ASTRIUM
AN EADS COMPANY

SIDMS FINAL REPORT

• Configuration : ACE7.TN.14924.ASTR • Issue: 1 • Rev.: 0 • Date: 11/22/2011

SYSTEM IMPACT OF DISTRIBUTED MULTI CORE SYSTEMS

Final Report

	NAME AND FUNCTION	DATE	SIGNATURE
Prepared by	**Mathieu Patte** (Astrium SAS) *SIDMS study manager* **Vincent Lefftz** (Astrium SAS) *Technical Expert*		
Verified by	**Vincent Lefftz** (Astrium SAS) *Technical Expert*		

ESTEC Contract 4200023100

NbCars : 61143
NbWords : 11373

Abstract

The main objective of the “System Impact of Distributed Multi core Sytems” study was to analyse the system level impacts of the introduction of multi core processors for European space missions. This document is the final report of the study. In this document we will present a survey of existing multi core architecture, and based on this survey perform an analysis of the NGMP hardware. We will then focus on software techniques specific for multi core programming. In the last section, we will propose two use cases for multi core usage in space missions and provide recommendations on the best way to implement these use cases.

In the frame of this study, the Xtratum hypervisor has been ported on top of the NGMP hardware. We will present the multi core specific features that have been implemented to optimize the implementation of partitioned systems.

Also in the last section, we will discuss the way forward, and identify several key topics which in our opinion should be worked on in order to ease the introduction of multi core processors.

The System Impact of Distributed Multi core Systems project supported by the consortium made of Astrium Satellites, Universidad Politecnica de Valencia is called SIDMS hereafter.

Document change log

ISSUE/ REVISION	DATE	MODIFICATION NB	MODIFIED PAGES	OBSERVATIONS
1/0	11/22/2011			

NbCars : 61143
NbWords : 11373

Table of contents

Reference Documents

[LEON4-NGMP] *Quad Core LEON4 SPARC V8 Processor. LEON4-NGMP-DRAFT. Data Sheet and User's Manual.* http://microelectronics.esa.int/ngmp/LEON4-NGMP-DRAFT-1-8.pdf

[SIDMS SOW] System Impact of Distributed Multicore Systems Statement of Work, Reference: TEC-SWE/09-249 – Issue 1, Revision 5 – 09.07.2009, Appendix 1 to AO/1-6185/09/NL/JK

[EBD_MultiCore] Embedded Multicore:An Introduction Freescale July 2009 http://www.freescale.com/files/32bit/doc/ref_manual/EMBMCRM.pdf

[MCAPI] Multicore Communications API Working Group web site http://www.multicore-association.org/workgroup/mcapi.php

[GO_LANG] The Go Programming Language: http://golang.org/

[OpenMP] The OpenMP API specification for parallel programming: http://openmp.org/wp/

List of Acronyms

(A)MP IRQ CTRL	(Asymmetric) Multiprocessor Interrupt Controller
API	Application Programming Interface
COTS	Commercial (or Component) Off-The-Shelf
CCSDS	Consultative Committee for Space Data Systems
DHS	Data Handling System
DMS	Data Management System
ECSS	European Cooperation for Space Standardisation
FDIR	Fault Detection, Isolation and Recovery
FPU	Float Point UAnit
HKTM	Housekeeping Telemetry
IMA	Integrated Modular Avionics
IOMMU	Input / Output Memory Management Unit
IPI	Inter-Processor Interrupt
Kbps	Kilobit per second
Mbps	Megabit per second
MMU	Memory Management Unit
MPI	Message Passing Interface
NGMP	Next Generation MicroProcessor
POSIX	Portable Operating System Interface for Unix
RTOS	Real-Time Operating System
SOW	Statement of Work
SPARC	Translation Look-ahead Buffer
TLB	Translation Look-ahead Buffer
TSC	Time Stamp Counter
VMM	Virtual Machine Monitor
WCET	Worst Case Execution Time

1. SURVEY OF EXISTING MULTI CORE HARDWARE ARCHITECTURES

This section presents a broad survey of the multi core hardware architecture currently available on the commercial market. This survey was established in 2010, so more recent architecture may have come up in the recent years. We start by introducing the terminology of the multi core architectures, then give a high level description of each family of multi core, and presenting an example for each one.

1.1 Introduction

Outside the critical embedded systems domains, 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, due to memory bandwidth bottleneck, power dissipation issues, but also the ILP wall. Multi core processing is recognized as a key component for continued performance improvements.

A wide variety of hardware and software multi core architectures have been developed, each being optimised for a given class of application.

From the beginning of the microprocessors up to about 2005, the increase of the processing power of the processors was a consequence of the increase of the frequency allowed by the continuous miniaturisation of the design, as well as architectural optimisations increasing the number of instruction executed per cycle (superscalar architectures, addition of instructions working on vectors, …) But in the middle of the current decade, the gain improvement based on these techniques started to slow down, mainly because memory bandwidth did not increased at the same rate as the processing core frequencies, and because the thermal dissipation of the processing cores reached the technology limits. In order to get around these problems, the microprocessor architects started to introduce support for parallelism at software level.

At hardware level, two approaches can be adopted or mixed:

- The multi-instantiation of general purpose processing cores in the same chip. Sometimes, like in the case of the Intel hyper-threading technology, the duplication of the core is only partial. Most of the time, the number of duplicated cores remains rather low, typically from 2 to 8. The lowest level of caches can be duplicated while the higher level of cache and the memory are generally shared despite the very latest designs tend to replace the inter-core data bus and the centralised memory controller by a crossbar switch allowing simultaneous access to multiple memory controllers.
- The coupling of more specialised processing cores to the general purpose processing cores, these specialised cores ranging from classical DSP cores, up to very specialized processing

blocks in HW implementing basic algorithmic building blocks. The number of independent core can be rather high, sometimes several hundredth. In most cases, the memory architecture of heterogeneous multi core systems is distributed: despite a global memory still exists, each core has one or more local memory bank, and highly configurable “data mover” mechanisms.

1.2 Terminology

ALP Architecture-Level Parallelism. Processors that contain a heterogeneous mixture of core architectures exhibit ALP. As more cores are added to a single processor, it can be beneficial from a power and area standpoint to provide some heavy cores oriented towards single thread performance and other simpler cores oriented towards highly parallel workloads. Another example of ALP is integration of specialized cores, such as graphics processing units, and general-purpose cores on the same chip.

CLP Core-Level Parallelism. CLP occurs when a single processor core provides support for multiple hardware threads. Hardware threads differ from OS-managed software threads in that they have much lower overhead and can be switched between one other (usually) on a cycle-by-cycle basis. This allows another thread to execute when the current thread stalls (e.g., on a memory operation), thus making more efficient use of the processor's resources. CLP is a technique for increasing memory-level parallelism (MLP).

ILP Instruction-Level Parallelism. Superscalar processors are capable of executing more than one instruction each clock cycle. ILP is a form of implicit parallelism that is usually identified by the hardware automatically (e.g., via out-of-order execution) or by the compiler (e.g., scheduling instructions for multiple-issue, VLIW). The programmer does not usually need to explicitly deal with ILP.

MLP Memory-Level Parallelism. Modern memory subsystems have very high latency with respect to processor speeds, and the gap is increasing. MLP occurs when multiple simultaneous requests are made to the memory subsystem. This allows pipelining to occur, thus hiding the latency. Architectural capabilities such as out-of-order execution, hardware multi-threading, and STREAM processing are all aimed at increasing MLP.

SIMD Single Instruction Multiple Data. SIMD parallelism occurs when a single instruction operates on multiple pieces of data. Vector processors provide SIMD parallelism - a single instruction operates on entire vectors of data at a time. Many modern commodity processors support SIMD parallelism through multi-word wide registers and instruction set extensions. Many compilers attempt to automatically detect SIMD parallelism and exploit it, however programmers often have to explicitly structure their code to get the maximum benefit.

SMT Symmetric Multi-Threading. SMT is a form of CLP (core-level parallelism). In an SMT processor, instructions from multiple hardware threads can be issued in the same clock cycle to different execution units of a superscalar processor. This is in contrast to fine-grained multi-threading where only instructions from one thread may execute each clock cycle. SMT is useful on very wide superscalar

processors (e.g., the Power6) where it is unlikely that a single thread will use all of the available execution units.

TLP Task-Level Parallelism. Multi-core processors support running more than one context of execution (e.g., a process, a thread) at the same time, one per core. Programmers must explicitly structure their application to break the problem into multiple pieces in order to utilize TLP. This is in contrast to ILP, which requires no action on the programmers' part, scientific applications for distributed memory supercomputers have very high levels of TLP.

1.3 Architecture families

1.3.1 Homogeneous

In this multi core architecture family, several identical cores (same Instruction Set Architecture, same cache hierarchy, same operating frequency) are instantiated. All the cores have the same Unified Memory Access for the overall memory hierarchy, that is to say, none core is privileged comparing the other ones. The main communication scheme is the shared memory model. The QorIQ processor from Freescale is an example of such homogeneous multi core architecture.

1.3.1.1 Example: Freescale QorIQ (P4080)

Freescale delivers a groundbreaking three-tiered cache hierarchy on the QorIQ P4 platform. Each core has an integrated Level 1 (L1) cache as well as a dedicated Level 2 (L2) backside cache that can significantly improve performance. Finally, a multi-megabyte Level 3 (L3) cache is also provided for those tasks for which a shared cache is desirable.

The CoreNet™ coherency fabric is a key design component of the QorIQ P4 platform. It manages full coherency of the caches and provides scalable on-chip, point-to-point connectivity supporting concurrent traffic to and from multiple resources connected to the fabric, eliminating single-point bottlenecks for non-competing resources. This eliminates bus contention and latency issues associated with scaling shared bus/shared memory architectures that are common in other multi core approaches.

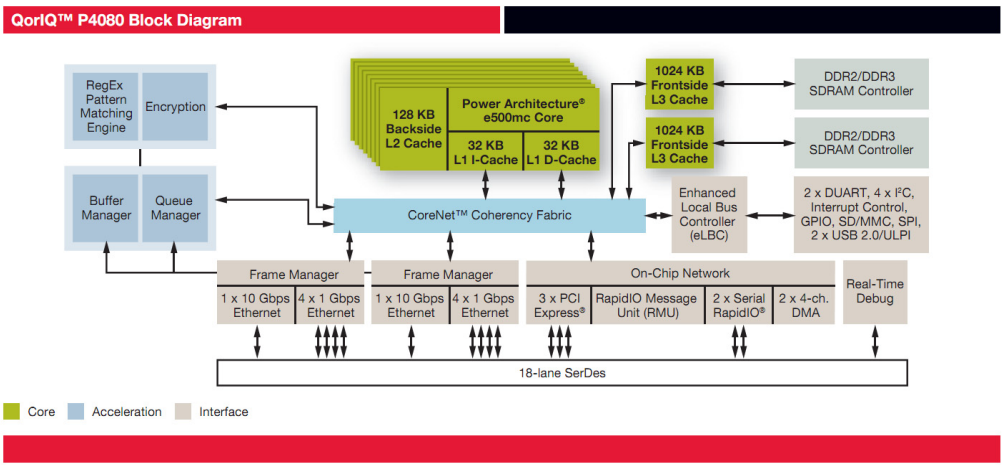


Figure 1 Freescale QorIQ Communication Processor (P4080) Block Diagram

This platform is deployed with advanced virtualization technology bringing a new level of hardware partitioning through an embedded hypervisor that allows system developers to ensure software running on any CPU only accesses the resources (memory, peripherals, etc.) that it is explicitly authorized to access.

1.3.2 Heterogeneous

This family of multi core architecture contains different alternatives:

1. the mixing of different core architecture (i.e GPP with DSP)
2. Each core does not see the system in the same way (memory hierarchy viewpoint).
3. some HW dedicated operators can be tightly coupled with one core specializing it

The main communication scheme remains the shared memory one, even if all the cores do not access the memory in the same way. In this architecture, often some cores can only see a sub-part of the memory hierarchy - their own tightly coupled memories – the other part can be accessed through dedicated DMA engine (programmed by the core itself or sometimes only by the System Controller)

Often, in this kind of architecture, the General Purpose Processor (homogeneous multi core or not) can be seen as System Controller who has access to all the resources of the system.

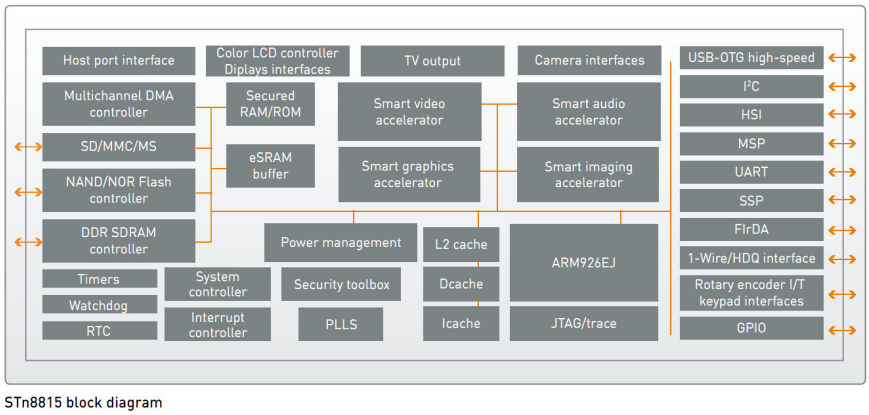
The Nomadik processor from STEricsson is an example of heterogeneous multi core architecture.

1.3.2.1 Example: STEricsson Nomadik

The Nomadik platform has been the STMicroelectronics (STEricsson) answer to the OMAP family fro Texas Instrument. It is an Application Processor to be used with a dedicated wireless modem.

It is a heterogeneous multi core architecture based on General Purpose ARMv5 superscalar core with 4 dedicated VLIW DSPs (MMDSP+). The Smart Video accelerator is HW-helped with optimized HW

pipeline (DCT/IDCT, Motion Estimation, …). The Smart Imaging is also HW-helped with dedicated sensor interface with pre/post processing features (resizing, rotating and so on).



STn8815 block diagram

Figure 2 Functional block diagram of STEricsson Nomadik (2003)

Public datasheets do not exhibit the internal architecture of the SoC. Nevertheless, figure 2 proposes a sketch highlighting the principles used for the internal interconnect.

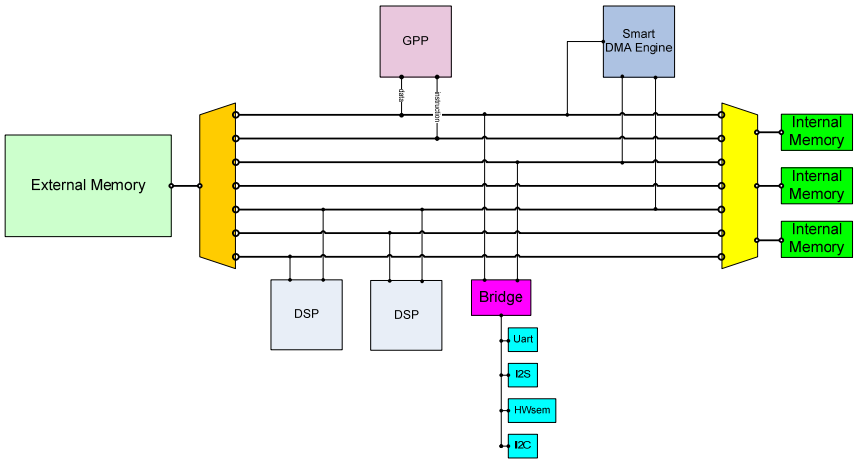


Figure 3 Interconnect Principles of STEricsson Nomadik

From architectural point of view, 3 points are interesting to highlight:

1. the multi-layered AHB interconnect , allowing each master to access any slave without any concurrency with other masters accessing other slaves.
2. the presence of embedded memory banks with a dedicated multi-port controller allowing to dedicate one bank per processing master, and the last one for shared memory exchanges. This embedded memory provide low latency local storage for any transient data.
3. the integration of an HW Semaphore IP providing synchronisation services for inter-cores (processing masters (DMA is a master, but not a processing master)). This IP manages 32 HW Semaphore with internal HW state machine ensuring atomic Set and Get and owner (mastered)

identification. This IP contains also an interrupt generator to each master core allowing to warn the core waiting for the semaphore.

This product does not provide any HW-assisted Virtualization feature, only a Trusted Zone, with dedicated HW state machine managing and supervising the entry and exit into a Secure Mode with some dedicated HW IPs only accessible under Secure Mode.

It can be highlighted that for the STn8220, successor of STn815, the AHB interconnect (AMBA-2) has been replaced by AMBA-3 and STBus interconnect in order to decrease significantly the latency introduced by AMBA-2 arbitrage.

1.3.3 Massively Parallel

This many-core family gathers the massively parallel architectures, where each core is seen as a node with its own resources and communicating with its neighbours.

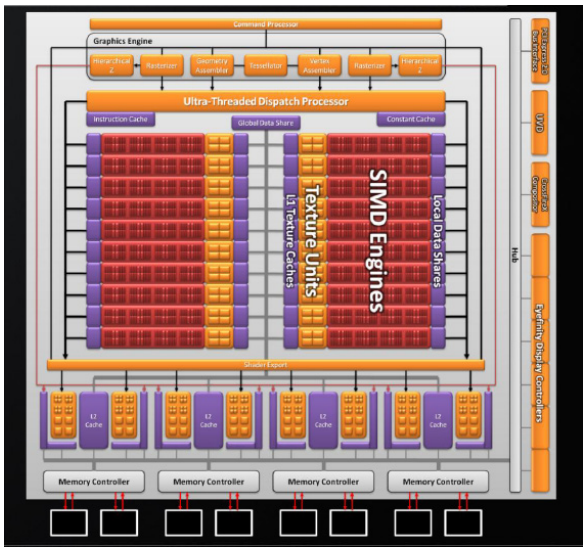
The main points that differentiate the many-core architecture with multi core one are:

1. The topology of the interconnection of the cores: 1d for multi core, multidimensional for many-cores;
2. The number of cores by itself: several for multi core, up to hundreds or thousands for many-cores.

The main communication scheme is more stream or message passing. Desktop computer graphic cards typically embed massively parallel processors.

1.3.3.1 Example: AMD HD5870 & NVIDIA Fermi

This architecture is an example of streaming one (massively parallel) really efficient for custom intensive data processing task but fully inefficient for any control tasks, and so are out of the scope of this document.



AMD/ATI High End Graphic Processor (HD5870) Block Diagram

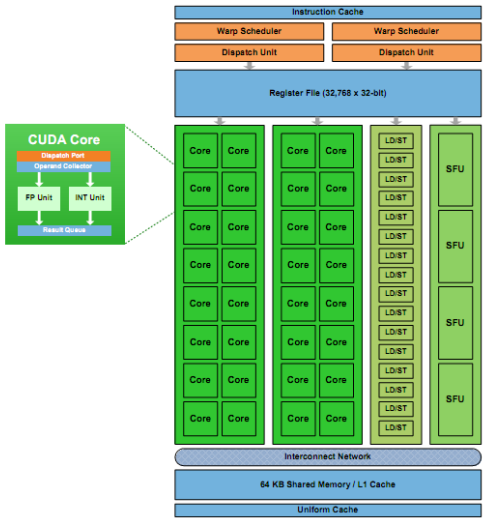


Figure 6. Each Fermi SM includes 32 cores, 16 load/store units, four special-function units, a 32K-word register file, 64K of configurable RAM, and thread control logic. Each core has both floating-point and integer execution units. (Source: NVIDIA)

NVIDIA Fermi Graphic Processor

Figure 4 Examples of massively parallel architecture

1.4 Lessons learnt from commercial multi core architecture

As shown in the previous sections, regardless of the type of the multi core architecture, one of the most important topics when designing a multi core processor is the memory architecture. Indeed, stacking several processing cores onto a chip is relatively easy, ensuring that each of these core is provided with an efficient access to the memory is much more difficult. The commercial industry has tackle this issue by developing advanced cache memories hierarchy (such as multi level caches, tightly coupled memories) and smart interconnect facilities between the cores and the main memory (such as multi layer buses, data coherency management units). These technologies, especially the interconnect facilities, are often proprietary and little information is available; this is understandable as they are key enablers of fast multi core processors. As shown on , a simple drawing clearly illustrates this issue.

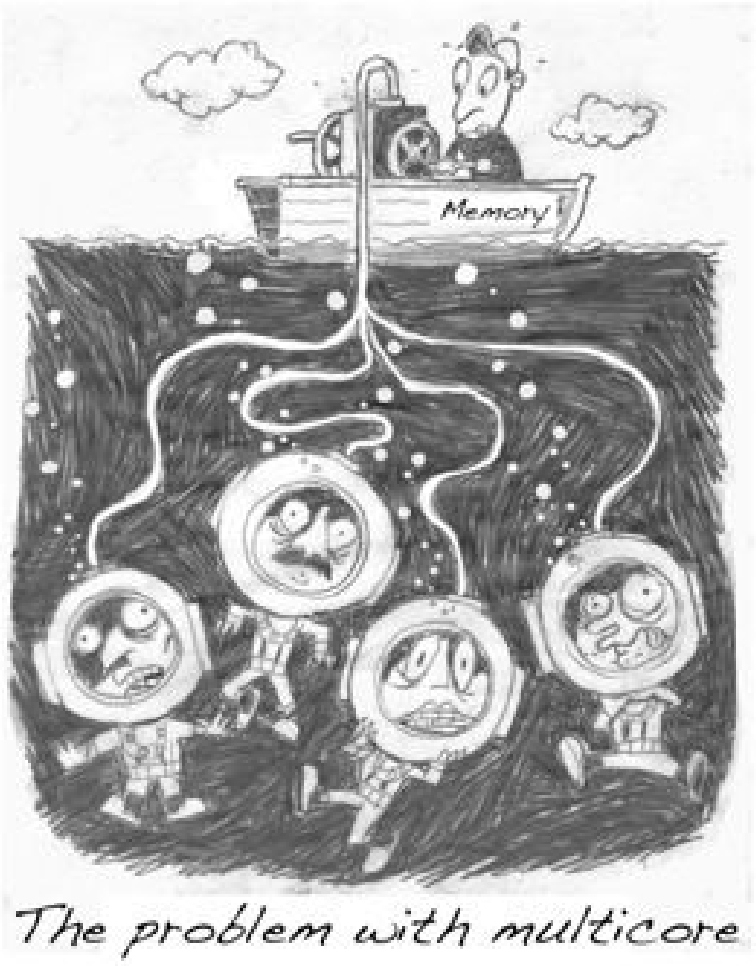


Figure 5 the problem with multi core

2. NGMP ASSESSMENT

The objective of this section is to provide feedback on the NGMP architecture based on the lessons learnt from the survey of existing commercial multi core architecture. First the issue of hardware coupling and indeterminism is discussed and then several improvements are proposed.

2.1 The Issue of Hardware Coupling

Multi core architectures distinguish themselves from true multi processors architectures in that only some parts of the processor are duplicated and that there is always a level a commonality between the cores. At some point in the architecture the cores are competing for access to a single hardware resource, thus creating a hardware coupling between the cores. Usually, the hardware ensures that the common resources are shared equally among the cores; the main issue is the time a core must wait before it gains access to a shared resource. For the NGMP design, the integer units, L1 caches, MMU and its associated TLB are quadrupled, the other resources of the system are shared among the cores of the system. For commercial multi core processors, the hardware coupling is usually at a higher level, at the L3 cache level for some architectures or at the memory controller level for some others.

On traditional mono core, mono threaded processors, the processing resources are time shared among the different tasks of an application. This time sharing can be completely handled by the operating system and has been the object of numerous researches and improvements, and, neglecting the effect of cache misses which are evened out, can be deterministic. For multi core processors, the way the processing resources are shared among the tasks of an application is different: tasks are allocated to different hardware processing resources at the same time and compete for access to shared resources. The operating system has no or little means to control the allocation of the shared resources. Thus when switching from mono core to multi core systems, the system integrator must accept to move from a finely controlled and almost deterministic way of sharing processing resources, to a level of indeterminism.

The issue of hardware coupling is even more complex for partitioned systems designers, as for those systems, the ultimate goal is to allow concurrent development and validation of software tasks for later integration on the same hardware platform. For this development approach to work, the system integrator must allocate guaranteed resources to each partition at the beginning the development. If there is too much hardware coupling between partitions executing on different cores, at integration time one partition may fail because it has not access to the foreseen resources. Too much hardware coupling thus prevents parallel development of independent partitions, reducing the interest for partitioned systems.

Though the hardware coupling is an intrinsic attribute of all multi core architectures, it can be minimised down to a level where its impact on the global system is negligible. This is accomplished by making the duplicated parts of the processors as independent as possible from the common parts. The key driving figure is the frequency of concurrent accesses to shared resources. In the following sections, we propose

several improvements of the NGMP architecture which in our opinion would allow reducing the frequency of the concurrent accesses. Our propositions can be sorted in two classes: the ones that suggest improving the capabilities of the duplicated cores in order to reduce the needs for them to access shared resources, and the others that propose to duplicate more parts of the systems. The first ones may be easier to implement than the second ones which require important architectural changes.

2.2 Core improvements

2.2.1 L1 caches size and associativity

When seeking to reduce the number of bus accesses, one of the first improvements that come to mind is to increase the size of the L1 caches in order to increase the probability of cache hits. This approach is well known and heavily used: commercial multi core processors have larger and larger caches dedicated to a single core and even multi level caches (L1 and L2 are per core, L3 is unified).

The important fact here is that it is more beneficial to the system to increase the size of L1 caches rather than the L2 cache because accesses to the L2 cache are made concurrently by all the cores. Increasing the size of the L2 cache may bring additional overall performance to the system but will not help in reducing the indeterminism.

When several partitions are executed on a single core one after the other, the spatial and temporal localities of data and instruction accesses are reduced as partitions are independent from each other. Thus in a partitioned system, for the same cost, increasing the associativity of a cache may bring more benefits than increasing the cache size.

2.2.2 Explicit cache use

As stated before, for partitioned systems, the temporal and spatial locality of the accesses may be low, thus defeating traditional hardware cache management mechanisms. However, since partitions are scheduled for execution in a cyclical fashion, the hypervisor has knowledge of which partition is executed when, and it could use this knowledge to efficiently preload partitions' data and code before their execution. It would result in less cache misses and therefore less concurrent access to the main bus.

In order to implement such behaviour, the hypervisor must be able to explicitly control the contents of the L1 caches; this could be implemented with cache way locking or by configuring the caches as local scratch pad memory. In order to be efficient, this requires that the hardware allows triggering a non blocking transfer of blocks of data between memory and L1 caches: some sort of pre-fetch operation.

Another simpler solution this behaviour is to add dedicated scratch pad memories for both instruction and data alongside the caches. The hypervisor would be in charge of preloading relevant data and instructions into the scratch pad memories before the context switch between two partitions.

2.3 Data cache write behaviour

At the time of writing of this document, the L1 data cache of the NGMP design implements a write through policy with some write buffers. Write buffers are used to allow the core to continue its execution while the cache waits for the bus availability to write the data. The use of write buffers thus allows reducing the indeterminism when writing data, however this mechanism is easily defeated by burst writes or sequences of reads and writes as a read miss causes the write of the buffers' contents to memory.

Implementing a true write back policy for the L1 cache would allow greatly reducing the number of bus accesses. Moreover, in a partitioned system, since independent partitions are running on different cores, there should be few overlapping memory areas between the cores, therefore cache coherency could be handled in software by the hypervisor. This is especially true if shared memory is only used for Inter Partition Communications, since all IPCs are by design managed by the hypervisor.

2.3.1 TLB size

Each TLB miss triggers at maximum three consecutive accesses to the memory for the hardware table walk during which time the core execution is halted. Thus the cost of each TLB miss is very high.

If several partitions are scheduled for execution on the same core, the TLB can become very quickly saturated, and therefore a lot of time can be lost when switching from one partition to another.

Due to the high cost of the TLB misses, increasing the size of the TLB is an operation with a high return over cost ratio.

2.3.2 Heterogeneous caches

We have seen in the previous sections that increasing the capabilities of L1 caches and TLB can helps to reduce the indeterminism. However the effect of each of the actions described above is very dependant on the use of the cores. For example increasing the TLB or the associativity of the caches will only be beneficial if several partitions are scheduled for execution on the same core. Upgrading the caches and /or TLB can necessitate a lot of hardware resources and may not be feasible. In order to save resources only some cores could be upgraded: there would be some "high end" cores with larger caches and TLB and some "low end" cores with regular caches. Since the partitions are allocated to processing cores statically by the system integrator, the scheduling plan can be built to make the best use possible of the high end cores while avoiding saturating low end cores.

2.4 Architectural changes

2.4.1 Multi port L2 cache and layered bus

In the current NGMP design only the integer units and their associated L1 caches and TLBs are duplicated, the bus used to access the L2 cache is shared among all the cores. Due to the use of small

L1 caches and write through policy, each core will generate a lot of traffic on the bus during its operation. The bus arbiter uses a round robin policy to grant the masters access to the bus; there are 5 different masters on this bus. When one core needs to make an access to the bus, it may get access to the bus immediately or have to wait for the four preceding masters to complete their accesses before it can get access to the bus. Moreover, if one of the preceding cores' accesses triggers a L2 cache miss, this core will hold the bus during all the time needed to load data from memory. Therefore the indeterminism induced by the sharing of the bus is very high.

With the current architecture the full potential of the L2 cache is not exploited: one core can trigger a cache miss processing and hold for bus for as long as necessary to load data from memory, while the other cores are waiting to perform accesses that could result in cache hits and could be served very quickly. The gain of a cache hit is therefore nullified by the fact that another core holds the bus.

For these reasons, a huge reduction of the indeterminism could be obtained by implementing a multi port L2 cache and a layered bus. Each core could perform concurrent accesses to the L2 cache, it would always be granted control of its private bus. For requests that need access to the main memory, the L2 cache could implement advanced sharing techniques in order to minimize latency while ensuring each core has its fair share of memory access. The point being that, the implementation of the sharing policy is no longer constrained by the bus arbiter implementation.

The L2 cache of the NGMP design already implements advanced techniques such as locked ways and master index replacement policy that can be used to segment the cache among the cores. These techniques can be used to transform the L2 cache into a scratch pad memory for some of the cores, which as stated before can yield important performance improvements for a partitioned system. However for this solution to be efficient, accesses to the scratch pad memory need to be deterministic. So in order to be able to fully exploit the advanced segmentation capabilities of the L2 cache, a multi port implementation is needed as well.

2.4.2 Separate I/O memory

Going down one more level, all the cores and I/O devices (through the IOMMU) are competing for access to a single memory controller. The round robin policy of the bus arbiter gives the equal priority to CPU or IO accesses; this may not be the best approach as a CPU can always wait, while stalling an IO memory access may result in data loss. In any case, the competition to access a single memory controller between cores and IO will result either in additional indeterminism of core execution or IO data loss.

The use of a separate memory and memory controller for I/O devices allows reducing this indeterminism. This solution is already deemed necessary for much simpler mono core devices like the SCOC3. For a system like the NGMP, for which the number of masters is much more important, it makes even more sense. I/O devices can execute their DMA transfers with a low latency without stealing memory cycles from the processing cores. However, except for pure routing operations, I/O data is bound to be used by

a processing core, and then data must be moved from the I/O memory to the main memory in parallel with the operation of the processing cores.

The main difference then between using an I/O memory or not in terms of determinism is that without an I/O memory, DMA transfers steal memory cycles in a non deterministic and asynchronous way from processing core operations, whereas when using an I/O memory, the transfers between the I/O memory and the main memory are initiated by the software and therefore controlled in a deterministic way.

2.4.3 Shared L2 TLB

TLB misses are important sources of indeterminism and latencies as a TLB miss can trigger as much as three consecutive memory accesses. In partitioned systems as well as for SMP systems, the page table tree is shared among all the cores: the same top level context table is used by all the cores, and the lower levels can be shared as well. Therefore it seems interesting to use a unified L2 TLB in the NGMP design. Since the costs of a data cache or an instruction cache miss are lower than the cost of a TLB miss, improving the TLBs should yield important performance improvements.

In a multi core systems with an unified L2 TLB, the cores' MMU would first interrogate their local TLB and then interrogate the shared L2 TLB, in case no matching entry is found in both TLBs, the hardware table walk is performed as before. This scheme works for SMP and partitioned systems but cannot be used for ASMP systems in which the same context number / page descriptor couple can be used differently on two independent cores.

2.4.4 L2 cache associativity

In the current NGMP design, the associativities of L1 and L2 caches are the same, both caches are 4 ways associative. The L2 cache is shared among all the cores which are potentially executing completely independent threads operating on separate blocks of data and instructions. For a partitioned system, the temporal and spatial locality is even more reduced at L2 level.

Therefore it seems that increasing the associativity of the L2 cache would yield performance improvements. In any case, it seems logical that shared cache have higher level of associativity, this is the case for most commercial processors: the cortex A9 MP architecture on which most of the current dual core embedded processors for smartphones are based (Apple A5, Samsung Exynos 4210) has an L2 cache which is up to 16-ways associative, the latest dual core processor from Qualcomm (dubbed Krait) has an 8-ways associative L2 cache. The processor from Qualcomm is a good example of a progressive cache hierarchy: it has 3 caches levels, from L0 to L2, L0 is direct mapped, L1 is 4-ways associative, and L2 which is shared among all the cores is 8-ways associative.

2.4.5 Cache hierarchy

In the current NGMP design, the caches hierarchy is neither inclusive (a cache line can be evicted from L2 while it is present in L1) nor exclusive (a cache line can be present in both L1 and L2).

The only advantage of a strictly inclusive hierarchy would bring is that if snoop is wanted on the AHB memory bus: having all the contents of L1 caches inside L2 would greatly simplifies snooping mechanisms. However, it is our belief that L2 cache coherency can be managed at application level, therefore snoop on the AHB memory bus is not a required feature. Note however that only the L2 cache coherency can be handled at software level, dealing with L1 caches coherency at software level would be to difficult. Moreover, given the size ratio between L1 and L2 caches, an inclusive policy would result in an L2 cache being mostly occupied by the data present in L1 caches as well.

For the same reason (L1/L2 sizes ratio) an exclusive cache hierarchy could be an interesting feature and would allow maximising the use of L2 cache. This is the hierarchy used by AMD for quad core processors. However, implementing a strictly exclusive cache hierarchy is tricky: cache lines must be exchanged between caches, L2 needs to be aware of the contents of all L1 caches… A doable improvement that can be performed to increase the exclusivity of the caches is to upgrade the replacement policy of L2 to replace in priority cache lines that are also present in one of the L1 caches. Also, when a line needs to be evicted from a L1 cache, it is written in the L2 cache, using a single 128 bit bus cycle; the next time a process needs to access this line, it is read from L2 instead of the main memory.

3. SOFTWARE TECHNIQUES FOR MULTI CORE

In this section we present the software techniques that have been developed and successfully used by the general industry to harness the processing power of multi core processors and efficiently manage the execution parallelism. Software techniques specific for multi core processors have been developed at various levels:

- At the execution model / operating system level: there are different ways to abstract the multi core hardware, we will present the most important ones.
- At the task distribution and synchronisation level: multi core hardware enables concurrent execution of multiple tasks, in order to maximize the use of the hardware the application developer must use techniques to split its application into several tasks that can be executed concurrently.
- At I/O management level: several ways of managing the I/O of multi core hardware exists, especially when using a hypervisor.

3.1 Execution models

Various software techniques are used to run application software on multi-core systems. The examples mentioned here below are amongst the most common, but many other exist since the choice for a given application is always a compromise between multiple application dependant constraints.

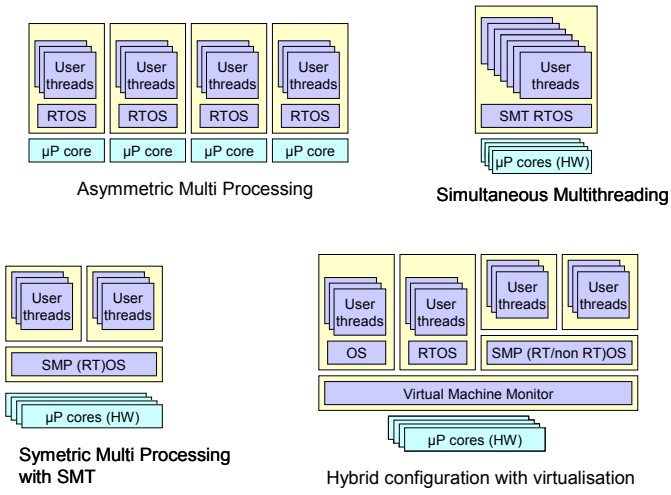


Figure 6 Example of Execution models

Asymmetric Multi Processing

In this model, a dedicated (RT)OS is deployed on each core, with the SW application running on it. So, the applications design is very similar to single-core ones.

Any required communication/synchronisation between the thread of different applications running on different core shall be done explicitly at application level. The application developer then keeps the full control on any inter-core communication.

The drawbacks of this model are that the central memory management is more complex and static. So, the isolation between the various application running on different (RT)OS/cores shall be handled by hand and very soon in the design flow of the overall system, reducing the ability to admit new SW during the design and after.

This model fits well a set of highly independent application with low interaction between them. For example, a given set of IOs shall be handled by only one application running on a given RTOS on a given core.

Symmetric Multi Processing

In this model, the (RT)OS is in charge of managing all execution cores.

The applications, running on top of it, see the execution platform like a single-core one.

Commonly, the core allocation is performed at process level. It exists some libraries and language extension (like OpenMP) allowing to express the parallelism at code level.

Most of the SMP OSes provide some facilities to create an affinity between a given process with a given core, in order to avoid process migration between the cores leading to an inefficient usage of caches and MMUs.

The high challenge to implement this kind of model is to manage the parallelism/concurrency internally into the kernel (see Linux big kernel lock story [LINUX_BKL1/2/3]) without introducing huge latencies.

This execution model is near to transparent for SW developers, but the price at determinism level could be high:

- Size and variability of the kernel latency,
- Low control on what is running at a given time, leading to “out of control” latency in shared resources accesses.

Virtual Machine Monitor/Hypervisor

The concept of partitioned software architectures was developed to address security and safety issues. The central design criterion behind this concept consists in isolating modules of the system in partitions. Temporal and spatial isolation are the key aspects in partitioned systems.

Two approaches to build a partitioned system can be found.

- **Separation kernel**: it is also known as operating system-level virtualisation. In this approach the operating system is extended (or improved) to enforce a stronger isolation between processes or groups of processes: memory is statically allocated to each partition; a cyclic plan is used to

schedule partitions; there is no visibility of the processes between partitions, etc. Each group of isolated processes is considered as a partition.

- **Platform virtualisation**: Platform virtualisation is performed on a given hardware platform by a layer of software called hypervisor or virtual machine monitor (VMM), which creates a simulated computer environment, a virtual machine. Each of the virtual computers is referred as “guest”, “domain” or “partition”1.

Although the idea of virtualizing is simple and clean, implement a fast and efficient virtualizer is far from trivial. There are several competing techniques depending on the underlying hardware, and on the degree of virtualisation.

Depending on WHAT is virtualised:

- **Full virtualisation**: The complete platform is virtualised. The partition has no direct access to the real hardware.
- **Partial virtualisation**: Some parts of the hardware can be directly controlled by a partition. This is known as dedicated devices. The virtualizer grants exclusive access to some hardware devices to a designated partition.

Attending to HOW the virtualisation is implemented:

- **Transparent**: The code on the virtualized platform is not aware of being executed on a virtualised environment. In this case, the same partition code can be executed both in the native or the virtualised platform.
- **Explicit**: Also referred as **para-virtualisation**. The hypervisor does not necessarily emulate the native hardware. Rather than an accurate recreation of the platform hardware behaviour, the same (or similar) services are provided with a special API. Those services are provided thought a set of hypercalls. The partition code (at least the operating system or the HAL2) shall be tailored to run in the virtual environment.

Hypervisor can also be categorised according to the execution environment where the hypervisor is executed:

- Type 1: executed directly on the native hardware, also named native or bare-metal hypervisors.
- Type 2: executed on top of an operating system as a regular process. The native operating system is called host operating system and the operating systems that are executed in the virtual environment are called guest operating systems.

Simultaneous Multithreading Processing

This model is based on:

- An unified address space shared among all the cores;
- A simple communication model through shared memory;
- Some communication-synchronisation primitives similar to SMP without access control/protection, but with the same challenge to develop efficient ones (big kernel lock issue).

Hybrid Configuration Processing Model

Any mix of the previous models can be imagined.

For example:

- Deploying a multi-threading or multi-processing model on some cores, and other one using an AMP or another SMT or SMP model, but without any underlay hypervisor. This configuration (not presented into 0) exhibits major drawbacks since it does not provide any control, neither on memory accesses between the different OSes, nor on the initialisation phase, including different cores booting, boot image deployment on each core and common resources (like memory Controller, L2 cache programming and peripheral control/sharing).
- The configuration can be deployed but with an underlay hypervisor providing mechanisms to manage memory access rights, and other issues listed above.

3.2 Task distribution and synchronisation

Regardless of the executive model actually used, when working on a multi core platform the system designer has the possibility to execute concurrently several tasks. The distribution of tasks onto processing core can either be explicit and under the control of the system designer or implicit and handled by the operating system.

For applications made of several relatively independents tasks, the system designer has to allocate each task to the most suited processing core, he can rely on the operating system to do that, or do the allocation himself. The main difficulty resides in the synchronization of these tasks. We present the most well known synchronization techniques in the following sections.

When an application is made of one single task, like simple data processing application, the system designer is forced to modify its application to make the most efficient use of the multi core platform. Two main techniques exist to transform a mono task application into a multi task application in order to run it on a multi core processor: data parallelism and task parallelism.

3.2.1 Data parallelism

When using the data parallelism technique, the mono task application is ported onto a multi core platform by duplicating the execution of the application task on all the execution unit of the platform. Each execution unit is executing the same program. The input data set to be processed by the application is split among the execution units and each unit is in charge of processing its subset of the data.

This approach is rather painless to implement as it allows reusing most of the original application. However since each core is executing the same program, each core will compete for access to the same hardware resource. On multi core architecture showing a strong hardware coupling between the cores like the NGMP, this could result in poor overall performance. Moreover, not all processing application allows splitting the input data set into subsets and independent processing of each subset. For instance some image processing algorithms require processing the complete image at once and not a sub part of it.

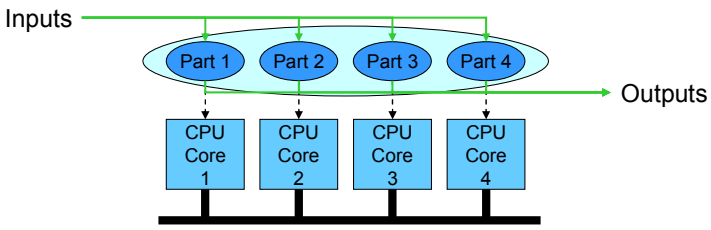


Figure 7 Data parallelism

3.2.2 Task parallelism

When using the task parallelism technique the mono task application is split into several sub tasks. Each sub tasks is executed on a different execution unit. Input data is streamed from one execution unit to another. This type of implementation is quite similar to a software pipeline.

This approach may be more difficult to implement as the system designer has to redesign its application. However the designer has a finer control on what each core is executing and therefore can try to minimize the competition between the cores when accessing shared hardware resource. As data is moving from one core to another, an efficient data sharing mechanisms must be available in order for this approach to be efficient. Shared high level caches (L2 or L3) are useful to implement the data movement between the cores.

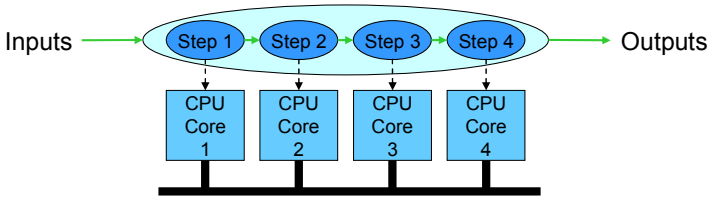


Figure 8 Task parallelism

3.2.3 Synchronisation on Time

Within this model, on a shared common timer event, each core executes its workload (all its inputs have been produced at previous time slot) producing its output before the end of the time slot. So, in this model, all tasks use predefined buffers (input, output, internal). The validity and the coherency of the data are insured by the time synchronization.

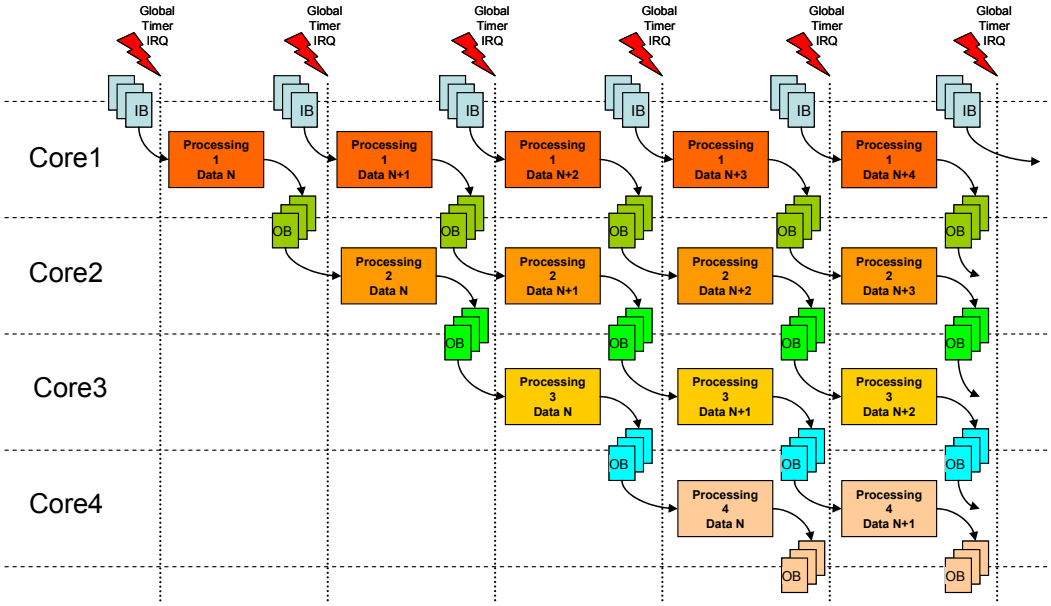


Figure 9 Synchronisation on Time

Advantages:

- High execution determinism since we know what is executed in parallel on each core
- Efficient use of the available computation power, by minimizing all non-processing tasks
- Fit well for intensive data processing

Drawbacks:

- Cutting up the overall processing into sub-task with the same duration is a complex activity
- Does not fit with data handling activities

Execution Platform:

This synchronisation model can be implemented with:

- Bare-metal machine
- with light-weight hypervisor
- SMP OSes by playing with CPU affinity, but with a low level of efficiency due to the non-usage of most of the OS services.

For deploying this model in an efficient way, the HW platform shall provide a global timer with an interrupt broadcasting facility.

3.2.4 Synchronisation on Data Flow

Within this model, the synchronization is performed on the input data or output buffers availability. So, in this model, buffers (input or output) are exchanged between the various tasks (possibly with zero-copy mechanisms).

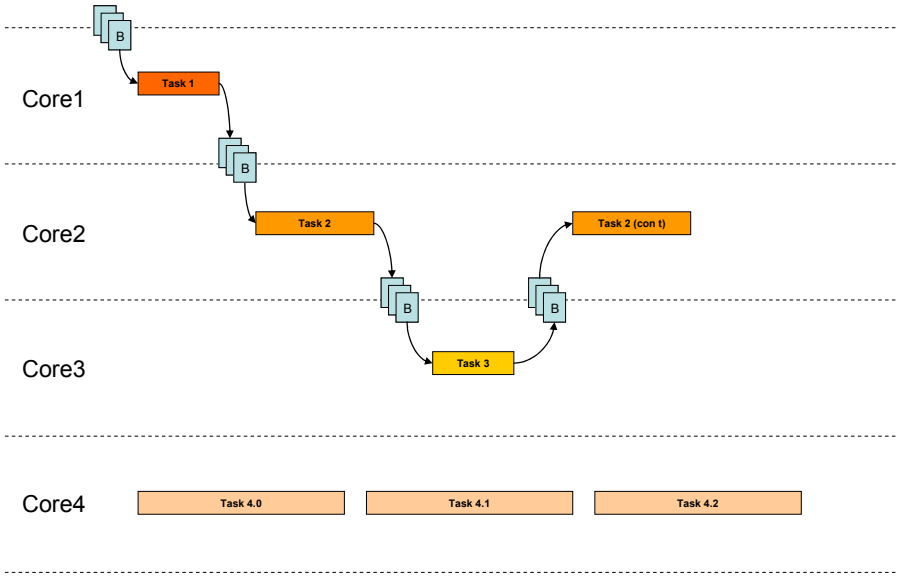


Figure 10 Synchronisation on Data Flow

Advantages:

- Closest to the optimal use of computing power
- Ease (comparing to synchronization on time) the processing parallelisation by using criteria on the data locality and data exchange minimization criteria.
- High configurability to add a new processing stage.

Drawbacks:

- Introduce some execution indeterminism due to the HW coupling effects
- Fit well with best effort QoS

Execution Platform:

This synchronisation model can be implemented with:

- Hypervisor
- AMP with dedicated inter-cores/OS synchronisation primitives
- SMP OSes

For deploying this model in an efficient way, the HW platform should provide some hardware-assisted mechanisms, like global HW semaphores, mailboxes and at least a way for each core to send an asynchronous event to each other core (that is to say for a quad-core, 12 interrupt lines)

3.2.5 Master/Slave Model

Within this model, one master distributes the workload on the different processing resources by providing all required data and output buffers. So, in this model, all buffers are under the control of the master.

Advantages:

- Synchronization/communication paradigm near to Remote Procedure Call

Drawbacks:

- High volume of exchanged data

Execution Platform:

- This synchronisation model can be implemented with:
- Hypervisor
- SMP OSes

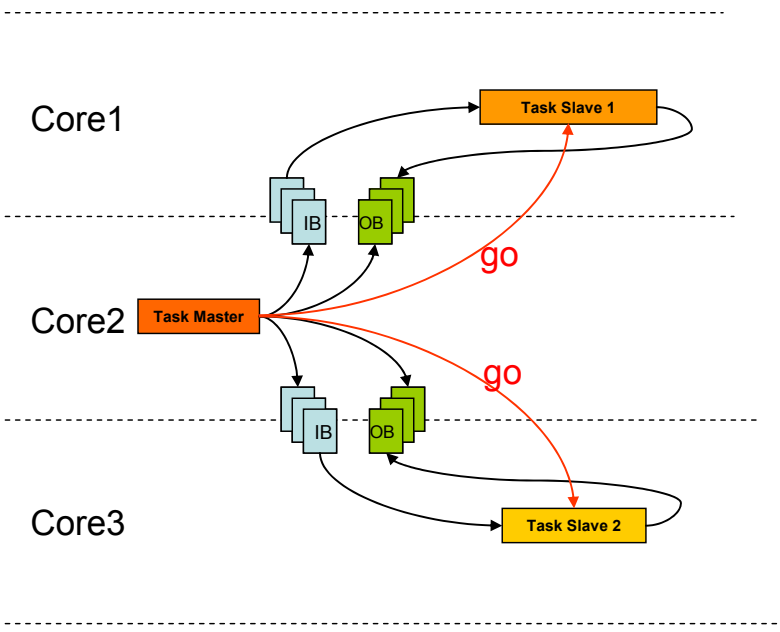


Figure 11 Master/Slave Model

3.3 I/O Management

Depending on the deployed execution model, the IO Management will be adapted:

- Asymmetric multi processing: in this model, each IO shall be dedicated to a given core with its associated OS. The only way to share some peripheral is to drive/manage it through a given core, and use inter AMP communication scheme to share it with other core.
- Symmetric multi processing: In this model, the SMP kernel will handle directly the IO in a transparent way for SW developer (not the driver developer, who that shall deal with his driver re-entrancy).
- With a hypervisor: This IO Management alternatives are presented in the next section.

3.3.1 I/O Management with a VMM/VM

In a VMM/VM system, applications executed in several partitions may require the access to a common I/O resource, e.g. communication interface (Ethernet, SpaceWire, etc.). The VMM/VM must support the virtualization of I/O requests from guest applications. I/O virtualization may be supported through different models. The most common models envisaged are:

Emulation: A VMM/VM may expose a virtual device to guest software by emulating an existing I/O device. The VMM/VM emulates the functionality of the I/O device in software over whatever physical devices are available on the physical platform. I/O virtualization through emulation provides good

compatibility (by allowing existing device drivers to run within a guest), but pose limitations with performance and functionality.

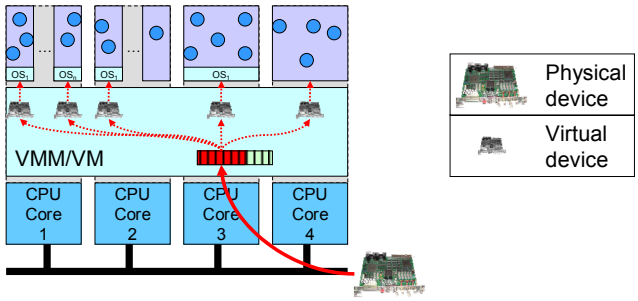


Figure 12 I/O device virtualization/emulation

System partition: The VMM/VM executes a special system partition. This partition is in charge to manage all the input and output operations. From one side, the system partition contains the drivers of all the devices connected to the system. It is able to control all the devices and receives all the interruptions. From the other side, the system partition is connected to all other partitions (requiring an access to any device) of the system through a dedicated communication bus or shared memory area in order to exchange the data and signals.

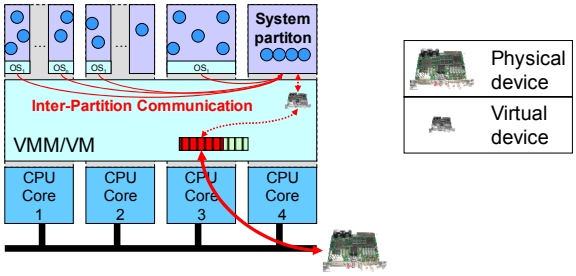


Figure 13 I/O through System Partition

Assignment: A VMM/VM may directly assign the physical I/O devices to VMs. In this model, the driver for an assigned I/O device runs in the VM to which it is assigned and is allowed to interact directly with the device hardware with minimal or no VMM/VM involvement. Robust I/O assignment requires additional hardware support (e.g. Input/Output Memory Management Unit or IOMMU) to ensure the assigned device accesses are isolated and restricted to resources owned by the assigned partition. The I/O-container-based approach removes the need for running the physical device drivers as part of VMM privileged software.

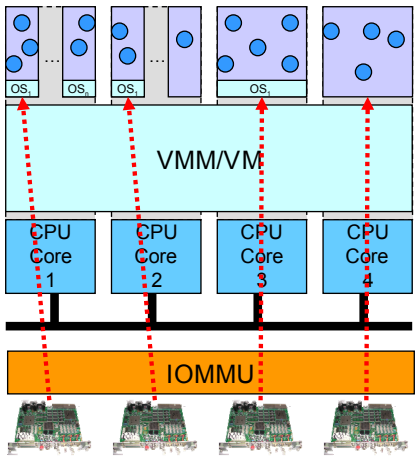


Figure 14 Direct I/O through IOMMU

The implementation of the first two models may have an impact on the performance of the system. To emulate the I/O devices, the VMM/VM will require more computing resource. A system partition will consume resources as well. The assignment model supports an optimized access to I/O devices requires dedicated hardware, the IOMMU that is foreseen in the NGMP architecture.

3.4 Software tools

A lot of software tools, APIs, libraries, language extensions have been developed to help the system designer with efficiently harnessing the processing power of multi core processor. We present in the following sections the two most important APIs used for parallel programming.

3.4.1 OpenMP

OpenMP (Open Multi-Processing) [OPENMP] is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran on many architectures, including Unix and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behaviour.

OpenMP is an implementation of multithreading, a method of parallelization whereby the master "thread" (a series of instructions executed consecutively) "forks" a specified number of slave "threads" and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

The section of code that is meant to run in parallel is marked accordingly, with a pre-processor directive that will cause the threads to form before the section is executed. Each thread has an "id" attached to it which can be obtained using a function (called omp_get_thread_num()). The thread id is an integer, and the master thread has an id of "0". After the execution of the parallelized code, the threads "join" back into the master thread, which continues onward to the end of the program.

By default, each thread executes the parallelized section of code independently. The runtime environment allocates threads to processors depending on usage, machine load and other factors. The number of threads can be assigned by the runtime environment based on environment variables or in code using functions. The OpenMP functions are included in a header file labelled "omp.h" in C/C++.

```
int main(int argc, char *argv[])
 {
   #pragma omp parallel
   printf("Hello, world.\n");
   return 0;
 }
```

Table 1 OpenMP example

The OpenMP API is best suited to implement parallelization technique such as the data parallelism technique described in section 3.2.1 and the master/slave model synchronization technique described in 3.2.5.

3.4.2 MPI

Message Passing Interface (MPI) [MPI] is an API specification that allows processes to communicate with one another by sending and receiving messages. It is typically used for parallel programs running on computer clusters and supercomputers, where the cost of accessing non-local memory is high. MPI was created by William Gropp, Ewing Lusk and others.

MPI is a language-independent communications protocol used to program parallel computers. Both point-to-point and collective communication are supported. MPI is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation. MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today.

MPI is not sanctioned by any major standards body; nevertheless, it has become a de facto standard for communication among processes that model a parallel program running on a distributed memory system. Actual distributed memory supercomputers such as computer clusters often run such programs. The principal MPI–1 model has no shared memory concept, and MPI–2 has only a limited distributed shared memory concept. Nonetheless, MPI programs are regularly run on shared memory computers. Designing programs around the MPI model (contrary to explicit shared memory models) has advantages over NUMA architectures since MPI encourages memory locality.

Although MPI belongs in layers 5 and higher of the OSI Reference Model, implementations may cover most layers, with sockets and TCP used in the transport layer.

Most MPI implementations consist of a specific set of routines (i.e., an API) directly callable from Fortran, C and C++ and from any language capable of interfacing with such libraries (such as C#, Java or Python). The advantages of MPI over older message passing libraries are portability because MPI has been implemented for almost every distributed memory architecture and speed because each implementation is in principle optimized for the hardware upon which it runs.

MPI uses Language Independent Specifications (LIS) for calls and language bindings. The first MPI standard specified ANSI C and Fortran-77 bindings together with the LIS. At present, the standard has several popular versions: version 1.3 (shortly called MPI–1), which emphasizes message passing and has a static runtime environment, and MPI–2.2 (MPI–2), which includes new features such as parallel I/O, dynamic process management and remote memory operations. MPI–2's LIS specifies over 500 functions and provides language bindings for ANSI C, ANSI Fortran (Fortran90), and ANSI C++. Object interoperability was also added to allow for easier mixed-language message passing programming. A side–effect of MPI–2 standardization (completed in 1996) was clarification of the MPI–1 standard, creating the MPI–1.2.

MPI–2 is mostly a superset of MPI–1, although some functions have been deprecated. MPI–1.3 programs still work under MPI implementations compliant with the MPI–2 standard.

MPI is often compared with PVM, which is a popular distributed environment and message passing system developed in 1989, and which was one of the systems that motivated the need for standard parallel message passing. Threaded shared memory programming models (such as Pthreads and OpenMP) and message passing programming (MPI/PVM) can be considered as complementary programming approaches, and can occasionally be seen together in applications, e.g. in servers with multiple large shared-memory nodes.

```
/* C Example */
#include <stdio.h>
#include <mpi.h>
int main (argc, argv)
     int argc;
     char *argv[];
{
  int rank, size;
  MPI_Init (&argc, &argv);  /* starts MPI */
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);  /* get current process id */
  MPI_Comm_size  (MPI_COMM_WORLD,  &size);   /* get number of processes
```

```
*/
  printf( "Hello world from process %d of %d\n", rank, size );
  MPI_Finalize();
  return 0;
```

Table 2 code extract using MPI calls

4. GUIDELINES FOR MULTI CORE USE IN SPACE APPLICATIONS

This section aims at providing a set of recommendations for use of multi core processors in space applications. We will start by describing the state of art multi core execution platform available to the space community: the Xtratum lightweight hypervisor ported on top of the NGMP. Then we will describe the two type of space applications that we think can benefit from the use of multi core processor, and provide a set of implementation recommendations for each of these applications.

4.1 Xtratum on NGMP execution platform

Xtratum is a lightweight, bare metal hypervisor which provides time and space partitioned execution environments to guest partitions. A partition is like a virtual computer, it is allocated, in a guaranteed fashion, a certain amount of memory and processing time. Each partition can run its own operating system or no operating system at all. Xtratum was initially developed for mono core architecture and is currently under evaluation in the ESA led study called Integrated Modular Avionics for SPace (IMA-SP). In the frame of this study Xtratum has been ported on top of the NGMP hardware, several features specific to multi core have been added to the standard version.

Each partition is allocated a number of Virtual CPUs, each of these virtual CPU executes a different thread inside the partition's memory context. The system integrator maps each partition's VCPU to hardware CPU. Each VCPU of a partition is scheduled for execution on a hardware CPU according to scheduling plan defined by the system integrator.

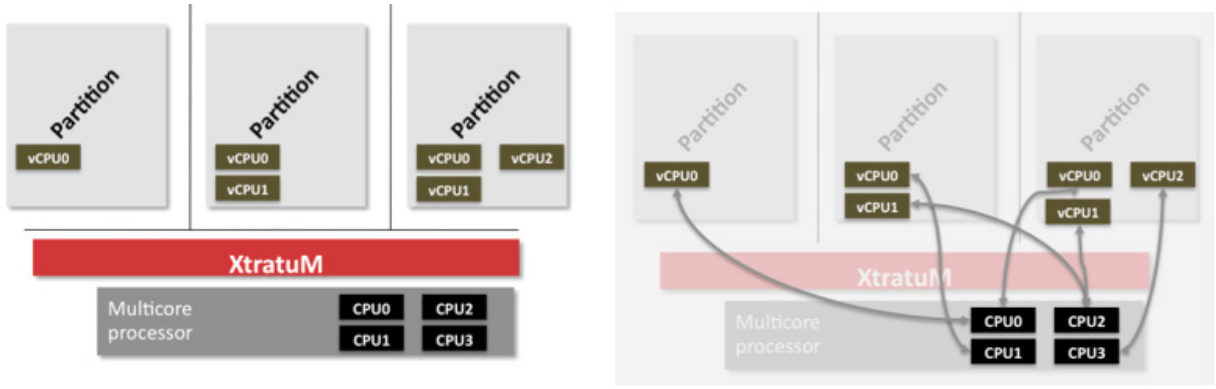


Figure 15 VCPU allocation to hardware CPU

This functionality enables the system integrator to allocate more than one CPU to a partition at a given time. The execution parallelism has to be managed inside the partition.

The system integrator defines a scheduling plan for each of the hardware CPU of the platform. Virtual CPUs of the partitions are executed on hardware CPU according to their respective scheduling plans. Two different kinds of scheduling policies are available: cyclic scheduling and fixed priority scheduling. Cyclic scheduling policy is the same scheduling policy that was used for mono core processors. Fixed

priority scheduling policy is a new feature, under this scheduling policy each virtual CPU has an associated priority number, among the virtual CPUs that are ready for execution, the one with the highest priority is executed. This scheduling policy is pre-emptive, meaning that if a virtual CPU with a higher priority than the one currently being executed becomes ready, the hypervisor will perform a context switch to that virtual CPU. Asynchronous events like external interrupts can be used to switch a virtual CPU into the ready state.

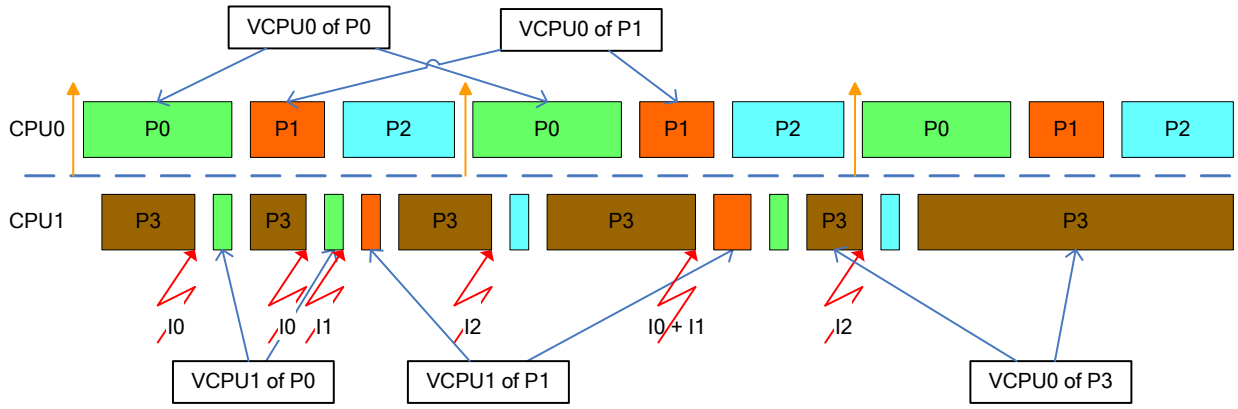


Figure 16 Scheduling plan example

Figure 16 illustrates an example of a complex scheduling plan on the NGMP. CPU0 is using a cyclic scheduling policy, three virtual CPUs from three partitions are scheduled for execution on this CPU. CPU1 is using fixed priority scheduling. Four virtual CPUs are scheduled for execution on CPU1, VCPU1 of partition P0, P1 and P2 and VCPU0 of P3. VCPU1 of P1 has the highest priority, and VCPU0 of P3 has the lowest priority. As shown on the figure, interrupts I0, I1 and I2 are respectively allocated to VCPU1 of P0, VCPU1 of P1 and VCPU1 of P2. When interrupt I0 is triggered, VCPU1 of P0 becomes ready for execution; therefore the kernel performs a context switch and executes VCPU1 of P0. This virtual CPU executes for a while until it yields back the CPU, VCPU0 of P3 is still ready for execution so it is scheduled for execution. When interrupts I0 and I1 are triggered simultaneously, VCPU1 of P1 is scheduled for execution before VCPU1 of P0 since it has a higher priority.

Another new feature of Xtratum version 4.0 is the support of the NGMP IOMMU. The system integrator can allocate an I/O device to a partition. Then the partition memory address space will be accessible from the I/O device, and only from this I/O device. This feature allows implementing the direct I/O assignment technique as described in section 3.3.1. Using this technique, the system integrator can delegate the management of the I/O units to the partitions including I/O units with DMA engines, while ensuring the space partitioning. No partition can program its I/O unit to modify the memory address space of another partition.

4.2 Multi core use cases

To this day and to our knowledge, the need for a multi core execution platform has never been expressed to implement an actual space application. On a spacecraft, processing units are required typically to implement three broad kinds of functionality. The first kind is the execution of central flight software which is in charge of the managing the spacecraft platform and implementing control and guidance algorithms. State of art satellites, like modern agile earth observation satellites, do not require more processing power than a standard LEON processor clocked at 32MHz provides. The second kind of functionality that requires processing units is the instrument/sensor control functionality. Some complex sensors like star trackers or navigation units require a processor to manage its hardware and implement specific algorithms. State of art star trackers use a LEON processor clocked at 96MHz. The last kind of functionality is the payload data processing functionality. General purpose processors are seldomly used to implement this functionality as the required processing performance is much higher that what can be provided with a general purpose processor. ASIC are regularly used to implement the data processing functionality. For small and simple payloads, it may be possible to implement the data processing functionality using an advanced general purpose processor.

As explained in the previous paragraph, with the current spacecraft system architecture, there are no actual needs for multi core processors. However new system concepts, like the Integrated Modular Avionics approach could trigger the need for multi core processors. In the two next sections, we present two possible use cases of multi core processor for space applications and provide a few recommendations for an efficient implementation.

4.2.1 Extended IMA

The IMA approach aims at regrouping on a single hardware the execution of different software. For instance, star tracker processing, payload control processing, and central flight software could run on a single processor. In order to implement this approach, time and space partitioning techniques are used, and Astrium has evaluated the use of the Xtratum hypervisor to implement this.

From the use case evaluation of IMA on mono core processor, we learned two things: first the performance impact of the IMA architecture is not negligible (up to 30% of the application CPU time is wasted in IPC communications and overheads of the hypervisor), and second, the scheduling plan is very constrained by I/O interactions.

LEON based multi core platforms are well suited for implementing IMA applications: since as of today no single partition would require more processing power than what can provide a single CPU core, each partition could be allocated one core of the multi core platform. For instance we would execute the CSW on one core, star tracker processing on another, and instrument control on a third one. The hypervisor would be used to ensure space partitioning of the memory and the I/O devices. In this frame, the direct I/O assignment technique is the one providing the best results: each partition can manage its I/O

hardware individually from the others. Xtratum support of the IOMMU allows implementing this technique and we have successfully prototyped the functionality in the frame of this study.

The main problem with this approach on the current NGMP / Xtratum multi core platform is the hardware coupling between the processing cores. This coupling has the effect of breaking the time partitioning between the partitions: one partition execution can influence another partition's execution timings on another core through the hardware coupling. The only solution today would be that the system integrator revalidates the system at integration time, making sure each partition has enough time to execute its tasks. This breaks the IMA approach, as in IMA the objective is that each partition is validated independently. Another solution would be that at design time, the system integrator allocates to each partition a CPU margin; this margin could be used by partitions to finish their execution when it has been delayed by the execution of another partition on another core. In fact, the system integrator should evaluate of the Worst Case Execution Time of each partitions' tasks, when executed on the multi core platform at the same time as other partitions. No tools or techniques are available today to perform such WCET analysis in multi core environment; it's the subject of several R&D activities, the most promising ones being the ones focusing on a probabilistic approach to WCET analysis like the PROARTIS and ProCXim activities.

In cases where more partitions than processing cores are needed, for instance when the central flight software is split other several partitions (eg: one partition for AOCS, one partition for platform management), it could become necessary to share a processing core among several partitions. We have learned from the IMA use case on mono core that in these cases, the scheduling plan becomes very constrained by the I/O interactions of each partition. Indeed the system integrator must plan when each partition has to perform I/O interactions, and build the scheduling plan so that each partition is scheduled for execution when it has to perform I/O operations. This could results in high frequency context switches, late scheduling plan changes, and overall an inefficient use of the execution platform.

We believe the new fixed priority scheduling policy developed in the frame of this study and presented in section 4.1 could help releasing the constraints on the scheduling plan. We have prototyped a system in which each partition's main tasks are scheduled for execution on one core of the platform using a regular cyclic scheduling policy, and in parallel each partition's I/O tasks are scheduled on another core using a priority based scheduling. By doing that, the constraints on the cyclic scheduling plan are released as I/O operations are scheduling for execution in a dynamic fashion on the core using fixed priority scheduling. Since I/O tasks are executed in the same memory address space as the main partitions tasks, there are no communication overheads when exchanging I/O data.

This approach though promising needs to be further developed before being fully operational. First, real time operating systems like RTEMS need to support the concurrent execution of I/O management tasks. Second the fixed priority scheduling policy needs to be updated in order to ensure a better timing isolation between the cores. With the current status of the fixed priority scheduling policy, there are no

protections against an I/O task overrun: one partition's I/O task can hog the CPU, preventing lower priority partitions from execution.

All in all, we believe multi core platform can be used to build powerful IMA systems by using the newly developed features like direct I/O assignment and fixed priority scheduling, however some further developments are still needed to make the approach fully viable.

4.2.2 Data processing

Some scientific payloads only generates a few Mbit/s of data, the data processing part of these kinds of payload could be implemented on general purpose multi core processor. Moreover, an integration of the instrument control functionality with the data processing functionality could be foreseen.

The NGMP / Xtratum platform could be efficiently use to implement these kinds of applications. For instance, three cores could be allocated to a data processing partition and one core to an instrument control partition. Each partition would be responsible for managing its own I/O devices under a direct I/O assignment scheme.

The main difficulty consists in the implementation of the data processing application in a multi threaded environment. For that purpose we think the task parallelism technique as presented in section 3.2.2 should be used preferentially. This technique allows keeping a better control on the task allocation between the processing cores, and therefore allows minimizing the hardware conflicts, when cores are trying to access a shared hardware resource. However software tools appropriate to deploy such technique on the Xtratum / NGMP platform are not available today. We therefore recommend that one of the existing execution runtime implementing the MPI API be ported on top of the Xtratum / NGMP platform. We believe the MPI family of execution runtimes is better suited to implement applications based on the task parallelism approach.

5. CONCLUSION

The introduction of multi core execution platforms for space applications will not be a simple task, and will have important impacts on the systems design. At the time of writing of this document, the first multi core execution platform becomes available to the European space community. It's a first iteration, of both the hardware and the supporting software. We are now in a position where we have learned from this first iteration, and are able to provide recommendations for the next iteration and foresee the introduction of multi core execution platform in actual missions. The NGMP hardware should be improved in order to reduce the hardware coupling between the processing cores, however we must be well aware that hardware coupling cannot be completely eliminated on multi core processors. We must therefore develop tools and techniques allowing a reliable worst case execution time analysis for multi core processors. Implementing the IMA approach could be simplified by using a hypervisor / multi core hardware platform, however the hypervisor must be fully multi core aware and provide specific functionalities like virtual CPUs, fixed priority scheduling policies and IOMMU support. Light data processing applications could also be implemented on a multi core platform, but for that specialized parallelization libraries should be ported on the multi core hardware.

All in all, it is clear that the move to multi core platform has been a bit anticipated by hardware manufacturers compared to the actual needs, however future applications may benefit from multi core providing the fact that some further developments and hardware improvement are implemented.

Distribution list	OVERALL DOCUMENT		SUMMARY
	ACTION	INFORMATION	