Chapter 4

# RESOURCE RESERVATIONS IN SHARED-MEMORY MULTIPROCESSOR SOCS

Clara Otero Pérez, Martijn Rutten[1],

Liesbeth Steffens, Jos van Eijndhoven, Paul Stravers

*Philips Research Laboratories, Eindhoven, The Netherlands*
*[1] Philips Semiconductors, Eindhoven, The Netherlands*

Abstract: Consumer electronics vendors increasingly deploy shared-memory multiprocessor SoCs, such as Philips Nexperia, to balance flexibility (late changes, software download, reuse) and cost (silicon area, power consumption) requirements. With the convergence of storage, digital television, and connectivity, these media-processing systems must support numerous operational modes. Within a mode, the system concurrently processes many streams, each imposing a potentially dynamic workload on the scarce system resources. The dynamic sharing of scarce resources is known to jeopardize robustness and predictability. Resource reservation is an accepted approach to tackle this problem. This chapter applies the resource reservation paradigm to interrelated SoC resources: processor cycles, cache space, and memory access cycles. The presented *virtual platform* approach aims to integrate the reservation mechanisms of each shared SoC resource as the first step towards robust, yet flexible and cost-effective consumer products.

Key words: Virtual platform, multiprocessor system, shared resources, shared memory

## 1. INTRODUCTION

The convergence of consumer applications in the TV, PC, and storage domains introduces new combinations of features and applications that execute in parallel. In addition, consumer multimedia devices are becoming increasingly flexible. Flexibility enables accommodating late changes in

standards or product scope during system design, and allows in the field upgrades. To address the flexibility and concurrency requirements, consumer electronics vendors increasingly deploy heterogeneous multiprocessors systems.

The high production volume of consumer products sets severe requirements on the product cost, leading to resource-constrained devices. To achieve a cost effective solution, expensive resources, such as memory and processor time, are shared among concurrent applications.

A typical multimedia application consists of independently developed subsystems with strong internal cohesion. At the subsystem borders, the real-time requirements are decoupled from the other subsystems. However, resource sharing induces temporal interference among otherwise temporal independent subsystems given the highly dynamic workload of the targeted media applications, such as audio/video coding, image improvement, and content analysis. *Figure 4-1.* depicts the load fluctuations (in time) of two independent subsystems sharing a resource. At a given point in time, both subsystems require more resources than the total available and one (or both) of the subsystems will suffer.

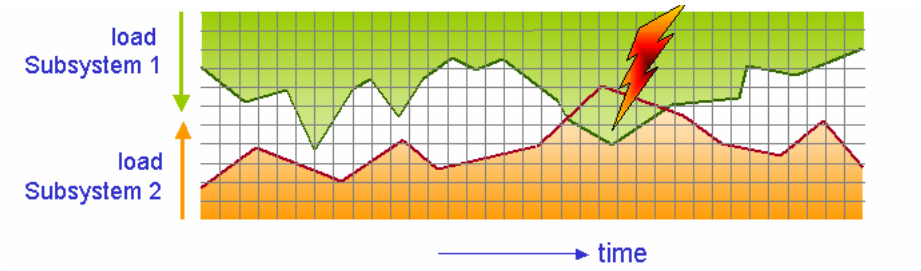


*Figure 4-1.* Load fluctuations of independent subsystems.

The concurrent execution of dynamic applications on shared resources is a potential source of interference, which leads to unpredictability and jeopardizes overall system robustness. We aim to bound interference by isolating and protecting independent subsystems from each other, while preserving typical qualities of multimedia devices, such as robustness and cost effectiveness.

Resource reservation is a well-known technique in operating system research to improve robustness and predictability. It is based on four components, admission control, scheduling, accounting, and enforcement. When properly combined, they provide guaranteed resource reservations. Resource reservations consist at least of two basic parameters: share and granularity. For example, a share of 5 milliseconds and a time granularity of

20 milliseconds determine a processing time reservation. Different resources have various types and degrees of share-ability. For example, a programmable processor is shareable in time and fully preemptable. An MPEG-2 (video coding) hardware accelerator may decode one high-definition stream or decode two standard-definition streams in time-shared fashion. Memory is shared in space.

To bound interference at system level, the multimedia device must provide the subsystems with a reservation mechanism for each resource. By configuring the resource reservations, we create an execution platform that is tailored to the resource needs of the subsystem. We term this a *virtual platform*. A virtual platform provides guaranteed resource availability, while restricting resource usage to a configured maximum.

## 2. RELATED WORK

Multiprocessor systems on chip (SoC) are rapidly entering the high-volume electronics market. Example SoC platforms are Philips Nexperia (de Oliveira & van Antwerpen 2003), Texas Instruments OMAP(Cumming 2003), and STMicroelectronics StepNP (Paulin, Pilkington, & Bensoudane 2002). A mayor challenge for these multiprocessor systems is to effectively use the available resources and maintain a high degree of robustness. Currently, these systems do not explicitly address interference between software modules that compete for shared system resources. The robustness problems caused by interference are typically evaded by a high degree of over provisioning.

Various research efforts address interference for specific resources through processor resource reservations (Lipari & Bini 2003), (Eide et al. 2004), (Baruah & Lipari 2004), interconnection guarantees, Chapter 1 of this book (Goossens & González Pestana 2004), and cache partitioning (Liedtke, Haertig, & Hohmuth 1997),(Molnos et al. 2005). For instance, Ravi (Ravi 2004) presents various mechanisms for cache management based on priority assignment and enforcement. Recent research aims for integrated approaches that considers combination of resources such as processor and network reservations(Rajkumar et al. 2001), (Nolte & Kwei-Jay 2002). We take one step further and develop an integrated approach to bound interference for *all* shared resources in upcoming multiprocessor systems. Our multi-resource reservation is the base for defining an embedded *virtual* platform.

Numerous systems have been designed which use virtualization to subdivide the ample resources of a modern computer since IBM introduced the 360 model 67, in 1967. In a traditional virtual machine (VM), the virtual hardware exposed is functionally identical to the underlying machine

(Seawright & MacKinnon 1979). However, full virtualization is not always desired. Some of the disadvantages lead to a performance penalty that current high volume electronics vendors are not willing to pay. One of the goals of recent research on virtualization is to overcome these disadvantages. Rather than attempting to emulate some existing hardware device, the Xen VM research of Barham et al. (Barham et al. 2003) exposes specially designed block (device and network) interface abstractions to host operating systems, in what they call *paravirtualization*. Barham et al. assume full resource availability. It is not clear whether or how their approach provides guarantees and performs admission control. The severe cost requirements in the consumer electronics domain oblige us to provide resource guarantees for resource-constrained systems.

The remainder of this paper is organized as follows. Section 3 describes a embedded system consisting of a multiprocessor SoC in which concurrent media applications execute. The interference problem is explored in Section 4 and the concept of virtual platform as a solution to bound interference is introduced in Section 5. Section 6 presents the different reservation mechanism for the three main SoC resources (processor cycles, cache space and memory access cycles) used to implement the virtual platform. Finally, the conclusion is drawn in Section 7.

## 3. MULTIPROCESSOR SYSTEM

Multiprocessor SoCs are deployed to cope with the market demand for high performance, flexibility, and low cost. Progressive IC technology steps reduce the impact of programmable hardware on the total silicon area and power budget. This permits SoC designers to shift more and more functionality from dedicated hardware accelerators to software, in order to increase flexibility and reduce hardware development cost. However, for at least the coming decade, these multiprocessor SoCs still combine flexibility—in the form of one or more programmable central processing units (CPU) and digital signal processors (DSP)—with the performance density of application-specific hardware accelerators. *Figure 4-2* depicts such a heterogeneous SoC architecture as presented in Chapter 5 of this book (van Eijndhoven et al. 2005) and (Stravers & Hoogerbrugge 2001). In providing a virtual platform for upcoming SoCs, we have to cope with the interaction between processing in hardware and software.

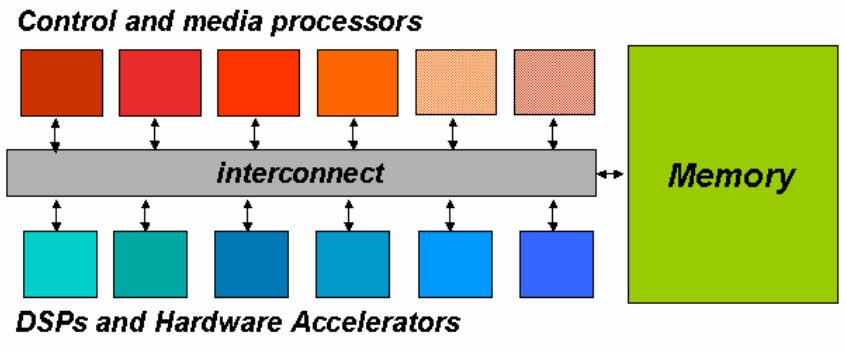


*Figure 4-2.* Heterogeneous SoC architecture with CPUs, DSPs, and accelerators communicating through shared memory.

With progressive technology steps, processing power and memory sizes increase, keeping the pace with the memory and processing capacity requirements imposed by media applications. In contrast, memory bandwidth scales slowly and memory latency remains almost the same. Thus, memory bandwidth and latency are becoming the dominant system bottleneck.

*Figure 4-3* details the data path of a multiprocessor such as in *Figure 4-2*, in which a number of DSPs, CPUs, and accelerators communicate through shared memory. The architecture applies a two-level cache hierarchy to reduce memory bandwidth and latency requirements. The cache hierarchy is inclusive: a memory block can only be in a L1 cache if it also appears in the L2 cache. When a processing unit produces new data and stores it in its L1 cache, the L2 copy of that memory block becomes stale; in such cases a cache coherence protocol ensures that any consumer of the data always receives the updated L1 copy of the data. Furthermore, such a coherent communication network allows direct L1-to-L1 cache transfers, e.g. when a consumer task on processing unit A reads data from a producer task on processing unit B. At any moment in time, a modified data item resides only in one L1 cache. This property is intended to facilitate the partitioning of applications, consisting of multiple producer/consumer tasks, over multiple processors.

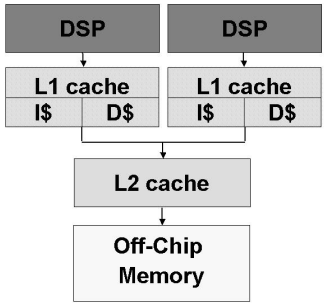


*Figure 4-3.* Data path for the memory hierarchy.

The applications we are dealing with are media applications (mainly audio and video). *Figure 4-4* depicts an example of a media application (Otero Pérez et al. 2003). Such applications are also known as streaming applications, because they process streams of data. A stream is a sequence of data objects of a particular type (audio samples, video pictures, video lines, or even pixels). For example, a video stream is a sequence of pictures, with a given picture rate: the number of pictures to be displayed per second. A stream is typically produced by one streaming task and consumed by some other concurrent asynchronous streaming task. The part of the stream that has been produced but not yet consumed, is temporarily stored in a buffer, or is being transferred, from producer to buffer, or from buffer to consumer.

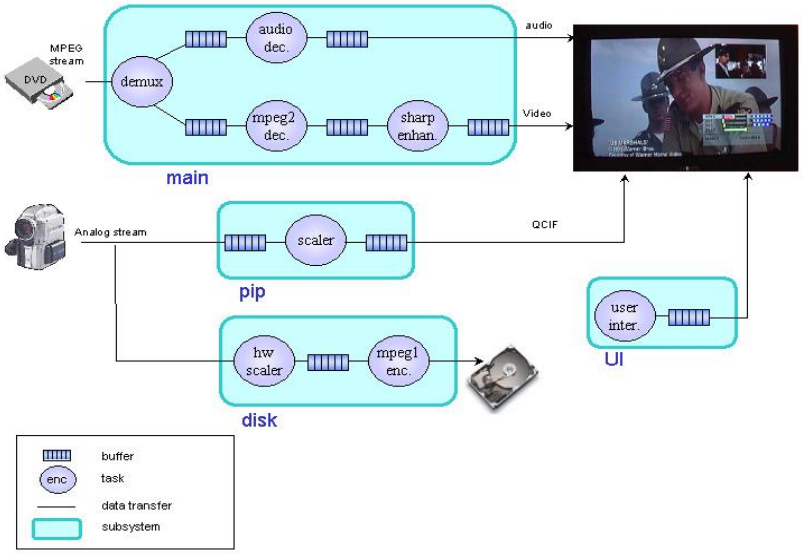


*Figure 4-4.* Media application example.

Our execution model for streaming applications consists of a connected graph in which the nodes represent either a task (an independent, asynchronous, active component that uses processing and memory resources) or a buffer (a passive component that uses memory resources). The interconnections represent the data transfer (memory access). The execution model is hierarchical. At higher levels of abstraction, a connected graph can again be viewed as a subsystem in a connected graph. *Figure 4-4* depicts four such subsystems: main, pip, disk, and user interface (UI). The subsystems are denoted with the rounded rectangles.

## 4. INTERFERENCE AMONG SUBSYSTEMS

The pressure on time-to-market, the emergence of multi-site development, and the ever-increasing size of software stacks are just some of the factors that enforce a radical change in the development of modern (multimedia) applications. In the past software systems were almost completely written from scratch as fully self-contained systems, developed under one roof. These days, systems are increasingly composed of independently developed subsystems, originating from different locations and in many cases from different companies. These subsystems are not designed as a specific part of a whole, but are intended to be deployed in many different systems, and serve different ranges of products.

Ideally, each subsystem is evaluated and tested in isolation for a specific system. The job of the system integrator is to mix and match the subsystems to compose the final system. Unfortunately, current subsystems are not compositional. The ad-hoc and implicit way in which the scarce SoC resources are managed, and the unbounded interference caused by resource sharing, introduces temporal interdependencies among these initially independent subsystems. If not properly managed, these interdependencies lead to unpredictable behavior for the integrated system.

Media-processing SoCs rely on priority scheduling in embedded real-time operating systems, such as VxWorks and pSOS, to manage real-time requirements. The current setting of priorities is an example of ad-hoc management. Traditionally, priorities were used to manage resource utilization in closed, real-time systems, where task activations and execution time are deterministic. Under these conditions, well-known priority assignment methods, such as rate monotonic assignment (Liu & Layland 1973), work fine. However, media-processing tasks tend to violate many of these assumptions, e.g., by generating idle time, dynamically fluctuating workloads, jitter, etc. The subsystem designers have to rely on trial and error to obtain a working system. Moreover, at integration time, when all tasks in

all subsystems come together, the integrator has to start again from scratch. The priority assignment of the subsystem tasks cannot be reused in the integrated system, and the system integrator is faced with the difficult task of evaluating different priority settings, while other factors such as importance of the task or response time requirements influence the priority assignment.

A second example is the priority assigned to the various processors for bus access. The processor's bus priority is fixed and unrelated to the tasks executed by the processors. Oftentimes, the priority setting is based on a complex relation among the various tasks that might execute on that processor.

The use of a cache introduces a third example of unpredictability, due to the difficulty in predicting when certain data is available in the cache or still has to be fetched from off-chip memory, causing the processor to stall. Interrupts in combination with caches are a further cause of unbounded interference. A typical system relies on interrupts to activate hardware accelerator, to handle exceptions, to wake up software tasks, etc. An interrupt causes a context switch, evicting the running task from the processor and invalidating the task data present in the cache. When the running task resumes execution, potentially all its data has to be fetched again from memory. Therefore, it is very difficult to determine an upper bound for the performance impact caused by interrupts in cache-based systems.

We conclude from the previous paragraphs that resource management based on bounding interference constitutes the foundation for compositional system design. We propose an integrated approach to resource management based on guaranteed resource reservation for all shared SoC resources, tailored to the needs of the resource consumers (subsystems). The concept of a virtual platform—as outlined in the next section—summarizes our approach towards such integration.

## 5. VIRTUAL PLATFORM

A virtual platform provides guaranteed resource availability, while restricting resource usage to a configured maximum. Like the real platform, a virtual platform provides a wide variety of resources: programmable processors, function specific hardware, memory space, memory access bandwidth, and interconnect bandwidth. In a SoC, a virtual platform can be implemented in various ways. For example, a set of tasks can execute concurrently on multiple slow processors or sequentially on a fast processor.

The implementation of a virtual platform is based on resource reservation mechanisms that provide temporal and spatial isolation among subsystems.

The resource manager is responsible for providing virtual platforms by ensuring that sufficient resources are reserved. For that, a resource reservation mechanism, for each main SoC resource, guarantees the availability of resources. As depicted in *Figure 4-5*, the resource manager translates subsystem requirements and sets the parameters for the virtual platform. This requires appropriate knowledge of the demands of the individual subsystems in terms of the specific platform resources. Characterizing performance and behavior of the subsystems is a subject of research, fundamental to the realization of a virtual platform.

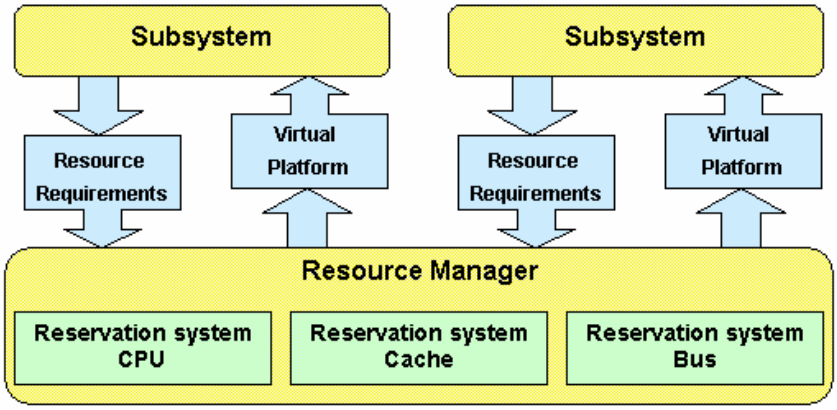


*Figure 4-5.* Subsystems and virtual platforms.

Furthermore, to actually deploy the virtual platform concept, the following three issues must be resolved. Firstly, to provide a virtual platform, the resource manager has to coordinate the resource reservations for each resource. For that, the interdependencies among resources must be modeled and analyzed. The effective CPU speed depends on the reservations made in the memory architecture, such as cache and bus bandwidth. For example, a memory controller that schedules processor requests to memory guarantees a given average latency for a given processor. This latency is used to calculate the execution time of a subsystem on this processor and determines its processing budget.

Secondly, the reservation of a resource in the resource hierarchy may not be based on the virtual platform using the resource, but on the actual physical components using this resource. An example is memory bandwidth. This bandwidth is allocated to the physical processors accessing the memory independently from which virtual platform this processor is allocated to. As a virtual platform is, in general, implemented by several physical processors, a complex hierarchical set of interdependencies is created. These

interdependencies are very difficult to understand and to analyze. Note that this complexity is not introduced by the virtual platform concept itself, but is inherently present in current SoC architectures and must be solved independently of the virtual platform.

Finally, given the dynamic behavior of the software, absolute guarantees are only possible when the reservations are based on worst-case load. For cost effectiveness reasons, this is unfeasible even if the worst-case load would be known (which is typically not the case). Structural load fluctuations can (to a limited extent) be addressed in the virtual platforms by reallocating unused reservations or dynamically adapt the reservations to increase/decrease the virtual platform capacity. However, high-volume electronics products stress the platform resource utilization to the limit. At a given point, the required load of the concurrently executing subsystem will exceed the resource capacity and some subsystem will experience a resource shortage. Resolving temporal overloads within a subsystem is specific to each subsystem; it is therefore the responsibility of the subsystem to resolve this.

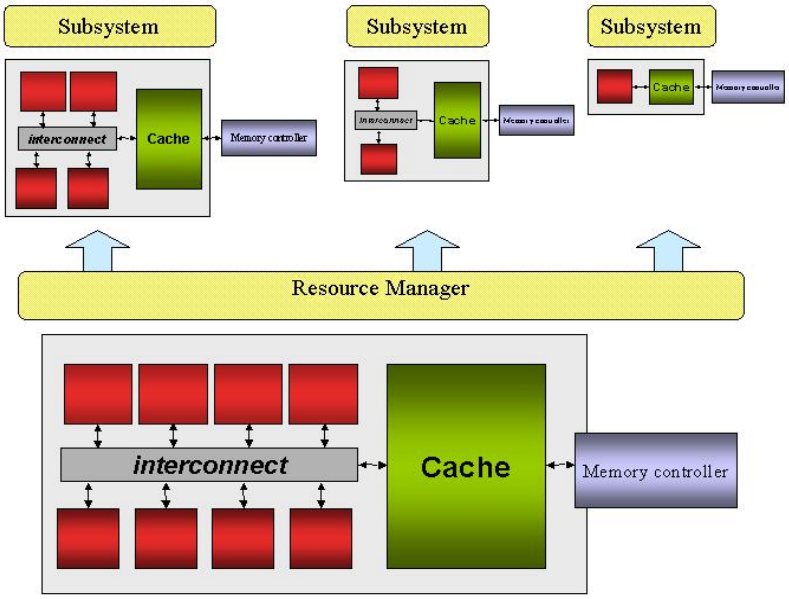


*Figure 4-6.* Providing virtual platforms to subsystems.

The following section presents the first step towards implementing a virtual platform: the resource reservation mechanisms for the three main SoC resources: CPU cycles, cache space and memory access cycles. *Figure*

*4-6* depicts the virtual platform vision, where the resource manager provides each subsystem with its own virtual platform, which are a share of the real SoC.

# 6. RESOURCE RESERVATION MECHANISMS

Resource reservation is a well-known technique to implement temporal and spatial isolation and to bound interference. A *resource budget* is a guaranteed resource reservation. The resource reservation mechanism consist of the following four components, identified in (Oikawa & Rajkumar 1998).

- The scheduling/arbitration/allocation algorithm determines the run time execution. The scheduling algorithm is such that it matches the budget requirements.
- Accounting keeps track of budget usage.
- Enforcement, denying resource availability when the budget is exhausted, is required to provide guarantees.
- Admission control ensures that once a reservation has been accepted by the system, the budget will be guaranteed.

Sections 6.1 through 6.3 detail these four components of the resource reservation mechanism for the three main SoC resources: processing cycles, cache space, and memory access cycles.

## 6.1 Processing cycles

Multiprocessor SoCs embed various providers of processing cycles, from a dedicated, non-shareable MPEG-2 accelerator to a multitasking programmable DSP. Managing resource reservations on a multitasking resource—shared by many tasks with diverse real-time requirements—is more challenging than managing access to a hardware accelerator that typically can handle only one task. Therefore, we focus on multitasking programmable processors.

There are different implementations of processor reservation mechanisms (Lipari & Bini 2003), (Eide, Stack, Regehr, & Lepreau 2004),(Rajkumar, Juwa, Molano, & Oikawa 2001). We present our approach to processing cycles reservations in the following subsections.

**Resource users**

The users of processing cycles are the software subsystems, where the subsystem is temporally independent from other subsystems. Typically,

subsystems consist of collections of connected tasks. We distinguish two types of subsystems.

- *Media processing*. This type of subsystem processes media streams with a highly regular pattern. A video decoder for example, produces a video frame every 20 milliseconds. The behavior of a media processing task can be described by a request period *T*, execution cycle requirement *C*, and a deadline *D*, where $D = T$.
- *Control*: Control subsystems have an irregular activation pattern with a minimum inter-arrival time, and their expected response time is short (compared with the inter-arrival time). They can be described by a minimum inter-arrival time *T*, a processing-cycles requirement *C*, and a deadline *D*, where $D << T$.

**Budget definition**

CPU-cycle budgets are provided to subsystems and must match the CPU cycle requirements of the subsystems, as described in the previous paragraph. Media processing subsystems typically require periodic budgets with a budget value *C* (number of processing cycles), a granularity *T* (period of activation), and a deadline *D*. A periodic budget is replenished at regular intervals. Control subsystems typically require sporadic budgets which provide a limited amount of computation budget, *C*, during a time interval called the budget replenishment period, T. The sporadic budgets preserve and limit a certain amount of CPU cycles for the control subsystems, while guaranteeing the deadlines of all the other subsystems in the system, even under burst conditions in the activation of control subsystems (i.e., large number of requests in a short time interval). The sporadic budget is easily incorporated into rate monotonic analysis (Klein 1993), because aperiodic activations can be analyzed as if they were periodic.

Budgets can be strictly enforced, the subsystem receives only its requested C per T, or weakly enforced, a subsystem may receive more than requested if all other subsystems are out of budget. The advantage of strictly enforced budget is predictability: the subsystems always receive the same budget. The advantage of weak enforcement is high utilization.

**Scheduling algorithms**

The reservation algorithm for CPU cycles can be based on rate monotonic scheduling (RMS) or earlier deadline first (EDF) scheduling. These algorithms, (Liu & Layland 1973) were initially conceived for independent executing task. In our case, individual tasks are not independent

whereas subsystems are. The same reasoning that used to apply to tasks applies now to budgets.

- *Rate monotonic scheduling.* Given fixed-priority scheduling, the optimal priority assignment for periodic budgets is the *rate monotonic* (RM) priority assignment. Budgets are ordered by increasing period, ties broken arbitrarily, i.e., $i < j \Leftrightarrow T_i \leq T_j$. The budget scheduling mechanism is built on top of a regular fixed priority scheduler. At the start of each period, the priority of all tasks within the subsystem is raised to the subsystem's running priority. When the subsystem budget is depleted, the subsystem's priority is lowered to background priority.
- *Earliest deadline first scheduling.* In earliest deadline first (EDF) scheduling, budgets are dynamically ordered by increasing deadlines, ties broken arbitrarily. At the start of each period, the deadline of the budget is set. The selected budget is the one with the earliest deadline among the non-zero budgets.

In the case of weak enforcement, when all budgets are exhausted, a slack allocation mechanism is used to immediately allocate the otherwise wasted, volatile, processor cycles. For example, in the case of fixed-priority scheduling, a very simple slack-allocation algorithm consists of making all budgets eligible for execution on a round-robin basis, by giving them the same background priority.

**Accounting**

Accounting takes place in the CPU reservation module, which maintains a subsystem descriptor per subsystem. This subsystem descriptor contains a down counter that keeps track of the processing cycles used by the subsystem. Every task context switch, the accounting system determines which task (and which subsystem) has executed and for how long. The corresponding amount is deducted from the budget counter. The processor clock is used to keep track of the time.

**Enforcement**

The budget is enforced by using high precision (hardware) timers that are fired when a budget is exhausted or when a budget has to be replenished.

**Admission control**

For a single processor system, we use an admission control algorithm that corresponds to the scheduling algorithm being used. If the admission control

fails, the corresponding budgets cannot be guaranteed, therefore the subsystem corresponding to the budgets that causes the failure is not allowed to start (or to modify its resource requirements). When using RMS as scheduling algorithm a simple formula (1) for response time calculation from (Joseph & Pandya 1986) is used:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil * C_j \leq D_i . \quad (1)$$

In this formula, index $i$ identifies the budget, $T_i$ is the period, $C_i$ is the budget capacity, $R_i$ is the response time, and $hp(i)$ is the set of all budgets with priority higher than $i$. When using EDF, an even simpler capacity check (2) is used:

$$\sum_{all\ i} \frac{C_i}{T_i} \leq 1 . \quad (2)$$

Note that it does not make sense to provide budgets to individual tasks in a single subsystem, because the temporal interdependencies among the tasks invalidate these acceptance tests assumptions.

## 6.2 Cache space

Caches are divided into cache lines, also called blocks. Cache lines are grouped into sets. A memory location is mapped to a cache set depending on its address and it can occupy any line within that set. A cache with 1 line per set is called direct-mapped, a cache with k lines per set is called k-way set-associative, and a cache with only 1 set is called fully associative. When a line is loaded into the cache, the address determines the set into which the line is loaded. In a direct mapped cache, there is only one choice for replacement, determined by the address. In a k-way set-associative cache, there are k lines that can be victimized.

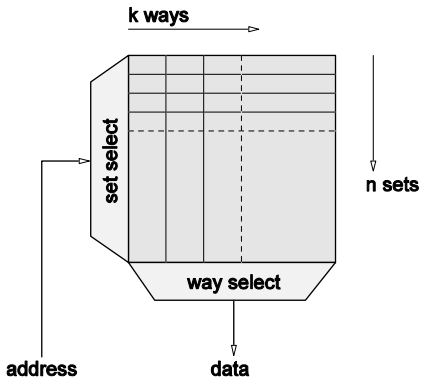


*Figure 4-7.* Generic cache architecture.

*Figure 4-7* shows a generic k-way set-associative cache architecture. The address of a load or store operation is first translated into a *set index* that uniquely identifies the set where the data is cached (if it is cached at all). Within each set there are *k* blocks. An associative search, *tag matching*, is required to determine which block, if any, contains the data corresponding to the specified address. If the addressed block is not found, one of the *k* blocks in the set is *victimized*: dirty data is copied from the victim block to memory, while the requested data is copied from memory to the victim block.

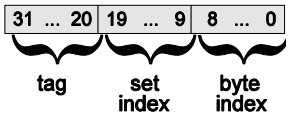


*Figure 4-8.* L2 cache addressing.

*Figure 4-8* depicts how a cache block is addressed. Tag matching is used to locate the place within the set where the data is placed. Since tag matching is performed on the most significant bits, the data of e.g. one MPEG frame is distributed over all sets. Thus, a linear memory access pattern results in a uniform distribution of accesses over the cache.

A known issue with hardware-managed caches is the interference between multiple independent software tasks that share the same cache. Various approaches in the literature address this problem. See, for example (Liedtke, Haertig, & Hohmuth 1997;Molnos, Heijligers, Cotofana, & van Eijndhoven 2005;Ravi 2004).

In the SoC depicted in *Figure 4-2*, there are two types of caches: L1 and L2. The L1 cache is shared when the corresponding CPU supports

multitasking. Upon a task context switch, new task data is loaded into the cache, evicting the exiting-task data out from the cache. When the evicted task is executed once again, its data has to be reloaded, causing an extra performance penalty compared to the case when the task runs without interruption. However, in a multiprocessor system, the CPUs are typically dedicated to a small number of tasks: for example, one CPU takes care of coprocessor management, another takes care of network traffic, etc. As a result, the remaining CPUs can concentrate on running applications without being interrupted by housekeeping jobs. Consequently, L1 cache interference appears to be a less urgent problem in a multiprocessor than it is in a single CPU system.

For the L2 cache, a different story applies. This single cache is shared concurrently by the tasks in the system. For example, a Linux operating system executing on one or more CPUs may suddenly require a lot of memory when it starts an Internet browser with Java support. We want to avoid that this action in the Linux domain evicts critical data and code sections in another domain, for example a real-time video codec running on the DSPs sharing the same L2 cache. We define a *domain* as the collection of tasks that share a determined cache space.

In this section, we focus on a cache management mechanism for the L2 cache. The following sections describe our approach to cache management to bound interference among the various application domains that execute concurrently on the multiprocessor.

**Resource users**

The users of the cache are the cache domains or collection of software tasks. The software tasks request from the cache load (read) and store (write) operations. This operation can result either on a hit (the requested data is cached) or a miss (the requested data is not in cache). Upon a cache miss, data has to be brought from main memory victimizing cached data.

**Budget definition**

Available cache partitioning methods (Liedtke, Haertig, & Hohmuth 1997;Molnos, Heijligers, Cotofana, & van Eijndhoven 2005) allocate parts of the available cache sets exclusively to a subset of the executing tasks (*Figure 4-9*). The disadvantage of this approach is that it affects the memory model as seen by the software programmer. For example, if task A writes to memory location X, the data is cached in partition A. If task B reads from memory location X at a later moment in time, it cannot find the data in cache

partition B and consequently the stale data is loaded from memory into partition B. This is probably not what the programmer expected.

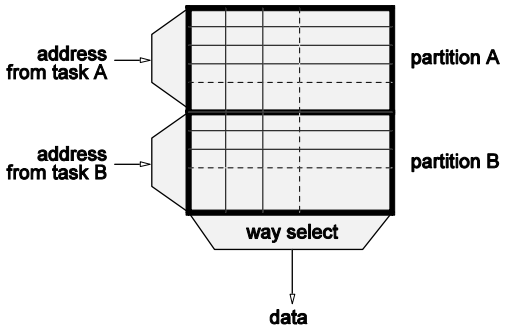


*Figure 4-9.* Traditional cache partitioning.

In contrast to set partitioning, we chose to partition the cache by limiting the number of *ways* a task can claim within each set in the cache. *Figure 4-10* depicts the resulting cache organization. Cache resource management is performed by allocating cache space to a domain. During cache lookup, all ways in the set are considered, including the ways associated with domains other than the one performing the lookup, i.e., any task can read or write any cache block with no restrictions. Consequently, the shared memory model remains intact. The programmer does not notice any functional difference between a traditional and a resource-managed cache.

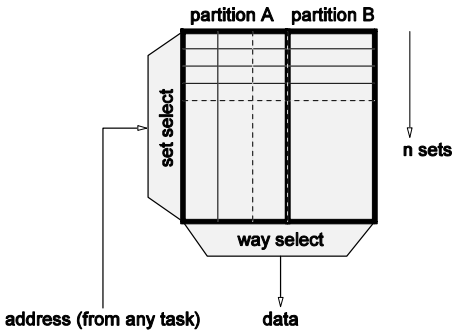


*Figure 4-10.* Cache way partitioning.

A cache budget determines the maximum number of ways the domain can claim, say *N*. If each way corresponds to a fixed number of bytes, say *M*, each budget corresponds to *N*M* bytes. The cache contains a bit vector for

every domain, where each way in the cache is associated with one bit in these vectors. By setting the bit vector in the cache, the system integrator can choose explicitly which domains share which cache ways.

The task descriptor in the OS contains a field that identifies the domain the task belongs to. On every context switch, the OS copies this field to a hardware register. On a cache miss, the cache controller inspects this register to determine which domain is causing the miss and the victim is selected from the ways belonging to this domain.

There are two types of reservations. Cache reservations can overlap (domains can share ways) or be disjoint (no ways are shared). When two domains share a way, in the worst case, one of these domains can evict all data of the other domain from the cache. If all domains reserve disjoint ways, the reservation is not shared: tasks belonging to domain A cannot evict data from domain B. Overlapping domains are useful when the worst case cache requirement is far from the average. Each domain reserves disjoint space to be used during normal behavior and overlapping ways for the worst case.

**Replacement algorithm**

The replacement algorithm, that selects which cache block is victimized, is the equivalent of the scheduling algorithm for the CPU. The cache employs a random replacement strategy. When a task belonging to domain causes a refill, a victim block is selected from the corresponding domain, such that the number of ways associated with the domain in the set does not exceed the predetermined budget for the domain. The replacement algorithm only applies during a cache refill, following a cache miss.

**Accounting**

Accounting has to keep track of the number of ways allocated to a particular domain in a particular set of the cache.

**Enforcement**

If a bit is set in the vector of a selected domain, the domain can access the cache blocks in the way corresponding to the bit position in the vector.

**Admission control**

The admission control for a cache reservation request is simple. The total amount of requested space should not exceed the total cache size.

## 6.3 Memory access cycles

As presented in Section 3, data transfer to and from memory is becoming the main system bottleneck. As an example, *Figure 4-11.* depicts the structure of the memory path of the SoC. The memory controller has three available ports. Two of these ports are used by the refill and victim engines of the L2 cache. The third port is used by the hardware accelerators.

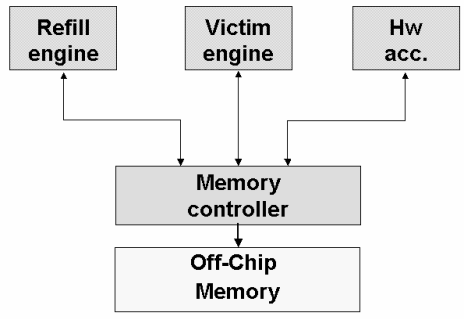


*Figure 4-11.* Memory controller ports.

This section focuses on the memory access cycles: the cycles available for data transfer from the memory controller ports to the off-chip memory. Similar to processing cycles, memory access cycles constitute a volatile resource: a memory access cycle that is not granted to a requester is lost forever. However, the allocation granularity for memory access cycles is a few orders of magnitude smaller than the allocation granularity for CPU cycles.

The order in which requests are presented to the memory has a large impact on the efficiency of the memory access. For example, if a write transfer follows a read transfer, the transition overhead, which consists of the cycles needed to invert the direction of the data channel, is similar to the cost of the transfers. It is very difficult, if not impossible, to guarantee net transfer cycles (excluding overhead cycles). Instead, gross transfer cycles (including overhead cycles) rather than net transfer cycles are guaranteed, and overhead cycles are attributed to the request that causes the overhead.

The reservation scheme for memory access cycles assumes a mix of low-latency traffic and high-bandwidth traffic and tries to minimize the average latency for the low-latency traffic while meeting the bandwidth requirements for the high-bandwidth traffic. Typically, cache engines generate low-latency traffic, whereas hardware accelerators generate high-bandwidth traffic.

**Resource users**

On behalf of the tasks they execute, hardware blocks and cache engines issue memory requests that consume memory access bandwidth. The arrival and servicing of memory requests is described by two functions of the number of memory-clock cycles (t): the request function *R* and the supply function *S*. Both functions are taken from (Feng & Mok 2002), and are depicted in *Figure 4-12*.

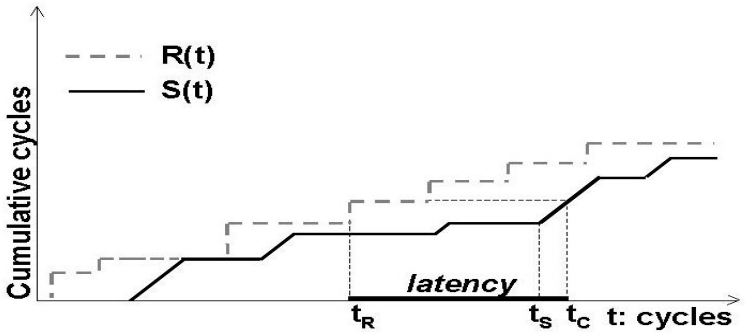


*Figure 4-12.* Request and supply functions.

The request function R(t) represents the total number of cycles *requested* in the interval (0, t), whereas the supply function S(t) represents the total number of cycles *supplied* in the interval (0, t). R(t) is a simple staircase function, for which every step represents the arrival of one or more multi-cycle requests. If the requests arriving at $t_R$ have total gross size $s$($s \geq 0$), then

$$\lim_{t \downarrow t_R} R(t) = R(t_R) + s. \quad (1)$$

In the $S(t)$ function, supply intervals alternate with still intervals. Cycles are supplied in the supply intervals only:

$$S(t + \Delta t) = S(t) + \Delta t, \text{ when } (t, t + \Delta t) \text{ is a supply interval;} \quad (2)$$

$$S(t + \Delta t) = S(t), \quad \text{when } (t, t + \Delta t) \text{ is a still interval.} \quad (3)$$

The number of supplied cycles can never be larger than the number of requested cycles. A request is characterized by size $s$, arrival time $t_R$, start time $t_S$, completion time $t_C$, and latency $\lambda$. From $t_R$ to $t_S$, the request is pending; from $t_S$ to $t_C$, the request is being serviced. ($t_S$, $t_C$) is the service interval for the request.

$$S(t) \leq R(t). \tag{4}$$

$$t_S = max\{t \mid S(t) = R(t_R)\}. \tag{5}$$

$$t_C = min\{t \mid S(t) = R(t_R) + s\}. \tag{6}$$

$$\lambda = t_C - t_R. \tag{7}$$

Different requesters will be identified by an index to the request and supply functions. For requester $i$, the request and supply functions are denoted $R_i(t)$ and $S_i(t)$. Since every cycle can be supplied at most once, different requesters have disjoint supply intervals:

$$S_i(t + \Delta t) = S_i(t) + \Delta t \;\Rightarrow\; S_i(t + \Delta t) = S_j(t)\ \forall\, j \neq i. \tag{8}$$

Typically, there are two different types of hardware blocks, with different request characteristics and different service requirements. *High-bandwidth requests*, typically issued by hardware accelerators, tend to have a regular request pattern, and require effective use of memory bandwidth. In media systems, these requests represent the bulk of the traffic. High-bandwidth traffic generally has latency requirements that are individually, but not tightly, bounded. *Low-latency requests* have an irregular request pattern with potentially large bursts, and require low average latencies. Individual request do not have bounded latency requirements. Low-latency bursts can be accommodated because of the relatively large latency bounds of the regular traffic.

The descriptions in this section are restricted to a single low-latency requester (*LL*) and a single high bandwidth requester (*HB*). This simplification helps to focus on the quintessence of the reservation mechanism. In this area, research is still in progress and details are not yet available for publication.

**Budget definition**

The budget definition for the low-latency budget is given in two steps. In the first step, we make a simplifying assumption: the budget boundaries are assumed to be hard, i.e., out-of-budget cycles are not supplied, even if no other requester is contending for them. With this assumption, the reservation mechanism is non-bandwidth preserving (idle memory cycles while requests are pending), but easy to explain. In the second step, this assumption is dropped.

The low-latency budget (*LL*) can be compared to a credit card, where the customer borrows from the bank, and pays back later. *LL* goes through a sequence of active and inactive intervals. During an active interval, *LL* is either borrowing or paying its debt.

By definition, an active interval starts at $t = 0$. During each active interval, the low-latency budget is defined by two functions $UB_{LL}(t)$ and $LB_{LL}(t)$, the upper and lower bound, respectively. In the first step we assume that these functions bound the supply function $S_{LL}(t)$ directly:

$$LB_{LL}(t) \leq S_{LL}(t) \leq UB_{LL}(t) \quad (9)$$

*Figure 4-13* depicts one active interval of a low-latency budget. The gray band represents the bounds that the budget imposes on $S_{LL}(t)$. At this point in time, *LL* is requesting, and $S_{LL}(t)$ starts its first supply interval after the arrival of the pending request(s). At $t = 0$, $S_{LL}(t)$ starts its first supply interval, and *LL* becomes active. At $t = a$, the upper bound is hit, no more credit is available, and the requester starts paying back. At $t = b$, there is sufficient credit again to resume supplying. At $t = c$, there are no more requests pending, and the requester starts paying back again. At $t = d$, a new burst of requests arrives. At $t = e$, supplying resumes. Finally, at $t = e$, the complete debt has been paid back. If there is no request pending and there is no remaining debt, the requester becomes inactive.

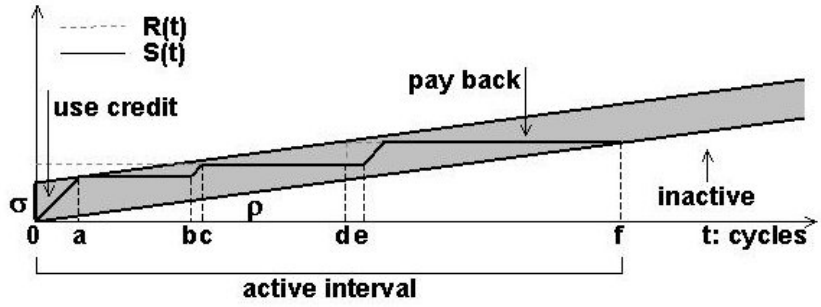


*Figure 4-13.* Low latency contract.

The lower bound corresponds to a function $\rho t$, where $\rho$ is a fraction of the available cycles, with $0 < \rho << 1$. The vertical distance between the two bounds, $\sigma$, determines the burst size accommodated by the budget. The $\sigma$ and ρ parameters are taken from the sigma/rho ($\sigma/\rho$) abstraction, used for

traffic characterization in network calculus (Cruz 1991). With these parameters, the low-latency budget is given by

$$UB_{LL}(t) = \rho t + \sigma, \tag{10}$$

$$LB_{LL}(t) = \rho t. \tag{11}$$

This completes the first step, in which we assumed that upper bound is *hard*. This hard upper bound implies that out-of-budget supply is not allowed, even when *HB* is not requesting. This is a waste of bandwidth, and has a negative impact on the average *LL* latency as well.

When the upper bound is not hard, equation (9) does not necessarily hold. To define soft bounds, some additional terminology is needed. The functions $IBS(t)$, intra-budget supply, and $XBS(t, \Delta t)$, extra-budget supply, are defined by the following equations:

$$S(t+\Delta t) = S(t)+\Delta t \Rightarrow IBS(t+\Delta t) = \min(IBS(t)+\Delta t, UB(t+\Delta t)), \tag{12}$$

$$S(t+\Delta t) = S(t) \Rightarrow IBS(t+\Delta t) = \max(IBS(t), LB(t+\Delta t)), \tag{13}$$

$$IBS(t') = UB(t')\ \forall t' \in (t, t+\Delta t) \Rightarrow XBS(t, t+\Delta t) = (S(t+\Delta t) - S(t)) - (UB(t+\Delta t) - UB(t)), \tag{14}$$

$$IBS(t') < UB(t')\ \forall t' \in (t, t+\Delta t) \Rightarrow XBS(t, t+\Delta t) = 0, \tag{15}$$

$IBS_{LL}(t)$ can take values between 0 and $\sigma_{LL}$. $XBS_{LL}(t_S, t_C) > 0$, extra-budget supply for an *LL* request with service interval $(t_S, t_C)$, is allowed only if *HB* is not requesting at $t_S$.

**Arbitration algorithm**

The arbitration algorithm decides on how to allocate the cycles. It is priority-based, and uses three priorities, two for *LL* (default and limit), and one for *HB*. The *LL* default priority is higher than the *HB* priority; the *LL* limit priority is lower than the *HB* priority. In the CPU domain, this dual priority scheme is known from bandwidth-limiting servers(Burns & Wellings 1993). In the following subsections it becomes clear when these priorities apply.

Arbitration is non-preemptive. Ongoing transfers are completed, even when a higher-priority request arrives. This has to be the case, because

preemption is detrimental to the efficiency of the memory (causes many overhead cycles). In the discussion of the enforcement mechanism, the consequences of the choice are addressed in more detail.

The description of the implementation corresponds to a very elegant solution, conceived by Hans van Antwerpen at Philips Semiconductors, used in the arbiter of a double data rate (DDR) memory controller (de Oliveira & van Antwerpen 2003).

**Accounting**

The accounting mechanism is depicted in Figure 4.12. It uses a saturating counter ACCOUNT, which saturates at 0 and CLIP. ACCOUNT is initially 0, and is increased or decreased every cycle. ACCOUNT keeps track of $IBS_{LL}(t)$. It is updated every cycle. If the cycle is allocated to the requester, ACCOUNT is increased with DEN – NUM, otherwise it is decreased with NUM. NUM stands for Numerator, and DEN stands for Denominator.

$$\text{NUM/DEN} = \rho_{LL}. \tag{16}$$

$$\text{CLIP/NUM} = \sigma_{LL}/(1-\rho_{LL}). \tag{17}$$

$$\text{ACCOUNT/NUM} = IBS_{LL}(t). \tag{18}$$

In the budget definition, NUM, DEN and CLIP replace the original $\rho$ and $\sigma$. One of these values can be freely chosen, the others then follow from (16) and (17). Choosing a round value for NUM, which is somewhat counter intuitive, the CLIP value becomes more intuitive.

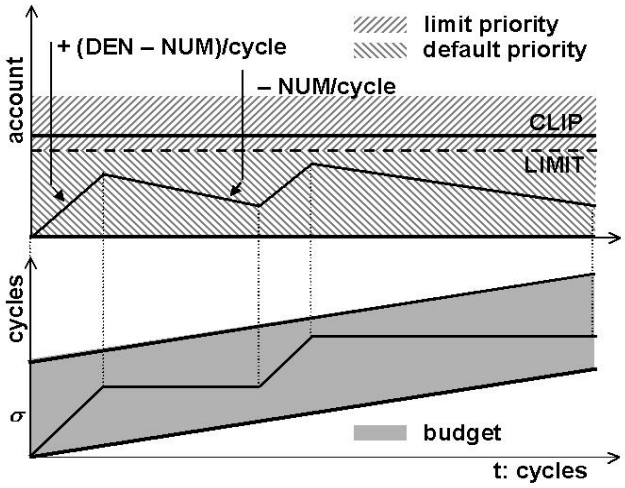


*Figure 4-14.* Priorities, accounting, and enforcement.

**Enforcement**

Enforcement makes sure that the *LL* priorities are switched at the appropriate times. The decision to raise or lower the *LL* priority is based on comparing ACCOUNT with a threshold LIMIT. If ACCOUNT < LIMIT, then *LL* has default priority; otherwise, *LL* has limit priority. If $\max(s_{LL})$ is the maximum gross *LL* request size, then

$$\text{LIMIT} = \text{CLIP} - \max(s_{LL})*\text{NUM}. \quad (19)$$

The threshold value LIMIT must be such that the boundary constraint of the *LL* budget is satisfied. Extra-budget supply, $XBS_{LL}(t_S, t_C) > 0$, requires that $IBS_{LL}(t_S) > \sigma_{LL} - s$, where $s$ is the size of the request. Because of (16) through (19), this implies ACCOUNT > LIMIT at $t_S$, which in turn implies that *LL* has limit priority at $t_S$. If *LL* has limit priority, the request can only be serviced if *HB* is not requesting. Hence, extra-budget supply is only possible if *HB* is not requesting, which was the desired effect.

For implementation simplicity, LIMIT is currently implemented as a programmable parameter. In order to minimize the number of stall cycles, the DDR controller has a small queue of *LL* requests after arbitration. Hence, in a real implementation, LIMIT/NUM has to be larger than $\max(s_{LL})$, depending on the size of this queue.

**Admission control**

By definition, admission control decides if a certain combination of contracts is feasible. Since there is only one contract, there is no admission control.

## 7. CONCLUSION

A major source of robustness problems in current generation systems is the unpredictable behavior caused by interference among concurrently executing applications that compete for access to shared system resources—such as processor cycles, cache space, and memory access cycles. Traditionally, these aversive effects of interference could be kept under control by deploying a real-time OS in combination with a sufficient degree of over provisioning.

For today's systems, this approach is no longer viable. The price erosion in the consumer electronics market forces chip vendors to integrate more and more functionality in an SoC, at the expense of system robustness. For instance, while previous generation SoCs separated real-time audio/video hardware from general-purpose hardware to handle user events, today's multiprocessor SoCs deploy generic processor and interconnect hardware that handle both.

This chapter outlines an approach to bound interference among independently developed subsystems. The system provides each subsystem with an execution environment—called a virtual platform—that emulates the environment in which the subsystem was developed and tested. A subsystem reserves a share of each required system resource. This set of reservations defines the virtual platform. All shared resources in the virtual platform must provide guaranteed reservations to subsystems, or deny a reservation request when the request exceeds the available capacity. The research challenge towards such compositional systems is threefold.

- Define hooks in hardware and software with associated strategies to provide and guarantee reservations for *every* shared system resource.
- Provide an overall resource management strategy that integrates the individual reservation strategies of each shared resource.
- Define an approach to characterize subsystems in terms of execution requirements that can be translated into the desired resource reservations.

This chapter takes on the first challenge and presents reservation mechanisms for the key resources in a multiprocessor SoC: processor cycles of a CPU, cache space in an L2 cache that is shared among multiple processors, and memory cycles arbitrated by a DDR memory controller. The

described DDR controller is currently deployed in Philips Nexperia solutions, while the processor reservations are proposed for integration in embedded operating systems, such as CE Linux. The presented cache space reservations are targeted for inclusion in the next generation Philips Nexperia SoCs.

## 8. ACKNOWLEDGEMENT

The authors want to express their gratitude to Peter van der Stok, and Kees Goosens for their review comments.

## REFERENCES

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., & Warfield, A. 2003, "Xen and the art of virtualization", *Proceedings of the nineteenth ACM symposium on Operating systems principles* pp. 164-177.

Baruah, S. & Lipari, G. "A multiprocessor implementation of the total bandwidth server", in *Proceedings 18th International Parallel and Distributed Processing Symposium*, pp. 40-49.

Burns, A. & Wellings, A. J. 1993, "Dual-priority Assignment: A practical method for increasing processor utilization", in *Proceedings of 5th Euromicro Workshop on Real-Time Systems*, Oulu, Finland, pp. 48-55.

Cruz, R. L. 1991, "A Calculus for network delay, part I: network elements in isolation", *IEEE Transactions an Information Theory*, vol. 37, no. 1, pp. 114-131.

Cumming, P. 2003, "The TI OMAP™ Platform Approach to SoC," in *Winning the SoC Revolution*, G. Martin & H. Chang, eds., Kluwer Academic, pp. 97-118.

de Oliveira, J. A. & van Antwerpen, H. 2003, "The Philips Nexperia™ Digital Video Platform," in *Winning the SoC Revolution*, G. Martin & H. Chang, eds., Kluwer Academic, pp. 67-96.

Eide, E., Stack, T., Regehr, J., & Lepreau, J. 2004, "Dynamic CPU management for real-time, middleware-based systems", in *Proceedings 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* , pp. 286-295.

Feng, X. & Mok, A. 2002, "A Model of Hierarchical Real-Time Virtual Resources", in *Proceedings IEEE Real Time System Symposium*, Austin, USA, pp. 26-35.

Goossens, K. & González Pestana, S. 2004, "Communication-Centric Design for Real-Time Consumer-Electronics Systems on Chip," in *Dynamic and robust streaming between connected CE-devices*, P. van der Stok, ed..

Joseph, M. & Pandya, P. 1986, "Finding response times in a real-time system", *British Computer Society Computer Journal*, vol. 29, no. 5, pp. 390-395.

Klein, H. 1993, *A Practitioner's Handbook for Real-Time Analysis* Kluwer Academic Publishers.

Liedtke, J., Haertig, H., & Hohmuth, M. 1997, "OS-Controlled Cache Predictability for Real-Time Systems", in *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS)*, IEEE Computer Society, pp. 213-227.

Lipari, G. & Bini, E. 2003, "Resource partitioning among real-time applications", in *Proceedings 15th Euromicro Conference on Real-Time Systems* , pp. 151-158.

Liu, C. & Layland, J. 1973, "Scheduling algorithms for multiprogramming in a hard real-time environment", *Journal of the ACM*, vol. 20, no. 1, pp. 46-61.

Molnos, A., Heijligers, M. J. M., Cotofana, S. D., & van Eijndhoven, J. 2005, "Compositional memory systems for multimedia communicating tasks", in *Proceedings of Design Automation and Test in Europe (DATE)*, Munich, Germany.

Nolte, T. & Kwei-Jay, L. 2002, "Distributed real-time system design using CBS-based end-to-end scheduling", in *Proceedings. Ninth International Conference on Parallel and Distributed Systems*, pp. 355-360.

Oikawa, S. & Rajkumar, R. 1998, "Linux/RK: A Portable Resource Kernel in Linux", in *Proceedings IEEE Real-Time Systems Symposium Work-In-Progress*.

Otero Pérez, C. M., Steffens, E., Loo, G. v., Stok, P. v. d., Bril, R., Alonso, A., Garcia Valls, M., & Ruiz, J. 2003, "QoS-based resource management for ambient intelligence," in *Ambient Intelligence: Impact on Embedded System Design*, T. Basten, M. Geilen, & H. de Groot, eds., Kluwer Academic Publishers, pp. 159-182.

Paulin, P., Pilkington, C., & Bensoudane, E. 2002, "StepNP: A System-Level Exploration Platform for Network Processors", *IEEE Design & Test of Computers*, vol. 19, no. 6, pp. 17-26.

Rajkumar, R., Juwa, K., Molano, A., & Oikawa, S. 2001, "Resource kernels: A resource-centric approach to real-time and multimedia system," in *Readings in multimedia computing and networking*, Morgan Kaufmann Publishers Inc., pp. 476-490.

Ravi, I. 2004, "CQoS: a framework for enabling QoS in shared caches of CMP platforms", in *Proceedings of the 18th annual international conference on Supercomputing*, ACM Press, pp. 257-266.

Seawright, L. & MacKinnon, R. 1979, "VM/370 -- a study of multiplicity and usefulness", *IBM Systems Journal*, vol. 18, no. 1, pp. 4-17.

Stravers, P. & Hoogerbrugge, J. 2001, "Homogeneous multiprocessing and the future of silicon design paradigms", *Proceedings of the International Symposium on VLSI Technology, Systems, and Applications(VLSI-TSA)*.

van Eijndhoven, J., Hoogerbrugge, J., Nageswaran, J., Stravers, P., & Terechko, A. 2005, "Cache-Coherent Heterogeneous Multiprocessing as Basis for Streaming Applications," in *Dynamic and robust streaming between connected consumer electronic devices*, P. van der Stok, ed..