statically so that each thread has an equal amount of work. However, in many cases the operating system does not balance each core equally with other tasks, and the loop does not return to the master until the last thread completes. Therefore, more complex schedulers should be used for load balancing among specific application's threads.

Example 3 shows how a data array can be divided among the threads so that each core obtains a new chunk of data upon completion of the previous chunk. In this example, core 1 is the fastest and core 3 is the slowest, which affects how the work is distributed. For dynamic scheduling, the chunk size is equal to or smaller than 1 ÷ *n*, where *n* equals the number of cores. For guided scheduling, the chunk size decreases incrementally to minimize the amount of time that the last core is executing by itself.

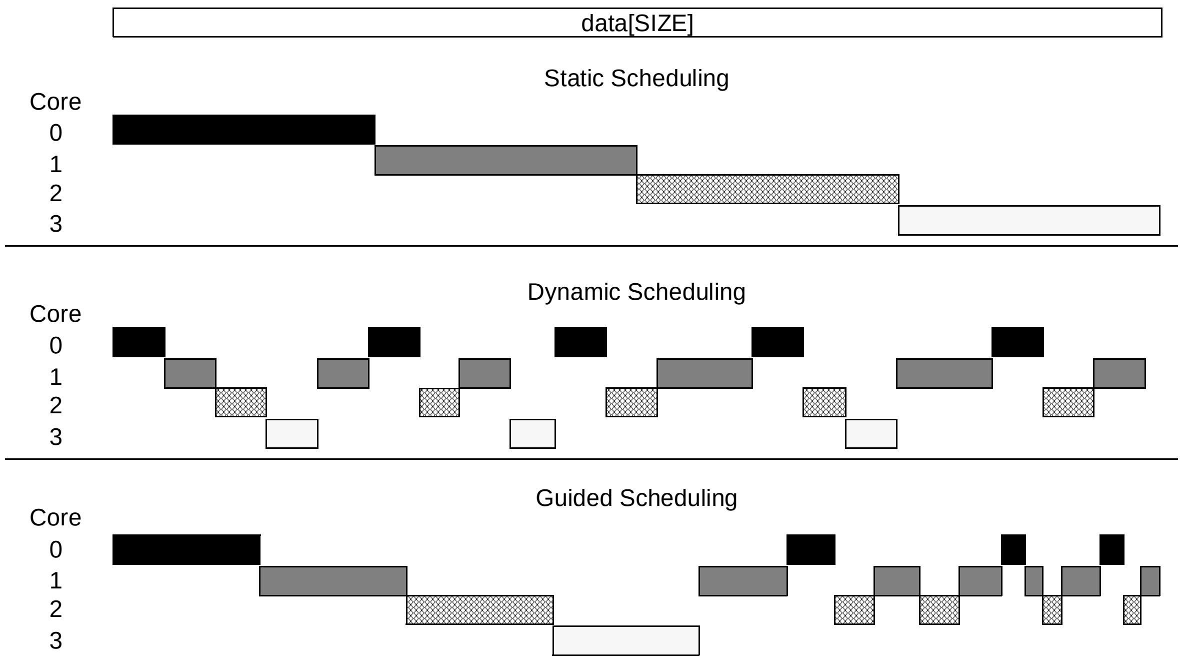


Figure 4-1. Scheduling of Parallel Loops under OpenMP

Example 3. Work Sharing

for → split up operation among threads.
section→ assign independent code blocks to different threads.
single→ block is executed by one thread
master→ lock executed by master, no implicit barrier at the end

```
int main()
{
 int i, buffer[1000];

 #pragma omp parallel for
 for (i=0;i<1000;i++)
    buffer[i]= i+1;

 return 0;
}
```

```
int main()
{
 #pragma omp parallel sections
 {
  #pragma omp section
  calculateA();
  #pragma omp section
  calculateB();
  #pragma omp section
  calculateC();
 }
}
```

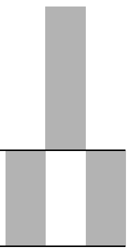
Thread synchronization is a key element to ensure that the threads move along as planned and that there are no conflicts among threads attempting to work with the same data at the same time. A barrier ensures that all threads reach a point before they can continue; a barrier is typically implicit after a work-sharing clause when worker threads join up with the master. Implicit barriers can be removed, which is beneficial for multiple independent loops in a single parallel region. Critical and atomic are standard functionalities that ensure data in critical sections can be accessed by only one thread at a time; typically this is implemented using a Pthread mutex.

OpenMP supports barriers and critical sections as shown in Example 4.

Example 4. Synchronization Clauses under OpenMP

barrier—all threads wait at this point until the last thread gets here.
nowait—removes implicit barriers.
critical—mutual exclusion, only one thread will be in this section at a time.
atomic—similar to critical, advice compiler to use special hardware if possible.
ordered—the structure block is executed in same order as if it was a sequential program.

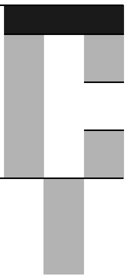
```
int foo()
{
 do_init();
 #pragma omp parallel
 printf(“Hello%d\n”,omp_get_thread_num());
```



Example 4. Synchronization Clauses under OpenMP (continued)

```
 #pragma omp barrier


 if(omp_get_thread_num()==0)
  printf("We have %d threads\n",omp_get_num_threads());
 return 0;
}
```



In a parallel section, such as a loop, each thread has its own copy of the index variable. By default, all other variables that are not defined within the parallel section are shared. However, it is possible to change this, as Example 5 shows.

Example 5 also shows that is is possible to copy larger sets of shared data into a private set as well as to broadcast private data. Typically, the computational work in a loop is to update a shared dataset with new information or the opposite, to calculate a single value such as a checksum from a dataset. By using reduction and specifying an operator, users can merge each thread's individual result into a final result in the master thread.

Example 5. OpenMP Data Scope

shared—Visible to all threads
private—All threads have their own copy
firstprivate—private but init value taken from master thread
lastprivate—private but exit value from last iteration/section

```
reduction(op:var)→ merge results from multiple iterations
  op: + - * & && | || ^
```

```
x[N];
main()
{
  int i, j, k;
  #pragma omp parallel for reduction(+:j) lastprivate (k)
  for(i=0; i<N; i++)
  {
     k = j%10;
     j += x[i] + k;
  }
}
```

OpenMP also contains an API for run-time functions, environment variables, and other functionalities for data and task parallelization. For more information, see www.openmp.org.

4.4 Performance Constraints and Common Pitfalls

By its nature, parallel software works on a shared dataset. At the same time, the dataset must be updated in a structured way, or the software can suffer from race conditions (read–modify–write). The solution is to use different types of locks to guarantee that only one thread at a time can enter into a critical region to update the data (see Chapter 3, “Embedded Multicore: Software Design,” for more information). These locks can also be expanded further, for example, as conditional variables to trigger on a memory region or barriers that ensure that all threads are synchronized at a given point in the program before they continue execution.

The use of locks is not new to multicore software design, and the implications are similar to those introduced with multitasking. But when a program runs on truly parallel hardware, the effects of software bugs are both more obvious and frequently encountered compared to a semiparallel single-core device. In a single-core system, the operating system apportions time slices to each thread, and the probability is low that a read–modify–write operation will be split by a time slice such that another thread updates the specific data in between the operation. Programs on parallel hardware devices are more vulnerable to such errors.

The following are some of the common pitfalls to be aware of when working with multithreaded software:

- Race conditions

  Multiple threads access the same resource at the same time generating an incorrect result. For example, Thread A reads out the balance of a bank account, adds $100, and writes the result back. Parallel thread B reads out the same bank account, subtracts $200, and writes its result back. Because both updates are done before any thread writes the result back, only one of them affects the account balance.
- Deadlocks

  Although locks are the solution to securing access to data structures, locks can also create problems. A deadlock situation occurs when two threads need multiple resources to complete an operation, but each secures only a portion of them. This can lead to both threads waiting for each other to free up a resource. A time-out or lock sequence prevents deadlocks.
- Livelocks

  A livelock occurs when a deadlock is detected by both threads; both back down; and then both try again at the same time, triggering a loop of new deadlocks. Randomizing the allocation algorithm, for example by waiting a random time before trying again, can remove livelocks.
- Priority inversion

  This occurs when a high-priority thread waits for a resource that is locked for a low-priority thread. With threads also running at normal priority, there can be considerable delay before the low-priority thread executes. A common solution to this is to temporarily raise the low-priority thread to the same level as the high-priority thread until the resource is freed.

Synchronization is essential to concurrent programs, but the efficiency with which this is implemented has considerable impact on performance. As was shown in Section 3.1, “Amdahl’s Law,” software running with 95% parallelization on a 10-core device yields only a 7× performance boost. Below are strategies to consider for optimization:

- Lock granularity

  If possible, place locks only around commonly used fields and not entire structures. Make all possible pre- and postcalculations outside of the critical section. This minimizes time spent in a critical section.

- Lock frequency

  Use locks only when needed and minimize synchronization overhead. Lock frequency and lock granularity must be balanced; lower granularity can require higher frequency. Users must evaluate the advantages and disadvantages of locking multiple fields with one lock.
- Lock ordering

  Make sure that locks are taken in the same order to prevent deadlock situations. In POSIX-based systems, there is a `trylock()` that allows the program to continue execution to handle an unsuccessful lock. Use this when needed and `unlock()` resources when all of the locks cannot be obtained.
- Scheduling

  Different scheduling algorithms can affect performance and response time. There are typically a number of schedulers in the system, with the operating system's process/thread scheduler as the most important. But a specific process can also have its own thread implementation and scheduler. Another example is OpenMP, with different schedulers for distributing load between the threads/cores. The scheduler should be tuned to the desired behavior, such as maximizing throughput, core utilization, fairness, or response time.
- Worker thread pool

  When using a peer or master/worker design, users should not create new threads on the fly, but should have them stopped when they are not being used. Creating and freeing processes and threads is expensive. The penalty caused by the associated overhead may be larger than the benefit of running the work in parallel.
- Thread count

  Ensure there are enough threads to keep all cores fully utilized, but remember that too many threads can degrade performance as they compete for the cores. This increases the time required for tasks such as thread switching and synchronization. OpenMP defaults to spawn one thread per core, which can be too many if multiple OpenMP applications are running simultaneously.

4.5 Summary

Symmetric multiprocessing has matured into a stable technology that maps well to multicore devices supporting shared memory. SMP allows multiple cores to smoothly share the workload of an application if it, in turn, has been parallelized into multiple processes or threads. POSIX threads offer a reliable base, with OpenMP built on top. Multiprocessing works much the same on multiple- and single-core devices. However, true parallelization intensifies the need to properly synchronize for maximum performance and to use locks for avoiding race conditions.

Chapter 5
Embedded Multicore: SMP Operating Systems

—Jonas Svennebring and Patrik Strömblad

By leveraging the concept of symmetric multiprocessing, an operating system in a multicore device with a shared memory architecture is able to allocate core resources and load balance tasks or threads between them. The result is simplified software development. This chapter evaluates the following SMP operating systems:

- Section 5.1, "SMP Linux," evaluates Linux, a general purpose system with a broad application base and runs on very large number of hardware platforms.
- Section 5.2, "Enea's OSE for Multicore," evaluates Enea OSE®, a message-passing based real-time system optimized for data plane processing with tougher requirements on stability, determinism, and low kernel overhead.

These two operating systems target different use cases and complement each other in a way that shows the advantages of SMP across a spectrum of OS environments.

5.1 SMP Linux

Linux is a popular desktop OS, and Freescale has broad, well-established support for it on a range of embedded devices, such as Power Architecture and ColdFire. Linux is tightly integrated with development at multiple stages. Freescale has its own release, Linux target image builder (LTIB), and is focused on work with supporting partners such as MontaVista and WindRiver.

The SMP support in Linux was introduced initially to handle multiprocessing, with many devices tied to the same memory, but it suits multicore just as well. The original SMP patch was added in 1995 to kernel version 1.3.42, but it was not considered stable until the 2.0 release in 1996. As a result, this implementation is now mature and very reliable.

One disadvantage of the Linux kernel for SMP is the requirement for reentrant kernel calls and fine granularity locks, which is a considerable portion of the legacy code that still relies on big kernel lock (BKL, that is, calls to **lock_kernel()**). The original 2.0 release used BKL to lock the kernel to one CPU for system calls as a way of ensuring safe concurrency. This locking would represent a drawback with respect to parallelization, and therefore to performance, when scaling to a larger number of cores. The solution is to split the BKL into fine-grained locks, but this poses a delicate problem because often it is hard to foresee the impacts of such a change when other code locks and uses the protected data. An incorrect change could be difficult to detect and could lower stability. Much work was done for version 2.2 and 2.4 to free up the locks, but in kernel version 2.6.6, there were still over 500 BKLs. However, many were in older, deprecated parts that will eventually go away, such as old device drivers. In general, focusing on this problem and on kernel parallelization ensures an increased SMP scalability as devices with more cores are introduced.

BKL (big kernel lock)

A lock needed for SMP support to implement concurrency control in the kernel.

A single, global lock is held when the thread enters the kernel space, for example, after a system call, and released when thread returns to user space. User-space threads can run concurrently in individual cores. Only one can run in the kernel space; threads in other cores must wait to access kernel space. This lock eliminates all concurrency in kernel space.

5.1.1 Task Schedulers and Load Balancing

The OS scheduler assigns tasks to cores by assessing a number of parameters, such as task priority, how much time the task has had, and how long it was since last run. Linux 2.6 has used an O(1) scheduler up to 2.6.23 when the CFS (completely fair scheduler) was introduced as the default choice. We will look first at the O(1) scheduler, then at how the CFS differs from it, and finally at how load balancing works among the cores.

The O(1) scheduler gets its name from the Ordo notation and indicates that the scheduling time is constant over the number of tasks to schedule among. (Constant scaling is good but does not by itself imply that it is fast, just predictable.)

The implementation, shown in Figure 5-1, is based on two run queues. The first queue (active) contains the tasks that are waiting to run. The second queue (expired) contains the tasks that have recently run and therefore must

Ordo (big O) notation

Notation used to describe the complexity level of an algorithm as it scales (for example, O(1) refers to a constant value; O(*n*) refers to a linear scaling; O(n^2) is quadratic;
O(*n*!) refers to factorial).

wait for the other queue to be empty. Note that these lists of schedulable tasks do not include tasks that are waiting for IO.

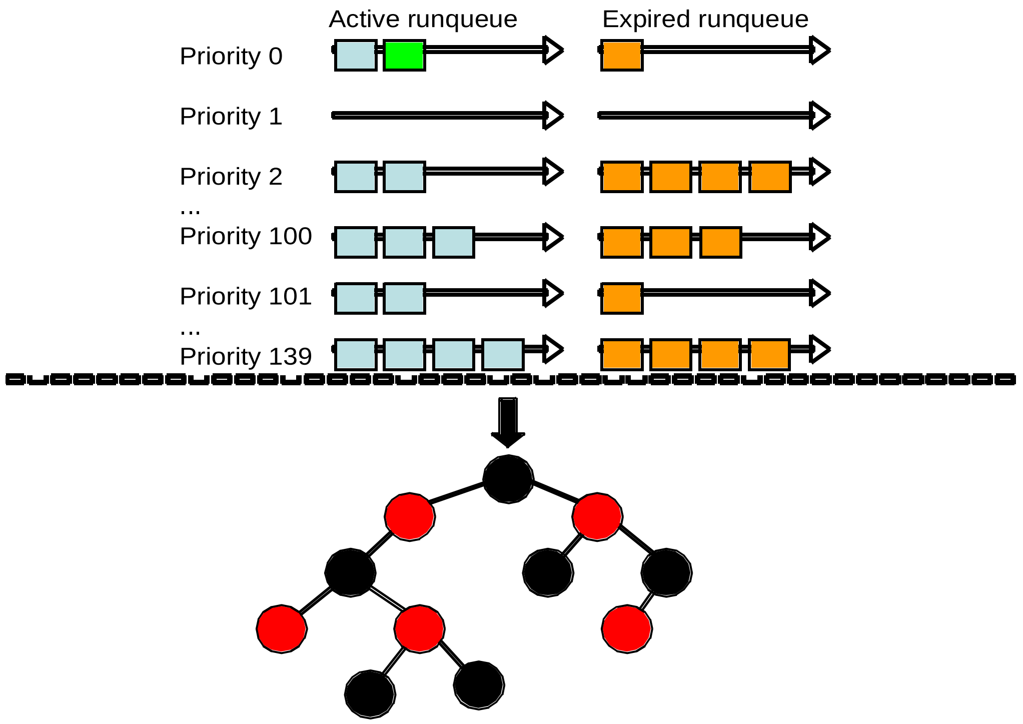


Figure 5-1. Schedulers (Using a Red–Black Tree Structure)

Each of the two queues contains 140 priority levels (lower numbers indicate higher priority); the top 40 are for normal user tasks and the lower 100 are allocated for real-time tasks. The scheduler picks the first task in the active queue of the lowest priority level and lets it run. As soon as the allocated execution time is used up the task is placed in the expired queue. When all tasks at the lowest priority level have run, the expired and active queues at that priority level switch places. If all the tasks are completed (for example, waiting for IO), the scheduler picks tasks from the second lowest priority level and so on. The tasks to be scheduled are always placed last in their priority levels queue.

The O(1) scheduler keeps one such run-queue pair per core with individual locks on them. When a core needs to reschedule, it looks only in its own private run queue and can quickly decide which task should execute next. Load balancing among the cores occurs every 200 ms as a higher level scheduler analyzes the run queues to choose which task should move from one core to another.

Instead of run queues, the CFS scheduler, shown in Figure 5-1, uses a red-black tree data structure (which is a derivative of the binary-tree structure) to form a future execution path of tasks. A red-black tree is roughly balanced, and has good deterministic properties such as insert and search is O(log *n*) to the number of elements (that is, tasks). Picking the next task to run is constant. The CFS is based on the concept of fair queuing in which no user should get more CPU time just because they run more tasks. For example, two users in a system running one task each get 50% of the CPU time. When user A starts a second task, they still get 50% of the CPU time, so their two tasks must now share that time, and, as a result, get 25% each. If an additional third user turns up, all users must share the CPU time equally and get 33% each with user A having 16.5% per task. The goal with CFS is to reach better core utilization in combination with interactive performance.

red-black tree

A binary search tree where each node has a red or black attribute that meets the following requirements:

1. A node is either red or black.
2. The root is typically black.
3. All leaves are black.
4. Both children of every red node are black.
5. Every simple path from a node to a descendant leaf contains the same number of black nodes.

Both O(1) and CFS make use of sleeper fairness, in which computation-intensive tasks are penalized relative to an IO intensive task that can execute only infrequently. This ensures it is given a small advantage because an IO-intensive task (sleeper task) needs the core infrequently.

5.1.2 Core Affinity

Although SMP systems have shared memory, which should allow any core to pick up any task at any time, there are still hardware factors that make some cores more or less well-suited than others for various tasks. For example, private caches have a natural affinity for a task to a specific core because the caches and branch tables are warm. This soft affinity prevents the task from being scheduled on another core unless there is a stronger force that pushes it there, such as an imbalance between the cores

It is also possible to manually set a hard affinity that forces a task to execute on a subset of cores. For each process there is a bitmask with one bit per core that determines which core the process can run on. A set bit for a given core indicates that the program is allowed to run on it, and a cleared bit indicates that it cannot. The Kernel API has two basic functions to access the mask: **sched_setaffinity()** and **sched_getaffinity()**, which are supported by a number of macros to create and decipher the mask (for example, **CPU_SET()**, **CPU_CLR()**, **CPU_ZERO()** and **CPU_SETSIZE()**). Example 5-1 shows a sample program that makes use of these macros to assign a hard affinity and then read it out. On an eight-core device on which the process can run on all but core 3, the output of this program would be “Process 35324 is allowed to run on core: 0 1 2 4 5 6 7.”

Example 5-1. Process Identity

```
#include <stdio.h>
#include <stdlib.h>
#define __USE_GNU
#include <sched.h>
#define MAX_CORES 8

int main()
{
         cpu_set_t mask;
         int core;

         /*** Write Affinity ***/
```

```
        CPU_ZERO(&mask);

        for(core=0; i < MAX_CORES; core++)CPU_SET(core, &mask);
        CPU_CLR(3,&mask);
        sched_setaffinity(0, sizeof(mask), &mask);


        /*** Read Affinity */
        sched_getaffinity(0, sizeof(mask), &mask);

        printf("Process %d is allowed to run on core: ",getpid());
        for(core=0; i < MAX_CORES; core++){
         if(CPU_ISSET(core, &mask) == 1)printf("%d ",core);
        }

        return 0;
}
```

From the command line, the user can also make use of the **taskset** command to display, alter, or start a process with a specific affinity. The user should be careful about using hard affinity to control task allocation because it limits portability and can lower performance in unexpected system situations. The scheduler often makes a better decision itself using the built-in soft affinity. However, it can be beneficial to give an application a private core to improve real-time behavior or to test how an application scales.

Interrupt request (IRQ) can also be assigned to specific cores. By default, all interrupts trigger core 0, but with large loads, this affects the core's ability to execute other applications. The affinity can be used to distribute the load among the cores just as with processes; for example, the Tx interrupts can be placed on one core and Rx on the other.

Example 5-2 provides a list of which interrupts are connected to which respective IRQs as well as how many times they have been triggered on each core by listing **/proc/interrupt**. It uses the Freescale QorIQ P4080, an eight-core machine. Now the serial interface should be altered and it is IRQ36.

Continue to **/proc/irq/36/** and list **smp_affinity** and see that it currently has the hexadecimal value 0xFF (0b1111_1111). This is again a bit mask, and as with processes, a 1 indicates that it can trigger on that core. The state can be changed by writing to **smp_affinity** by using, for example, the **echo** command. However, it is not possible to turn off an IRQ by writing all zeros to the mask; this setting is ignored.

Example 5-2. Interrupt–Core Assignments

```
/ # more /proc/interrupts

      CPU0  CPU1  CPU2  CPU3  CPU4  CPU5  CPU6  CPU7
36:   1390  0     0     0     0     0     0     0     OpenPIC  Level  serial
38:   10    3     0     0     0     0     0     0     OpenPIC  Level  i2c-mpc
39:   8     0     0     0     0     0     0     0     OpenPIC  Level  i2c-mpc
87:   5735  0     0     0     0     0     0     0     OpenPIC  Level  enet_tx
88:   0     0     5722  0     0     0     0     0     OpenPIC  Level  enet_rx
89:   0     0     0     8298  0     0     0     0     OpenPIC  Level  enet_error
251:  0     0     0     0     0     0     0     0     OpenPIC  Edge   IPI0 (call function)
252:  0     0     0     0     0     0     0     0     OpenPIC  Edge   IPI1 (reschedule)
253:  0     0     0     0     0     0     0     0     OpenPIC  Edge   IPI2 (unused)
```

```
/ # more /proc/interrupts
254:  0      0      0      0      0      0      0      0      OpenPIC  Edge   IPI3 (debugger break) BAD
/ #
/ #
/ #
/ # more /proc/irq/36/smp_affinity
ff
/ #
/ #
/ # echo 80 > /proc/irq/36/smp_affinity
/ #
/ #
/ #
      CPU0   CPU1   CPU2   CPU3   CPU4   CPU5   CPU6   CPU7
36:   1390   0      0      0      0      0      0      18     OpenPIC  Level  serial
38:   10     3      0      0      0      0      0      0      OpenPIC  Level  i2c-mpc
39:   8      0      0      0      0      0      0      0      OpenPIC  Level  i2c-mpc
87:   5735   0      0      0      0      0      0      0      OpenPIC  Level  enet_tx
88:   0      0      5722   0      0      0      0      0      OpenPIC  Level  enet_rx
89:   0      0      0      8298   0      0      0      0      OpenPIC  Level  enet_error
251:  0      0      0      0      0      0      0      0      OpenPIC  Edge   IPI0 (call function)
252:  0      0      0      0      0      0      0      0      OpenPIC  Edge   IPI1 (reschedule)
253:  0      0      0      0      0      0      0      0      OpenPIC  Edge   IPI2 (unused)
254:  0      0      0      0      0      0      0      0      OpenPIC  Edge   IPI3 (debugger break) BAD
/ #
/ #
```

5.2 Enea’s OSE for Multicore

Enea OSE® is a real-time operating system aimed at high-performance data plane applications, such as processing of user data packets and control signaling within both telecom- and datacom area. OSE is a truly distributed operating system that uses a message-based programming model that provides application location transparency.

This section discusses the Enea OSE architecture and its advantages over other multicore models.

5.2.1 Architecture Overview

Enea OSE is based on a micro-kernel architecture; see Figure 5-2.

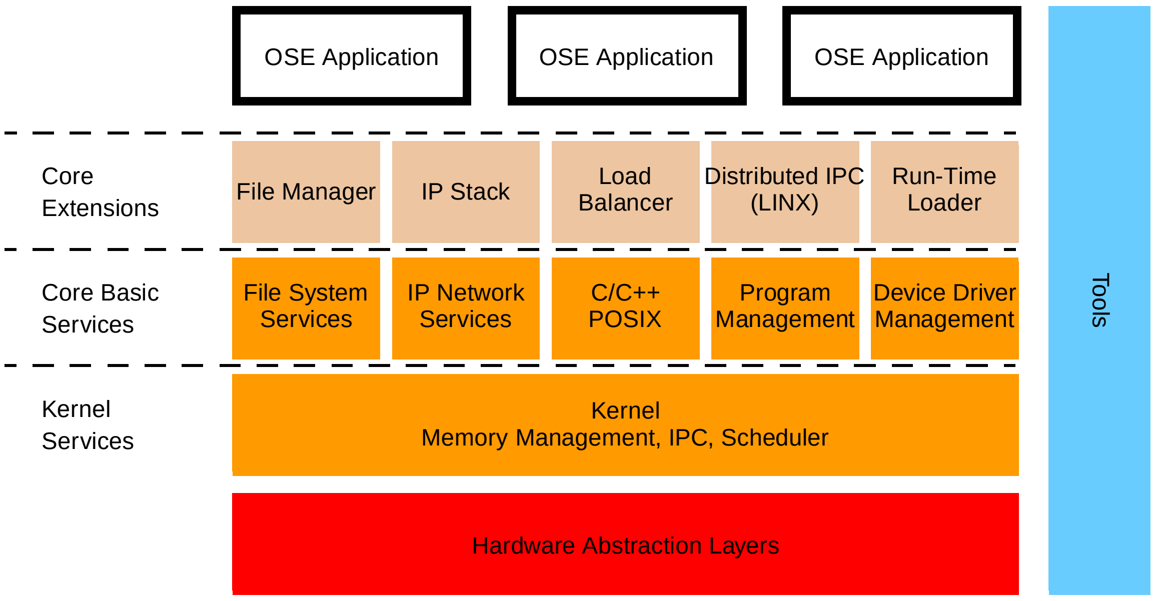


Figure 5-2. Micro-kernel Architecture

message passing

The hypervisor components of the Power Architecture define many messaging resources to provide communication from core-to-core and across hypervisor and guest-supervisor layers of the operating system. These are described in *EREF: A Programmer's Reference Manual for Freescale Embedded Processors.*

The OSE kernel is designed around the exchange of messages between processes, which in OSE is the equivalent to POSIX threads. This mechanism for inter-process communication (IPC) is the foundation of the OSE programming model, and it is implemented as a simple API that provides the ability to send messages between processes/threads in a distributed system running on a single, or several, processor nodes. OSE also provides an addressing model that enables application scalability, making it possible to let a system run on a single processor node or several nodes in a distributed cluster without changing the programming code.

When processors are physically on different devices, OSE kernels use the IPC protocol Enea® LINX for passing messages. LINX is a kernel concept used for the implementation of a message passing back plane that is adaptable for different media.

Services in OSE are mainly implemented according to the client-server model, providing a distributed C/C++ run-time library where parts of the POSIX API are included. Examples of such services are the File System Managers and the IP stacks. These services run on a single processor node, while the API is available to client applications on all processors via a C/C++ run-time function library that uses message passing to reach the operating system servers. An example of this is the call to *fread()* that use internal message passing toward the file system server process, which can be located anywhere in the system even when the system is spread geographically.

The OSE programming model encourages an object-oriented, parallel design of applications where each process uses its own private memory. In the OSE model the message passing is the main mechanism for exchanging data and for synchronization. Using the message passing programming model as the foundation for parallelization and synchronization makes the transition to the multicore technology significantly easier for the customers. An application that uses this model is already designed for distributed scalability, and therefore the migration to multicore devices becomes a straightforward task. This makes the software architecture of the customer systems future-proof in regards to the paradigm shift caused by the multicore technology; the applications can be reused without expensive software investments when the customer wants to use new hardware architectures.

5.2.2 Various Multicore Models

Designing an operating system for multicore processors can be carried out using a number of different approaches. When it comes to OSE, the most interesting models for multicore are the SMP model, AMP model, and the bare metal model.

SMP model	The symmetric multi-processing (SMP) model is the classic model for designing multicore operating systems, such as Linux, where data is shared to a large extent and where a number of different locking mechanisms and atomic operations are used frequently for synchronization. The SMP model is a very simple model from the application perspective, but its implementation is complex and difficult to make correct at the operating system level. Moreover, it does not scale very well to more than 4–8 cores, especially not on an application level.
AMP model	The asymmetric multi-processing (AMP) model uses an approach where each core runs its own isolated software system. These may even use different RTOSes. The advantages of an AMP system are that high performance is achieved locally and that it scales well. The trade off is that the system becomes difficult to manage and can become fairly static.
Bare-metal model	The bare-metal model is an approach for running a regular operating system on one or several cores and letting the rest of the cores execute a single thread without support from any operating system. Its advantage is that maximum performance is achieved when running without an operating system, but its disadvantage is that the software becomes hardware-specific, forcing a redesign of any applications when upgrading the hardware. The code running without an operating system must also be very simple because it does not have any OS-like programming or debug environment. The lack of visibility and debug ability therefore transforms the bare metal application into a black box.

5.2.3 Enea OSE Advantages

The Enea OSE®: Multicore Real-Time Operating System (RTOS) combines all the advantages of the above mentioned models without having to deal with the disadvantages.

Enea OSE: Multicore Real-Time Operating System (RTOS)
More information about this product can be found on the Enea webpages at www.enea.com

OSE Multicore RTOS is similar to SMP in terms of simplicity, flexibility, application transparency, and error tracing, and it is similar to AMP in terms of scalability and the lack of an extra load due to using several cores with shared memory. In other words, the performance level on each core is the same as on a unicore. Furthermore, an OSE application can be implemented in supervisor mode and therefore has similarities with the bare metal model in that the hardware can be accessed without involving the operating system at all except for interrupt handling. All together this allows for maximizing CPU resources for the application level and minimizing OS overhead.

The micro-kernel architecture and the message passing model allows common operating system services such as loaders, memory managers, IP stacks, and file systems to be located on different cores. Applications can then access these services regardless of location in the system, which gives a seamlessly shared resource model like in the SMP model.

The OSE kernel instantiates a scheduler on each core with associated data structures, achieving similarities with the AMP model. One important design goal with the separated kernel scheduler instances is to avoid the need for kernel-internal synchronous transactions from a thread running on one core towards the data structures that belongs to the scheduler on another core. This occurs typically when a thread executes a system call that needs to modify the state of another thread. Instead, a concept called “kernel event,” which is a low-level, light-weight-kernel, internal IPC, has been invented. It is used to perform various kinds of asynchronous, cross-core scheduler transactions. The goals are to do the following:

cache line bouncing

Describes what happens when threads on different cores frequently modifies the same data cache line. Every time data is written, the master data is moved exclusively to the L2 of the actual core and all other core’s corresponding L1/L2 cache line is invalidated according to the MESI protocol. This data can then exclusively bounce across the system and steal memory bandwidth and thus decrease performance. If the data the different threads want to modify is located within the same cache line, an even worse bouncing occurs, which is called false sharing. The hypervisor components of the Power Architecture define many messaging resources to provide communication from core-to-core and across hypervisor and guest-supervisor layers of the operating system. These are described in *EREF: A Programmer’s Reference Manual for Freescale Embedded Processors*.

- Avoid using spinlocks for synchronization in system calls that need to modify the scheduler data structures or process the control block. Instead, the much cheaper interrupt locking mechanism is used to synchronize the scheduler and interrupt handling locally on a core.
- Centralize the use of an optimized lock-free algorithm used to allocate and free message buffers from the global, shared pool.
- Avoid the effect of “cache-line bouncing” between L2 caches; it is kept on a minimum because system calls do not operate on other cores’ data structures except in rare cases.
- Avoid internal “false sharing” of data cache lines when accessing kernel data structures inside the RTOS.
- Optimize for low overhead in inter-core messaging by using asynchronous intercore kernel event queues. This ensures high application throughput performance.
- Use hardware support to implement the intercore kernel event queues where possible. For example, Freescale’s P4080 provides a programmable queue manager that can be used for this purpose.
- Provide full OS API (all OS calls and debug features) to applications on all cores. From the programming view and debug view, the operating system still looks like an SMP operating system. All threads on all cores are visible to the debugger. For example, it is possible to debug all threads on all cores using the Enea® Optima Tool.
- Achieve linear scalability to many cores, which is possible due to the asynchronous design.

A system with OSE5 Multicore Edition based on a Freescale eight-core P4080 might look like Figure 5-3.

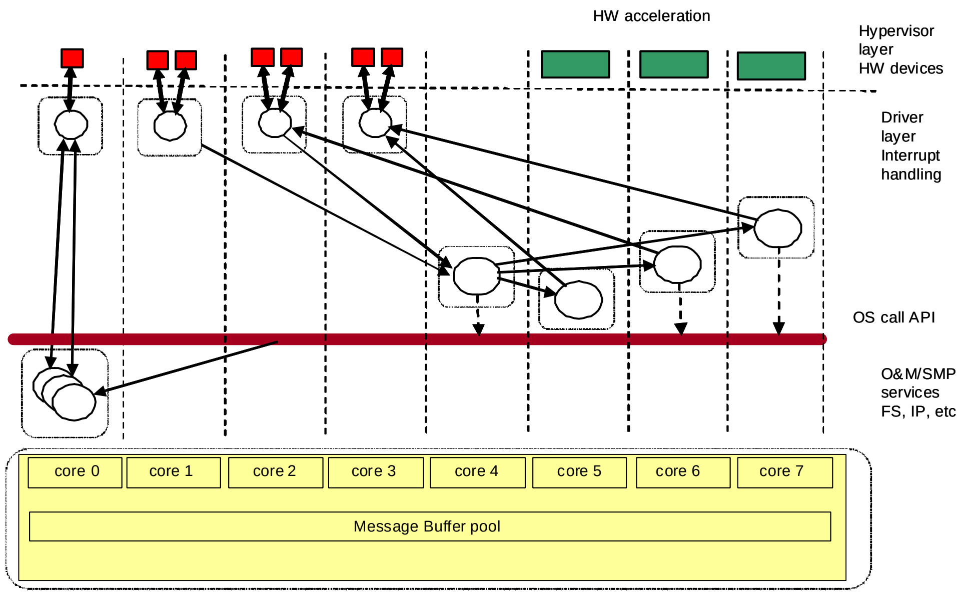


Figure 5-3. OSE5 Multicore Output Simulation

In the system in Figure 5-3, the scheduler on core 0 runs OS services such as File System or IP stack, and all other cores run the data plane application that is designed as a flow, or a “pipe,” between processes on different cores. Potentially cores 1–3 can terminate the gigabit Ethernet devices and pass the data on to a flow of IP processing either by zero-copy LINX messages or by using the hardware support for data transport. Cores 4–7 are running busy-looping processes/threads that consume data buffers, perform deep packet inspection, encryption, or decryption, and then pass the buffers on to the next process in the flow. Finally, a buffer is sent to one of the outgoing processes responsible for sending the packet out on a Ethernet device.

The sequence of a message send in Figure 5-3 contains the following steps:

1. A process/thread allocates a message buffer from the global message buffer pool.
2. The process performs the system call *send* specifying a destination address (PID) and a message buffer.
3. The send call reads the destination core from the destination address, and “posts” a kernel event containing the message buffer pointer and the destination address to the destination core. (Kernel event may use hardware accelerators to transport the event)
4. The destination core either takes an ISR to get the event or polls it in. This thread of execution then performs the core-internal transaction towards its scheduler data structure, which involves queuing the message buffer and the proper state transition. This transaction only uses the much cheaper interrupt lock synchronization for the reason described above.
5. The destination process/thread is then eventually scheduled, and consumes in the message buffer using the receive system call. It then either frees the message or resends it.

Note that all processes/threads in such an OSE Multicore Edition system can still make full use of the operating system services regardless of which core the process runs on. Because each scheduler is isolated, just like in an AMP system, most system calls can be designed without using spinlocks or atomic transactions that uses global shared data. In non-frequent cases a system call need to perform operations on global data structures global locking must be done using a BKL (Big Kernel Lock). An example is when a process is moved from one core to another.

5.3 Conclusion

Various software models have been discussed by the industry in order to address the multicore challenge, and all of them have benefits and drawbacks. Enea OSE5 Multicore Edition, having a kernel design based on an innovative combined SMP/AMP/“bare metal” technology, attempts to combine the advantages of them using the message passing programming paradigm both on an RTOS level and on an application level. Legacy applications that use this model of programming and parallelization have been proven to be easy to migrate to new multicore devices such as for example MPC8641D.

Enea OSE 5 Multicore Edition clearly challenges the industry-perceived difficulty to get linear scalability equal to AMP system performance and still be perceived as an SMP RTOS on the application level. When used on processors like Freescale P4080, the OSE kernel can also utilize hardware support to accelerate the inter-core transactions, which maximizes application performance. The Enea OSE 5 Multicore Edition architecture primarily targets the data plane application domain, including both user data processing and control signaling.

The Enea OSE 5.4 product, released in the end of 2008, contains initial multicore support according to the SMP model for Freescale’s MPC8641D and P2020. The release of Enea® OSE5 Multicore Edition, which will be released in Q2 of 2009, will be designed completely according to the architecture described above. This not only provides even higher performance for two cores, but also more or less offers linear performance scalability to devices with 8 cores or more, such as the P4080.

Chapter 6
Virtualization and the Hypervisor

—John Logan

This chapter explores the concept of virtualization and describes a software component, the hypervisor, which is used to enable virtualization on a processor. It also examines how embedded processor hardware has been developed to support virtualization.

It contains the following sections:

- Section 6.1, "Virtualization—An Overview," provides an introduction to virtualization, particularly its advantages for embedded multicore systems.
- Section 6.2, "Privilege Levels, Addressing and Exceptions," discusses the hypervisor level introduced in multicore devices.
- Section 6.3, "Hardware Features to Improve Virtualization," discusses hardware features that improve virtualization.
- Section 6.4, "Security and Protection between VMs," discusses strategies for limiting the damage caused by software bugs.
- Section 6.5, "Messaging Between VMs," discusses the different techniques for sending messages between virtual machines.
- Section 6.6, "Debugging and Run Control," discusses debug and run control in the context of embedded multicore systems, with a section on how to use the hypervisor for debug.

6.1 Virtualization—An Overview

In a system with a single SMP operating system, the OS has control of all cores, memories, and peripherals in the system. It provides all of the necessary scheduling, messaging, synchronization, memory management, and other services required to implement the complete system. However, in many embedded applications, especially with newer multicore processors, it is desirable to run multiple operating systems.

For example, in a network router application it can be advantageous to use a real-time operating system (RTOS) for the data plane processing—packet encryption/decryption, classification, forwarding—where low latency and predictable operation is required, and a general purpose OS, such as Linux, for the control plane processing—higher level protocols, such as ARP (address resolution protocol, a standard low-level protocol used to sync local IP addresses with Ethernet addresses), for which latency and timing are not so critical, but where a wide range of diverse tasks must be handled. A subset of the device's cores and memory can be assigned to each OS. But running multiple operating systems in a system raises new issues to address and new problems to solve. For example, how do the operating systems share the resources available on the chip (memory, peripherals, etc.)? Is it possible to communicate between operating systems?

Virtualization can be used to solve these problems. Virtualization is a computing concept in which an OS runs on a software implementation of a machine—that is, a virtual machine (VM). This VM does not have to have the same characteristics or features as the underlying hardware. Multiple VMs can run on a single hardware device, allowing multiple operating systems to coexist. The VMs are managed by a virtual machine manager, also called a hypervisor layer, which provides abstraction between the underlying physical hardware and the VMs. It can also provide communications between VMs if required as well as security and reliability (for example, one VM could crash but without affecting the rest of the system).

VMs and hypervisors have become common in network server consolidation. Rather than having separate units for a web server, file server, email server, etc., all running at low utilization, server suppliers are now looking at running servers within VMs and consolidating many systems on to a single multicore device. This reduces capital expenditure and running costs. Using VMs allows legacy code running under older operating systems to coexist with newer operating systems, removing the need to redesign code.

There are two types of virtual machine monitor, or hypervisor, as follows:

- *hosted*—the hypervisor runs as a program within an operating system. VMWare is a popular example. This tool allows a user with a Windows PC to run other operating systems (Linux, QNX Neutrino, etc.) within secure virtual machines without requiring complex dual-boot setups or multiple machines.
- *native*—the hypervisor runs directly on the processor. In an embedded application, a native hypervisor is preferred.

Figure 6-1 shows the relationship between VMs, the native hypervisor, and underlying hardware on an embedded multicore device. In this example, three VMs are running on a four-core device. Each VM has access to a subset of the devices' cores, memory and I/O. Each VM also has its own address space. The hypervisor layer provides the link between the real hardware and the VMs. All VMs can access some shared resources—cache, interrupt controller, and shared I/O—via the hypervisor.

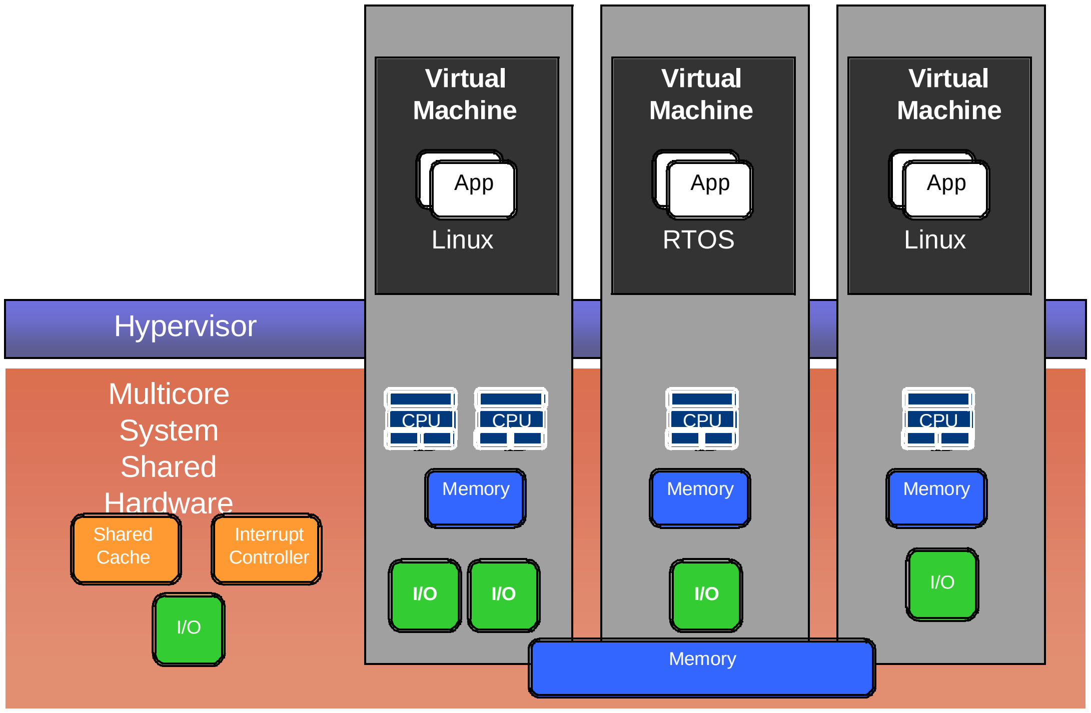


Figure 6-1. Relationships among VMs, the Native Hypervisor, and Underlying Hardware on an Embedded Multicore Device

Note that in this example, the VMs map to real hardware on the device: the VM with two cores uses two real cores on the device. It is possible for the VM to have a completely different architecture than the underlying hardware; for example, one could create a VM that thinks it is running on a four-core device when in reality there is only a single physical core. It is also possible to run multiple VMs on a single core. However, both of these options incur software overhead and are typically not seen in embedded applications.

A native hypervisor for an embedded application can provide the following features:

- VM management—creating, removing, starting, and stopping VMs
- Security among VMs
- Messaging among VMs
- System-level event handling—memory mapping, interrupt routing, etc
- Debug support

Before examining these features, we need to look at some hardware concepts required for efficient virtualization in an embedded system.

6.2 Privilege Levels, Addressing and Exceptions

On a typical single- or dual-core processor there are two privilege states: user and supervisor. Applications run in user state whereas the operating system runs in supervisor state. System-level events, such as handling interrupts and memory mapping, are handled by the OS and require the processor to be in supervisor state. This is to prevent individual applications from compromising system integrity and crashing the entire system by, for example, code runaway corrupting the memory map of other applications.

A multicore device running a hypervisor needs three privilege levels, the user or problem state for applications, supervisor state for an operating system within a virtual machine, and hypervisor state, a higher third-level for the hypervisor. System-level changes, such as memory mapping and allocation of dedicated peripherals, occur at the hypervisor state.

privilege levels

These are set by the OS for applications and by the hypervisor for OSes. Changes in the privilege level typically occurs at interrupts.

For example, imagine a user program running under a Linux OS within a VM. Such a system requires three address spaces—one for the user program, one for the VM (and the Linux OS), and one for the physical memory map of the device.

The user program, running in user mode, attempts to write data to a memory location. When this happens, the processor checks for the requested memory address in its translation lookaside buffers (TLBs). The TLB converts this memory address from the user program’s address space to the VMs address space. The first time this memory location is requested, the processor does not have an entry for it in the TLBs. This generates a TLB miss exception (a supervisor-level exception), which is handled by the Linux OS. See Figure 6-2.

TLB

Translation lookaside buffers. Used to speed up virtual memory systems.

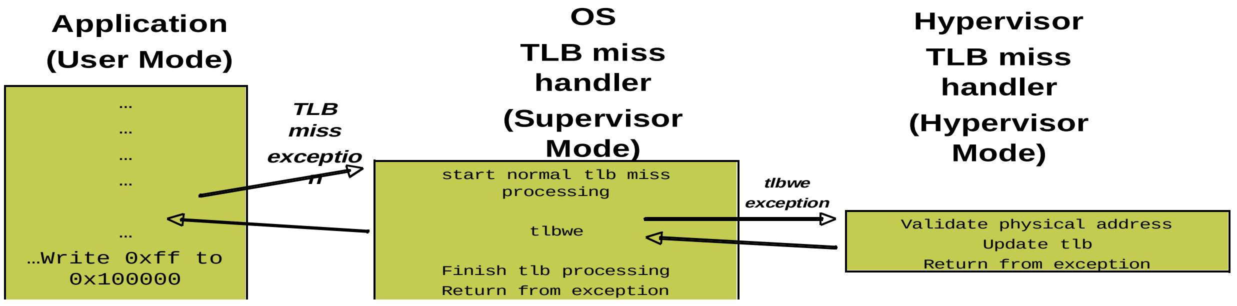


Figure 6-2. TLB Miss Handler

The Linux OS, running in supervisor mode, handles the exception and runs the routines to update the processor TLBs to point to the memory location. The Linux OS also determines which TLB entry can be used for the update, figures out the address translation between the VM address space and the user program’s address space, and performs the TLB update, usually by executing opcode (for example, the TLB Write Entry (**tlbwe)** on a Power Architecture processor).

At this point, the original request has still not been mapped to a physical address, which must be done by the hypervisor. To make this happen, the **tlbwe** instruction generates another exception (hypervisor-level exception) that is handled by the hypervisor. Because the hypervisor has a higher privilege level, it begins processing this exception before the TLB exception completes. The hypervisor can do the final translation from the VM's address space to the physical address space of the device, put these values in the processor's TLB, and then return control to the VM.

This process can be done in such a way that Linux is completely unaware of the hypervisor update. This means that users can easily take a Linux OS and applications running on an SMP system today and run it in a VM on a multicore, multiple-VM system without needing to change the operating system's memory-mapping routines.

This principle of using exceptions to jump to the hypervisor to handle system-level events can be extended to external interrupts, too, which reduces the amount of recoding needed when moving to a multiple-VM environment.

6.3 Hardware Features to Improve Virtualization

For efficiency, the hypervisor software layer should be as thin as possible. Typically, it runs code when a VM needs to access a resource that could be shared with another VM—memory, peripherals, debug ports, etc. Each time the hypervisor runs, it adds software overhead. Hardware support can reduce this overhead.

For example, if the device has multiple memory controllers, each VM can have its own. Each VM can also have dedicated (that is, non-shared) peripherals. For resources dedicated to a virtual machine (that is, that machine has sole access to them), there is no need for the hypervisor to be called.

Another approach is to provide a level of abstraction between the cores and peripherals. One way to achieve this abstraction is with queuing mechanisms. Rather than each VM trying to write directly to each peripheral, each VM can access send and receive queues. The queue contents are routed to and from peripherals using dedicated hardware.

Figure 6-3 shows an example. Each VM has an associated queue that can be accessed with a driver. The interface to the queue can be a memory-mapped portal where data can be written to or read from the queue. Data can be sent to/from any I/O block connected to the queuing hardware to/from any VM with a connection. The queuing management hardware takes care of any contention issues. With this kind of platform, it is possible to develop families of devices with varying numbers of cores and peripherals and make software that easily ports across the whole family.

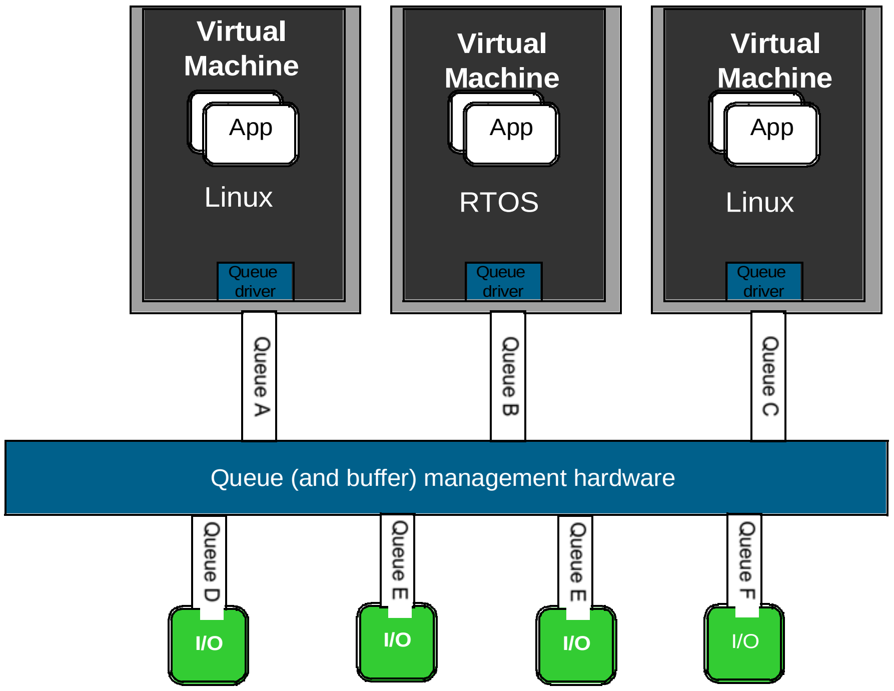


Figure 6-3. Abstraction Of Peripherals Using Queues

Freescale's P4080 eight-core device is an example of a device implementing such a scheme (see Figure 2-1 for a block diagram of the P4080 processor). It has sophisticated hardware queue and buffer management that handles all data traffic to/from its Ethernet ports. It also has parsing and classification hardware to allow the Ethernet ports to examine incoming packets and route them to the appropriate virtual machine.

6.4 Security and Protection between VMs

Bugs are a fact of life for any embedded developer. In a single-core multithreaded environment, an OS provides some protection against bugs in one application causing a crash in another. For example, the OS typically provides each user application with its own virtual address space, provides memory allocation/deallocation functions, prevents user programs from directly manipulating device registers (forcing the use of drivers/API instead), provides scheduling so that one program does not block others from running, and more. Of course, the determined (or inexperienced) programmer can find flaws in these setups.

In multicore systems running multiple VMs, the hypervisor can provide similar protection between different VMs to prevent a crash in one VM from causing a crash in another. For example, imagine writing a Linux driver that runs in supervisor mode. A bug is in the code, and the driver starts writing values to memory randomly. It tries to write to a physical memory location that is assigned to another VM. The hypervisor can detect these writes and take appropriate action to prevent them, as discussed in the virtualization example above. With this kind of protection, an entire VM can crash, but it is prevented from corrupting the memory space of another VM due to erroneous writes from the processor cores.

In an embedded processor, however, cores are not the only blocks capable of corrupting memory. Many integrated peripherals can act as bus masters and move data around the memory map, such as DMA controllers and Ethernet and RapidIO network interfaces. A bug in the software can set up one of these peripherals to write to memory assigned to a different VM. This can be prevented in hardware by making peripherals aware of the particular memory ranges they can access. For example, the Freescale P4080 processor has a feature called peripheral access management unit (PAMU), which allows peripherals to be assigned different access rights to programmable ranges in memory. This is similar in concept to the memory management unit (MMU) in a core.

6.5 Messaging Between VMs

In a system with multiple VMs, there is often a need to send information between VMs, for example to synchronize events or signal a problem. This can be done using various methods, depending on the device design. On devices with network connections such as Ethernet controllers, users can send messages using the network as the path. On devices with hardware abstraction features, such as the queuing mechanisms on the Freescale P4080, users can use the queues as a message path. The hypervisor can also provide messaging. Cores can have dedicated instructions allowing data to be passed from core to core using the cores internal registers. In the case of the Freescale P4080 device, each core can make a request to the hypervisor to send a message. The hypervisor uses Message Send, **msgsnd**, and Message Clear, **msgclr,** instructions to send messages between cores. These instructions can be executed only at the hypervisor privilege level to prevent their use by malicious software.

6.6 Debugging and Run Control

There are several standard ways to debug a single-core or single-OS-based system. The user might use a hardware debug cable that connects to a debug port (typically a JTAG based port) on the processor, or on a target device, the user might run a debug monitor program that transfers debug info on a serial port or network connection. There is also usually a need for a simple console or a command-line interface for simple debugging and system configuration. For example, in embedded Linux applications, the classic interface is a simple UART connected to a terminal emulator

JTAG

Joint Test Action Group. An interface initially developed to test PCBs, but now commonly used for low-level debugging devices.

In a multicore, multiple-VM environment, each VM requires debug support. This can be achieved by replicating debug hardware in each core, so that each core has some basic run control, breakpoint, and tracing functionality. Providing a means to access this debug functionality without requiring a separate set of debug pins per core is desirable—imagine how many pins would be needed on a 128-core device!

In JTAG-type debug setups, each device core can be seen as an individual unit on a single JTAG chain and can be debugged separately. The device does not need to have an individual JTAG connector for each core; they can all be accessed through one port.

Similarly for software debug setups, each VM could send/receive debug info on a network interface, for example an Ethernet interface. Each VM could use an unique IP address for debug and would require only one physical Ethernet controller.

Some debug connections are not so easy to multiplex, such as the classic UART connection. This is where the hypervisor can help by providing virtual serial ports for each VM and multiplexing this information to a single physical serial port.

All of the above options imply having a separate debug session or connection to each VM. Another option is to have a single debug session that connects to the hypervisor. It receives all debug commands for all VMs and passes them to the correct VM. Because the hypervisor runs at a higher privilege level than the VMs, it can access their debug functionality directly.

Freescale's embedded hypervisor for the QorIQ family of processors provides all of the debug functionality mentioned above and also additional hardware to allow streaming of debug data by means of a high-speed SerDes interface. Figure 6-4 shows a block diagram of the QorIQ processor family concept.

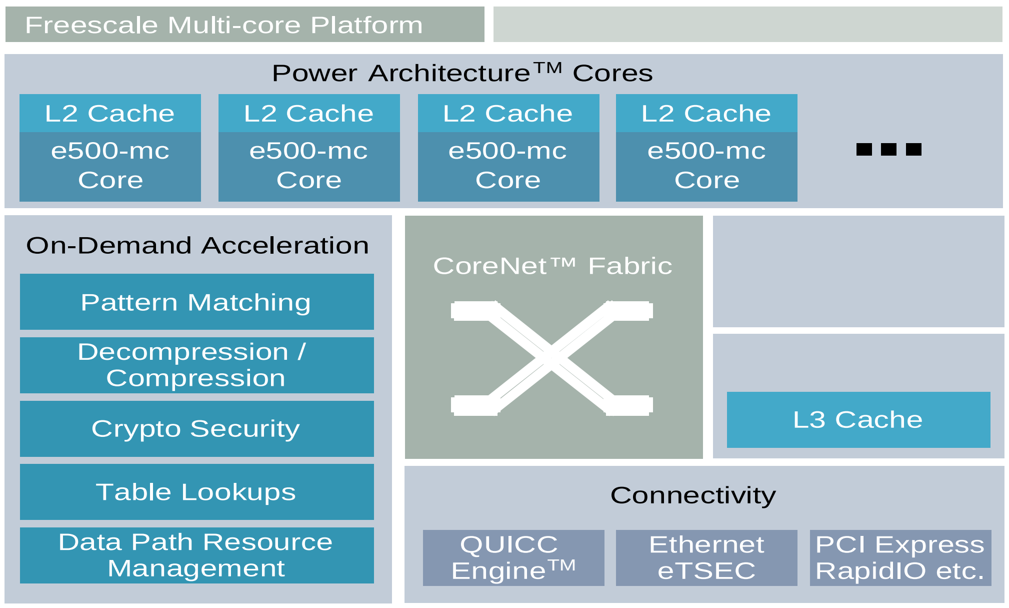


Figure 6-4. QorIQ Processor Family

6.7 Conclusions

In this chapter, we discussed the concept of virtualization and how a hypervisor software layer coupled with some hardware features can support it efficiently. As multicore devices become more prevalent in embedded designs, software developers will increasingly use virtualization to help them partition applications across these complex devices. Additionally, device manufacturers will develop new hardware features and architectures to keep pace.

Chapter 7
Embedded Multicore: Virtual Platforms

—Jakob Engblom

The shift to multicore processors and multiprocessor software systems calls for new software tools to help developers transform their code into parallel applications. Traditional debugging techniques and debugging tools do not work very well on an inherently nondeterministic system. Virtual platform technology reintroduces control and determinism to the software debug and analysis process, even for multicore processors. Virtual platforms also make it possible to develop software before the physical hardware becomes available, and they are an important tool for very early performance assessment and analysis of the hardware design.

This chapter describes how a virtual platform is essentially a simulation of a computer system, and the ways in which such a system provides opportunities for modeling, design, and debug in the context of multicore computer systems.

- Section 7.1, "Simulation of Computer Systems," describes the historical role of simulation in scientific and technological advancement and the increasingly important role of simulation in complex multicore environments.
- Section 7.2, "Obtaining Hardware Early," describes how virtual processor simulations provide an environment for designing with complex multicore processor devices long before the integrated device is physically available.
- Section 7.3, "Using a Virtual Platform for Debugging," describes the advantages that a virtual platform brings to debugging and presents a real-life example.
- Section 7.4, "Multiprocessor Software Debugging," describes the different types of simulation, their purposes and scope, and how they are used in the design, development, and system-level integration of cores and SoCs.
- Section 7.5, "Using Virtual Platforms for Hardware Design," describes different types of simulation, their purpose and scope, and how they are used in the design, development, and system-level integration of cores and SoCs.

7.1 Simulation of Computer Systems

Simulation has been used as a tool since ancient times in many areas of science and technology. Weather prediction, virtual car crash tests, aerodynamic modeling, and other physical system simulations use computers to model and understand the behavior of the physical world.

Simulation is used whenever trying things in the physical world would be inconvenient, expensive, impractical, or impossible. It allows experimenters to try things with more control over input data and parameters and to gain better insight into the results and system states on the way to those results. It reduces the cost of experiments and enables work with systems that do not yet exist in physical form. It cuts lead times and improves product quality. In a sense, we use simulation because reality is no fun.

For many of the same reasons, computers also can be used to simulate other computers. In the embedded systems space, such simulations are known as *virtual platforms* and are used to develop and test software independently of hardware availability.

virtual platform

Simulated computing environment used to develop and test software independently of hardware availability

Figure 7-1 shows what we want to achieve from a virtual platform: a piece of software that mimics the hardware so that the target software can run. This is not just an academic exercise in hardware–software equivalence, but rather a very useful tool for software developers, hardware designers, and system integrators. Virtual platforms provide the same benefits as simulation of physical systems do, making it possible to develop systems faster and with higher quality. As the complexity of embedded systems increases, virtual platforms are increasingly important as a system development tool.

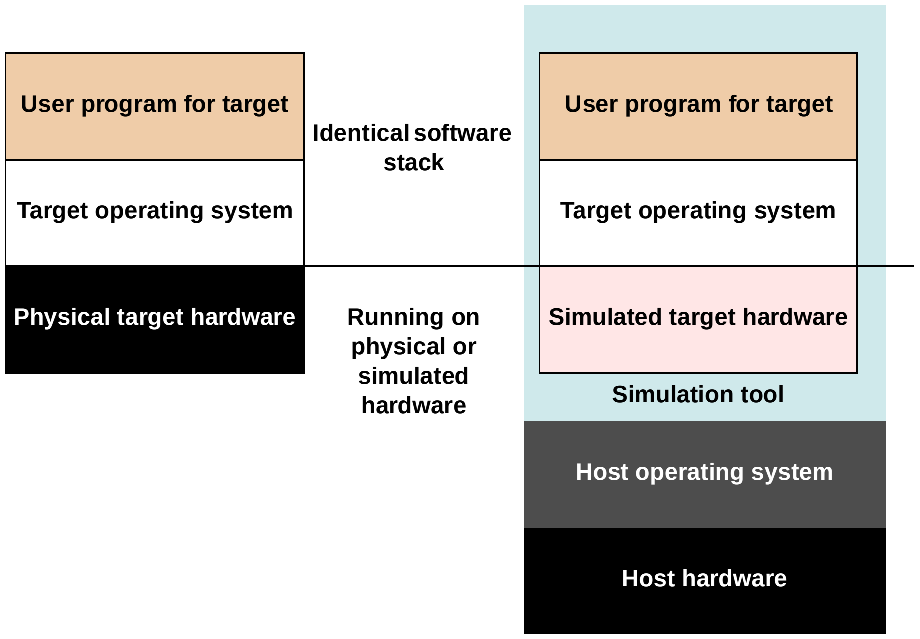


Figure 7-1. Virtual Platform Running the Same Software as the Physical Hardware

Using a virtual platform to develop the target software for a system requires a very fast simulator that can execute programs approximately as fast as a physical machine would. No user would wait hours for an operating system boot that takes seconds on a physical machine. The trick is to gain execution speed and reduce simulator implementation time by reducing the level of detail in the simulator to the bare minimum needed to run the software. We will return to the topics of simulation speed and abstraction levels. First, let us look over the benefits that virtual platforms can bring to software and systems development.

7.2 Obtaining Hardware Early

The most obvious benefit of a virtual platform is that it is available for software developers much earlier than the physical hardware. As Figure 7-2 shows, creating a virtual platform of hardware allows starting software development tasks earlier, which shortens time-to-market.

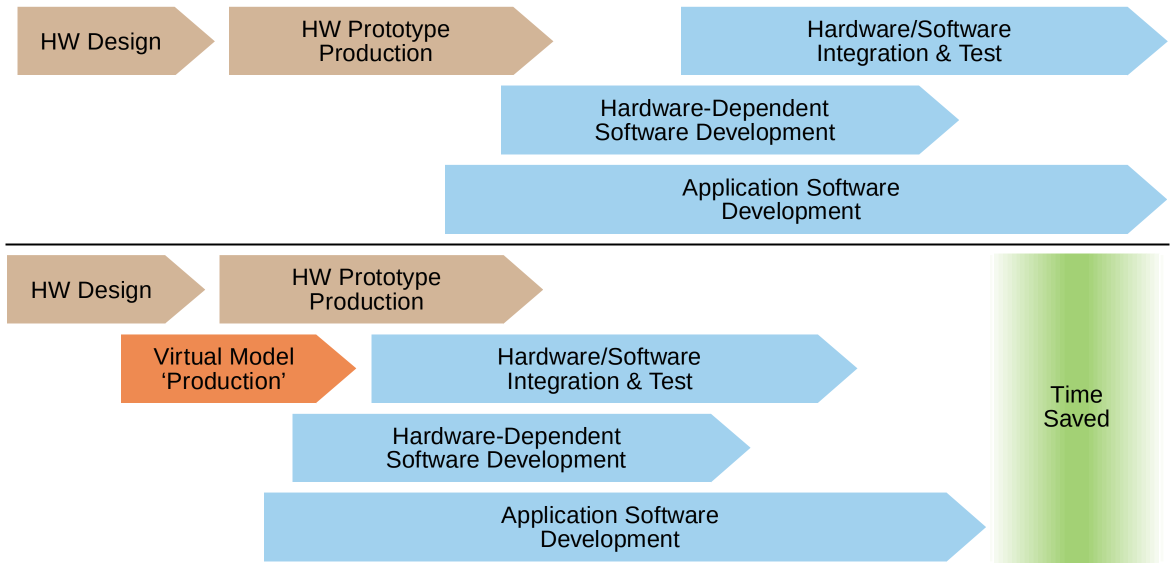

HW Design
HW Prototype Production
Hardware/Software Integration & Test
Hardware-Dependent Software Development
Application Software Development
HW Design
HW Prototype Production
Virtual Model 'Production'
Hardware/Software Integration & Test
Hardware-Dependent Software Development
Application Software Development
Time Saved

Figure 7-2. Virtual Platforms Make the Hardware Available Sooner

A second benefit is that virtual platform availability is not affected by unexpected delays in physical hardware availability. This reduces the risk that hardware delays will affect the final product's shipping date. It is the experience of Virtutech and others within the industry that between three months and eighteen months are typically saved in time-to-market when virtual platforms are used to provide early access to new hardware generations.

Furthermore, software development can ramp faster because hardware availability is no longer a constraint on the development process. Every engineer can have a hardware board very early, at their desk, inside their PC. Usually, the virtual platform remains in use as a virtual hardware system, long after hardware physical prototypes start to appear, because they are more convenient to use, offer better insight than the physical hardware, and are easier to configure.

7.3 Using a Virtual Platform for Debugging

A primary benefit of a virtual platform is that it provides superior debug and analysis features compared to the physical hardware. Anyone who has ever developed code for an embedded board can appreciate the convenience of a virtual environment: it provides a system that is not randomly flaky and that offers better control over the target, faster communication, and conveniences such as unlimited numbers of hardware breakpoints, deterministic repeatability of execution runs, and reverse execution and debugging. If the target freezes completely, you can stop it and check what happened. You can change system parameters, such as clock speeds and memory size, as well as network setups, with complete freedom and ease. For an idea for how this works in practice, here is a real-world example of debugging a multiprocessor system with Virtutech Simics:

> A virtual platform based on Simics was used to port a popular real-time operating system to the Freescale MPC8641D multicore processor. In one test, the clock frequency of the target system was changed from 800 to 833 MHz, and suddenly the system froze early in the boot process. The system was completely unresponsive, with no input or output.
>
> A preliminary investigation revealed that the problem occurred only between 829.9 and 833.3 MHz. Thus, it had not been seen before, because the clock frequency had been 800 MHz.
>
> Thanks to the repeatability of a virtual platform, the bug was trivial to reproduce. Each time the virtual platform was booted with one of the bad clock frequencies, the same crash happened at the same time. Unlike the real world, where all we would have had was an unresponsive brick, the virtual platform made it possible to examine the state of the processor, memory, and software at the point where the processor froze.

To home in on the problem, we used reverse execution and interrupt tracing on the serial port, the interrupt controller, and the processor cores. With this, we could pin down the exact cycle in which the problem occurred and the sequence of events that lead up to it. We did stack back traces at the critical point to determine the locations in the operating system where the freeze occurred.

In the end, it was determined that the problem was caused when an interrupt service routine attempted to lock a kernel spinlock before re-enabling interrupts. In the case that froze, the lock had already been taken when the service routine was entered, and with no interrupts enabled, there was no way for any other code to run to release it.

The bug was found only because the virtual platform ran the complete real software stack, including interrupt handlers and hardware drivers. It was triggered by changing the system configuration, demonstrating the value of configurability of a virtual platform. Because of the repeatability of the virtual platform, bug reproduction was trivial whereas in a physical system, it would have happened only occasionally. The ability to trace and inspect any part of the state was crucial to understanding what happened and in which order. With reverse execution (see Figure 7-3), we simply backed across the freeze to inspect the path that the system took to get there and were also able to move back and forth over the execution path to investigate it.

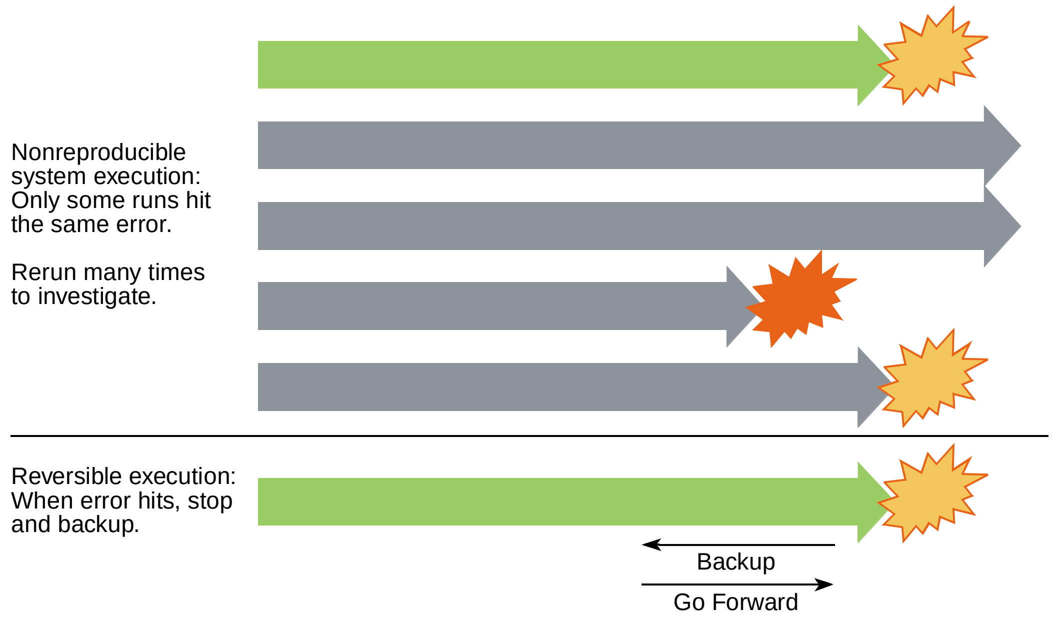

Nonreproducible system execution: Only some runs hit the same error.
Rerun many times to investigate.
Reversible execution: When error hits, stop and backup.
Backup
Go Forward

Figure 7-3. Reverse Execution

Note that we used a fast virtual platform and still triggered and solved a race-condition bug. This is the general case. Most multiprocessor software bugs are due to logic errors in the code, and they do not require timing to be identical to physical hardware to trigger. Indeed, the key to finding bugs is not to faithfully reproduce the actual behavior of a particular configuration of a physical machine, but rather to explore a large range of potential behaviors. This requires configurability and execution speed, rather than a high level of detail.

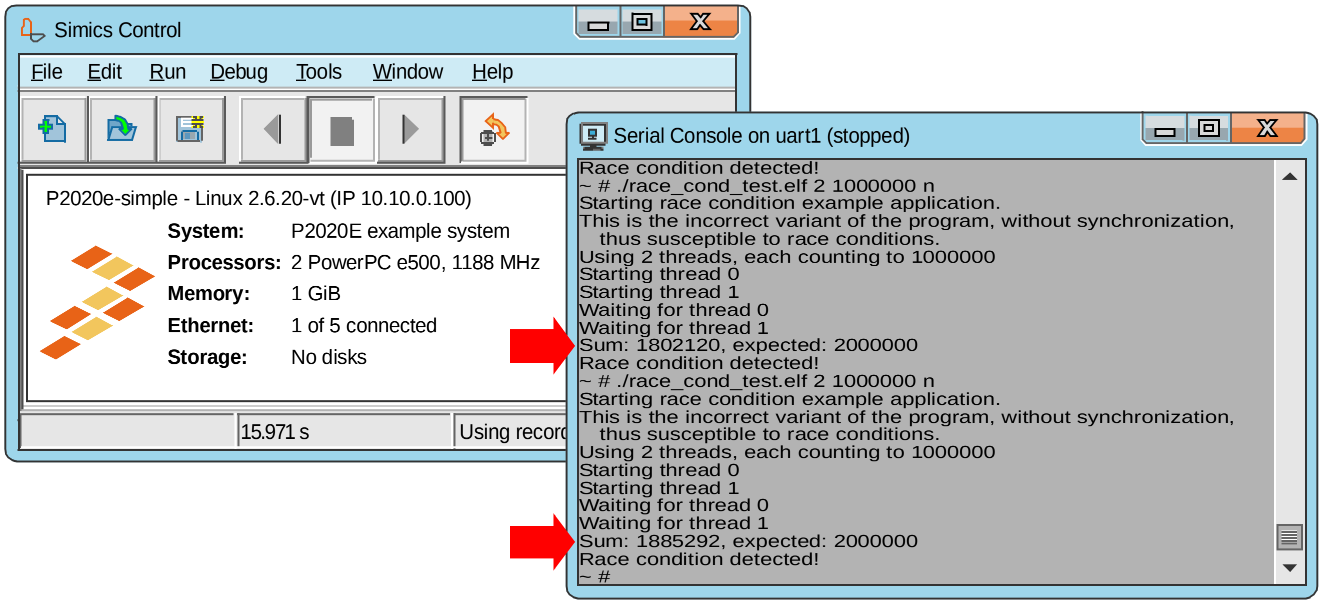
7.4 Multiprocessor Software Debugging

Especially for multicore and multiprocessor systems, virtual platforms provide a much needed boost to inspection and debug power.

The main problem in identifying and fixing software bugs in parallel software is the lack of determinism in the execution of the software system. Each run of a program exhibits a different order of events in the program, and even very small timing changes to the system state or timing result in very different program execution. This complicates debugging, as the very act of debugging a parallel program makes timing-sensitive bugs such as race conditions disappear or appear in a different place.

A virtual platform provides determinism and repeatability. The simulator has explicit control over the execution of instructions and propagation of information between processors, and can thus impose a repeatable behavior on the software running on a multicore processor. Determinism does not mean that the behavior of a software program is always identical; it means that when running the same software from the same initial state with the same sequence of asynchronous inputs, the same execution sequence is seen. If anything changes, a different behavior is seen.

Figure 7-4 shows an example of this change, where the same intentionally buggy program is run several times on a simulated multiprocessor. Each run produces a different result because they are run from different initial states. The simulator can go back and reproduce each run, which is not possible on physical hardware.



P2020e-simple - Linux 2.6.20-vt (IP 10.10.0.100)

System:	P2020E example system
Processors:	2 PowerPC e500, 1188 MHz
Memory:	1 GiB
Ethernet:	1 of 5 connected
Storage:	No disks

```
Race condition detected!
~ # ./race_cond_test.elf 2 1000000 n
Starting race condition example application.
This is the incorrect variant of the program, without synchronization,
   thus susceptible to race conditions.
Using 2 threads, each counting to 1000000
Starting thread 0
Starting thread 1
Waiting for thread 0
Waiting for thread 1
Sum: 1802120, expected: 2000000
Race condition detected!
~ # ./race_cond_test.elf 2 1000000 n
Starting race condition example application.
This is the incorrect variant of the program, without synchronization,
   thus susceptible to race conditions.
Using 2 threads, each counting to 1000000
Starting thread 0
Starting thread 1
Waiting for thread 0
Waiting for thread 1
Sum: 1885292, expected: 2000000
Race condition detected!
~ #
```

Figure 7-4. Example Output from a Multicore Run on a Virtual Platform

Another benefit of a virtual platform for multicore debugging is that the simulator can stop the execution of the entire system at any point, making it possible to single-step code where processors communicate with each other without changing the behavior of the code. Code running on other processors cannot swamp a stopped processor with data to process.

7.5 Using Virtual Platforms for Hardware Design

Virtual platforms and the simulation of key components is an important tool for computer hardware designers, both for processor cores and entire system-on-chip (SoC) designs.

In processor design, detailed simulation models of the pipeline and memory system of a processor core have been the mainstay architecture tool for many years. Every microarchitectural idea is first evaluated with the help of a detailed simulator before being used in an actual design. When creating new cores or new variants of cores (like the Freescale e500mc core), the design teams use their own detailed simulators to assess performance, power consumption, and other metrics.

Moving from processor cores to complete SoC designs, virtual platforms are used to evaluate interconnect architectures, required bus widths, and other performance factors. They are also used to verify that the design works as intended when isolated devices and processors are combined to form a whole system. For multicore designs in particular, users must validate cache-coherency protocols and check that systems scale to the number of cores desired.

The simulation models used for hardware design are very detailed because they need to depict the precise cycle-by-cycle execution at a system level. The models take time to write and to run, but in return users obtain a tool that lets them collect data that would be impossible to collect on physical hardware and that provides modeling opportunities for architectural experiments.

Usually, computer architecture simulators are internal engineering tools. However, for the Freescale QorIQ P4080, Virtutech and Freescale have collaborated to package the clock-cycle level internal simulators into a user-accessible simulation system. This *hybrid system* combines the fast functional simulators and the slow but detailed clock-cycle level simulators to give end users access to the detailed target timing. The fast virtual platform is used to boot operating systems and position workloads in interesting locations, and then the simulation can be switched over to the clock-cycle level models to allow detailed studies of software and system performance. In essence, the resulting combined solution lets users zoom in on performance details when and where they need to without compromising on the ability to run large workloads.

fast functional model

Simulation model used for functional testing and verification

cycle accurate model

A slower, more detailed simulation that can collect performance data

hybrid system

Allows for a switch between cycle accurate and functional models to obtain performance data as needed

7.5.1 Execution Speed

Because detailed performance models obviously provide more information, one might ask why they are not the norm. There are two main reasons: It takes much longer to build a detailed model than a fast model, and detailed models are too slow to run large workloads. The latter problem of execution speed is the most important reason. Table 7-1 shows execution speeds at different levels of timing detail. Note that virtual platforms can be faster than physical hardware.

Table 7-1. Simulation Speeds

Simulation Detail Level	Typical Slowdown	Indicative Speed In MIPs	Time To Simulate One Real-world Minute
Gate-level simulation	1000000	0.002	2 years
Clock-cycle level	10000	0.2	7 days
Virtual prototypes	5	400	5 minutes

Table 7-2 shows statistics that provide an idea of the workload sizes. Workloads very quickly get into billions of instructions, even when we are looking at single processors. For multiple processors in a multicore device, these numbers have to be multiplied accordingly.

Table 7-2. Workload Sizes

Workload	Number of Instructions
Booting Linux 2.4 on a simple StrongARM machine	50 million
Booting a streamlined but full-featured real-time operating system on a PowerPC 440GP SoC	100 million
Booting Linux 2.6 on a single-core MPC8548 processor	1,000 million
Booting Linux 2.6 on a dual-core MPC8641D processor	3,600 million
Running 10 million Dhrystone iterations on a single-processor UltraSPARC machine	4,000 million
Running one second of execution in a rack containing 10 boards with one 1-GHz processor each.	10,000 million

Thus, for modern multicore chips, we need virtual platforms to be as fast as possible in order to run the complete software in reasonable time. To accomplish this, we make the trade off of reducing the detail level of the virtual platform model. It is better to cover the whole problem in some detail than a tiny part of the problem in great detail. The vast majority of software development can be performed on a fast virtual platform with approximate timing, and with a hybrid model, clock-cycle level timing can be applied when and where it is needed.

7.6 Conclusion

Simulation is an established methodology in many fields of engineering, one that should also be applied to software and systems engineering for embedded systems. With the shift to multicore processing, simulation technology in the form of virtual platforms offers very attractive capabilities for software development, testing, debug, and optimization. Virtual platforms can run the real software stacks, well in advance of the physical hardware availability and through the entire system life cycle, being a useful tool all the way to full-scale system development and maintenance.

About the Authors

Jonas Svennebring is a Software Field Applications Engineer at Freescale in Stockholm, Sweden. He holds a Master of Science in Physics from University of Stockholm and has written academic and technical papers in the field of mobile robotics as well as embedded systems. Jonas’ focus is on applications within the wireless infrastructure domain, both using general purpose as well as signal processing devices.

John Logan is an Applications Engineer for Freescale Semiconductor’s Networking and Multimedia Division, based in East Kilbride, Scotland. John supports Freescale’s EMEA netcomms customers with a special focus on PowerQUICC and QorIQ coimmunication processors.

Jakob Engblom holds a Ph.D. in computer systems from Uppsala university and is currently a technical marketing manager at Virtutech in Stockholm. Jakob’s professional interests include embedded systems, embedded software development, multiprocessors, simulation technology, computer architecture, and compiler technology. He has authored more than 40 papers in the field of real-time and embedded systems, and is regularly presenting at embedded trade shows and conferences.

Patrik Strömblad is the Chief Architect for the OSE5 product line at Enea in Stockholm, Sweden. He has been designing distributed real-time operating system kernels for 20 years, mainly focusing on the telecom application domain, both within the infrastructure and the mobile market.