Grant Agreement number: 288574

Project acronym: **vIrtical**

Project title: SW/HW extensions for virtualized heterogeneous multicore platforms

Seventh Framework Programme

Funding Scheme: Collaborative project

FP7 -ICT -2011-7

Objective ICT-2011.3.4 Computing Systems

Start date of project: 15/07/2011 Duration: 36 months

D 3.4 *Cost-efficient primitives, extensions and protocols for system-level monitoring of NoC-based multicore SoCs with feasibility study based on KVM*

Due date of deliverable: July 2012

Actual submission date: August 2012

Organization name of lead beneficiary: **UPV**
Contributors for this deliverable **TEI**, VOSYS, STM
Work package contributing to the Deliverable: WP3

Dissemination Level		
Note: After joint discussions, TEI (task leader) and VOSYS and ST-F (contributors), acting through the project coordinator (UPV), proposed to re-characterize the dissemination level of D3.4 as "public for wide dissemination" (PU) from "confidential, only for members of the consortium, including the Commission Services" (PU). The motivation behind this proposed change was that the material contained in the report is quite generic and the supporting code is open source. Email response by the Project Officer has been affirmative, with a remark **not** to make an amendment now, but later on and only if there are any significant points to change during project execution. This change is also reported within the WP3 section of the periodic management report (M12).		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

APPROVED BY:

Partners	**Date**
All	**28/08/2012**

Table of Contents

Abstract

Within the ICT vIrtical project - workpackage WP3, the current deliverable D3.4 examines joint research and development efforts by TEI, VOSYS and ST-F towards innovative system-level monitoring techniques and virtualization of embedded SoC and NoC-based multicore SoCs, including design primitives, protocols and supporting system libraries. The monitoring information collected from cost-efficient, non-intrusive design primitives, protocols and system libraries can be exploited at different system levels by distributed hardware controllers in order to dynamically optimize operating system, hypervisor and application performance, scalability, power consumption, reliability and overall quality of experience with a small relative cost during virtualization.

In this context, VOSYS has evaluated system wide profiling solutions of full hardware virtualization using the Linux Kernel Virtual Machine (KVM) on top of the ARM Cortex-A15 Fast Models virtual platform. While the limited accuracy of Fast Models has deprived the opportunity to quantitatively evaluate the KVM component, VOSYS has defined a generic methodology for system wide profiling of virtualized systems based on the use of both hardware and software counters, detailing its innovative aspects and identifying complexity issues. This methodology could be applied on the ARM Cortex-A15 hardware platform, when it becomes available.

In order to illustrate the effect of multicore SoC monitoring at operating system and hypervisor level, TEI and ST-F have implemented a SystemC-based clock-accurate, transaction-level virtual platform (VP) prototype equipped with a unified, generic and flexible time-driven and event-based monitoring API. Using this VP, a first case study optimizes the performance of shared memory-based array processing (parallel matrix multiplication) via a distributed hypervisor whose threads execute on several CPUs and coordinate with the application threads to perform load balancing and memory utilization sharing requirements. A second study improves the distribution of relative NoC power dissipation of the MPEG4 application model using dynamic frequency scaling with or without real-time constraints.

Glossary

DFS - Dynamic Frequency Scaling

DMA - Direct Memory Access

DPM - Dynamic Power Management

KVM - Linux Kernel Virtual Machine

NoC - Network-on-Chip

SoC - System-on-Chip

SLA - Service Level Agreement

VM - Virtual Machine

VMM - Virtual Machine Monitor

VP - Virtual Platform

1. Introduction

This deliverable highlights first year progress and collaboration activities on hypervisor design for virtualization within workpackage 3 of the **vIrtical** ICT project.

More specifically, Sections 2 and 3 of this report focus on virtualization, including open source solutions (KVM, QEMU) and innovative Linux-based profiling and monitoring strategies, such as Oprofile, PAPI and perf. In an experimental study in Section 3.4, VOSYS has evaluated system wide profiling implementations of full hardware virtualization using the Linux Kernel Virtual Machine (KVM) on top of the ARM Cortex-A15 Fast Models virtual platform prototype. Although the limited accuracy of Fast Models has deprived the opportunity to derive quantitative data leading to the evaluation of the KVM component, VOSYS has proposed a generic methodology for system wide profiling on virtualized systems based on hardware and also software counters, detailing its innovative aspects and identifying related complexity issues. This generic methodology outlined in Section 3.5 would be applied for the evaluation of KVM profiling on the actual ARM Cortex-A15 hardware platform, when it becomes available. In addition, another alternative method for monitoring KVM micro-code, pursued in a different context by TEI within this deliverable, is based on cross-compiling and applying reference delays provided for each assembly instruction in the architecture technical reference manual; this info is currently available for ARM Cortex-A7 and Cortex-A9.

In Section 4, the report outlines specifications of a system-level multicore system-on-chip (SoC) virtual platform (VP) prototype architecture developed jointly by TEI and ST-F, including connectivity, functionality and interaction among major components. The proposed SystemC VP connects together via a configurable hypercube-based network-on-chip (NoC) model based on parameterizable high performance, low latency inter-tile routers: processor tiles (seen as abstract processor models) attached through special network interfaces, external shared memory modules attached through memory controllers, I/O Memory Management Units (IOMMUs) and DMA controllers. Moreover, IOMMUs, DMA controllers, and DMA-capable external devices (integrating a DMA controller) exchange data and control information, such as DMA source and destination address, transaction length and data, through one or more SoC buses, such as AMBA AXI. The VP is also equipped with a unified, generic and flexible time-driven and event-based monitoring API, available at both user- (via user-defined extensions of C++ base classes) and system-level (via appropriate enable mechanisms on specific system classes).

In Section 5, TEI considers two interesting case studies related to virtualization and monitoring solutions on the SystemC-based clock-accurate, transaction-level VP; notice that the VP is expected to be extended and released as an open source software package in 2013. The first study examines performance optimization of virtualized shared memory-based array processing through a distributed hypervisor model that performs best-effort memory bandwidth sharing (e.g. arising from a service-level agreement) and application load balancing. The second one outlines the design of distributed dynamic power management (DPM) module for dynamic frequency scaling (DFS) and examines relative dynamic NoC power savings on a hypercube NoC-based multicore SoC architecture. In this case, the VP is stimulated through an MPEG4 traffic speed test communication pattern.

Finally, Section 6 provides conclusions and promising extensions towards hypervisor-based dynamic system management of NoC-based multicore SoCs supporting high performance, power-efficient and reliable services. We conclude this report with an extensive list of references and bibliography.

2. Virtualization

In the last decade virtualization has been established as a very powerful tool, expanding the capabilities of servers and enabling disruptive technologies, such as cloud computing. At the same time, virtualization has also been proven as a powerful tool for end users, system administrators, security researchers, and system developers. Virtualization has only started to show its capabilities on mobile and embedded platforms, however not unlike the desktop and server world, a very wide range of new use cases can be supported.

Virtualization is a technique where an abstraction of the physical hardware is created in order to run applications and operating systems while hiding the details of the hardware used. The software that manages this abstraction is often called a Hypervisor or a Virtual Machine Monitor (VMM), and the abstractions created are called Virtual Machines (VM). Using a Hypervisor, one can run multiple operating systems on the same machine, at the same time. Each operating system is run under its own Virtual Machine and accesses physical hardware which is abstracted with the help of the Hypervisor [33].

2.1. Hypervisor Software

It is common to classify virtual machines as Native Virtual Machines (Type I) and Hosted Virtual Machines (Type II). In the former case the Hypervisor is run directly on the hardware and can load different virtual machines side to side. In the latter case however, the Hypervisor is run as an application under an existing operating system, which is called Host. Virtual machines are run alongside the regular processes of the host and are called guests.

Not all Hypervisors are alike, since there is more than one way to do virtualization. For example a Virtual Machine may be designed to run software not intended for the hardware architecture used, as is the case with various emulators or with the Java Virtual Machine. However we are mostly interested in Virtual Machines that can run the same software as the hardware architecture, unchanged or with minimal changes. Hypervisors that can run complete operating systems intended for the underlying hardware, with no changes to the code, are said to implement Full Virtualization; the software running under the VM is under the illusion it runs under real hardware. At the same time, there are Hypervisors that require the cooperation of the Operating System running under the VM; in this case the operating system needs to be patched to run under a virtual architecture slightly different than the real hardware. This kind of virtualization is called Paravirtualization.

One of the most significant barriers to efficiently virtualizing an architecture is the presence of instructions that are sensitive to the current mode of operation of the processor. Typically an operating system running under a VM executes in a lower privilege mode than what it is designed for, and attempts to use instructions that control the state of the hardware. If these do not cause a trap to the Hypervisor, but instead fail silently, or just behave differently in the lower privilege mode, then the Hypervisor would have to implement complicated binary patching techniques to intercept those instructions. Other challenges to efficiently virtualizing an architecture include the way memory management and virtual memory is implemented, which mean that often Hypervisors have to maintain Shadow Page Tables incurring additional overheads.

In order to overcome these performance overheads, one solution is to implement paravirtualization instead of full virtualization. However running unmodified guest operating systems is desirable, so hardware vendors have started shipping extensions to their processors so they are efficiently virtualizable. In that case, when a Hypervisor may take advantage of the hardware support for efficient virtualization, the system is said to support Hardware Virtualization.

2.2. Open Source Virtualization Solutions

Besides KVM, Linux Container is an operating system virtualization implementation, and uses namespace isolation features inside the Linux kernel in order to create isolated domains, where applications can execute without breaking free from the container [16]. The kernel functionality needed by LXC [18] is already part of the Linux kernel, so no special patches are necessary; this includes e.g. namespace isolation for Process IDs (PIDs), the file system, network

interfaces, ensuring containers are under the illusion they are running alone on the system, while at the same time keeping each container securely isolated from each other. This kind of virtualization allows for very good performance, since everything is done by the Linux kernel without excessive context switches, however all containers run under the same Kernel and there is a larger attack surface in case of a malicious user inside a container.

2.3. Monitoring Tools in Linux-based Multicore Processors

There is a variety of tools under Linux based systems which can use a Performance Monitor Unit of platform architecture. These vary from dynamic monitoring application APIs, such as PAPI [20, 26], to performance profiling oriented tools for system developers, such as OProfile [24]. OProfile supports hw events, multithreading and monitors all processes (e.g. kernel, libc), but unlike gprof, it requires kernel support and can't handle call graphs or cummulative timings. Other open profilers include DTrace, SystemTap and Lttng.

Software profiling is a common technique used to dynamically study the behavior of a program in terms of frequency of function calls, or the cost of instructions with regards to processor time consumed or other hardware metrics, such as TLB misses. Different types of profilers exist, depending on the method used to gather data. For example, instrumentation profilers depend on special instructions inserted by the programmer or the compiler to collect data, or by running the code under the control of the profiler.

A common methodology of collecting data for profiling is by sampling in intervals determined by hardware events, like a real time clock or performance counters. In this case, the clock or the counter is configured to cause an interrupt when an overflow occurs, at which point the profiler "takes a sample" by recording the state of the program (e.g. the last executed instruction). With a high enough amount of samples, we can get a very good approximation of which parts of a program are more expensive with respect to time spent (if a clock is used), or more sensitive to hardware events measurable by performance counters made available by a Performance Monitoring Unit.

Sample based profilers can usually get measurements for an executed program by having a negligible effect on the performance to be measured, since most of the time no extra code is being executed. A reasonable sampling period allows the program to execute as usually up to the point an interrupt occurs in order to take a sample. Some inaccuracies in the data measured can still occur, since the interrupts can be delayed sometimes, or there can be sections where interrupts are disabled. Throughout this deliverable, we consider sample-based profilers, one of the most important use cases of performance monitoring hardware available on existing platforms.

Another important point of classification between profilers (which also refers to the virtualization point of view) is whether a given profiler is system wide. A lot of profiling tools only gather data for a given process, but not for the kernel. In contrast, a system wide profiler collects data from all system processes, as well as from the underlying kernel.

In addition to profiling there are numerous monitoring tools.

Perf is a performance monitoring tool implemented on top of perf events [10], and is also able to do performance counter monitoring, as well as sample based profiling. The types of counters we can track with perf are not limited to hardware counters implemented by the Performance Monitoring Unit (PMU), but we can also count trace events declared within the Linux kernel, as discussed earlier. Thus, context switches, page faults or even KVM specific trace events, can be tracked by using hardware performance counters to monitor and profile code [8].

The Performance API (**PAPI**) is a popular performance monitoring tool built on top of perfmon, perfctr or the new perf_events monitoring interface [20, 26]. PAPI functions as a high level C interface for applications to leverage monitoring capabilities present in the underlying hardware and software stack. Through PAPI, applications can track a number of software and hardware counters, ACPI thermal sensors or even Myrinet network counters while running multicore applications, dynamically tweaking execution parameters to adjust to application-specific performance, power consumption or other requirements. PAPI supports several architectures: AMD, Intel, Cray, IBM BG/POWER, MIPS, Alpha, SPARC, SH. ARM Cortex A8/A9 (A15 on the

way) and provides extensions: a) PAPI_M for Multicore, b) PAPI_G for GPUs (enhancing performance info & presentation), and c) PAPI-V for Virtualization (hypervisor info support).

Since PAPI, is a high level library, it has powered the development of numerous open advanced monitoring and visualization tools extending its capabilities, such as HPCToolkit/HPCView [12], VProf [37], MUMMI [21], PerfSuite [28], PPW [30], Scalea [31] and TAU [34]; in fact, there are so many open tools (also Paradyn, Pablo, Titanium, Kojak, IPM, Scalasca, OpenSpeedShop etc), that it is difficult to select the most powerful, reliable, easily extendable and learnable tool.

3. KVM Hypervisor

The Linux Kernel Virtual Machine (Linux KVM) is one of the most successful and powerful Virtualization solutions available, enabling the Linux kernel to boot guest Operating Systems under a process. Linux KVM has been designed to be portable, and has proven itself in a number of architectures, like Intel VT-x, AMD SVM, PowerPC and IA64, and of course also on the ARM Cortex-A15 platform.

KVM works by exposing a simple interface to user space, through which a regular process can request to be turned into a virtual machine. Usually QEMU is used on the user space side to emulate I/O devices, with KVM handling virtual CPUs and memory management.

3.1. KVM

The Linux Kernel Virtual Machine (KVM) is an established system virtualization solution, implemented as a driver running within Linux, which effectively turns the Linux kernel into an hypervisor. This approach takes advantage of the existing mechanisms within the Linux kernel, such as the scheduler, and memory management. This results in the KVM code base to be very small compared to other hypervisors; this has allowed KVM to evolve with an impressive pace and become one of the most well regarded and feature full virtualization solutions [15].

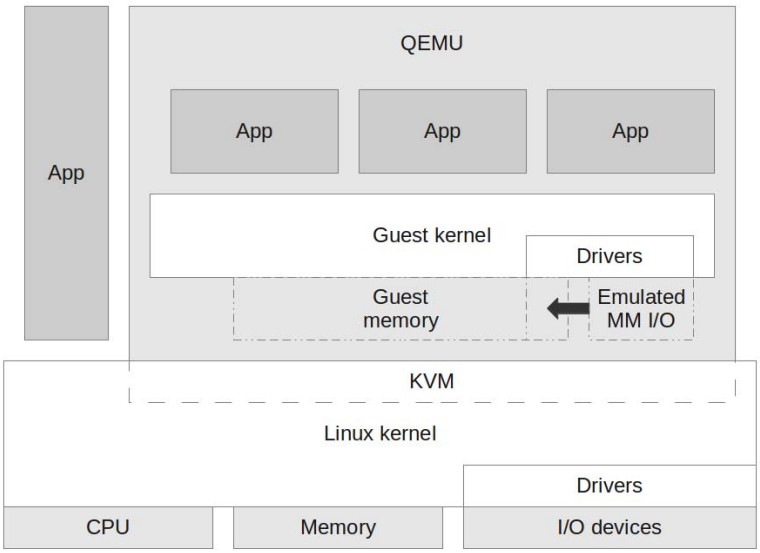

Figure 1. Virtualization using KVM and QEMU.

KVM is designed with a simple architecture in mind (see Figure 1), leveraging existing Linux infrastructure, including process scheduling, memory management, thread and process creation. This is done by exposing an `ioctl` interface towards user space which allows a user space application to enable virtualization functionality, turning the Linux kernel itself into a Hypervisor. Through this interface, regular Linux processes are turned into virtual machines, with threads acting as virtual CPUs. KVM handles switching the context of the processor when the process of a virtual machine gets scheduled by Linux, using the hardware virtualization support present in the hardware in order to virtualize the processor and the memory. To virtualize I/O devices, such as network interfaces and storage, an interface to user space exists, so these can be emulated by the application setting up the virtual machine (usually the QEMU emulator).

The KVM port on the ARM Cortex-A15 has been under development since 2011, and is already in a level of maturity where it is able to boot unmodified guests compiled for the Cortex-A9 or Cortex-A15 processors. The port utilizes the hardware virtualization extensions present in the ARM Cortex-A15 in order to efficiently switch between guests [36].

Since virtualization on ARM did not have to go through the various stages of poor hardware virtualization support that x86 had to go through, the port is much simpler than the KVM on x86 code. For example, second stage memory translation is assumed to be always present removing the need for complicated shadow page table techniques.

3.2. QEMU

QEMU functions as a caller from user space for KVM, e.g. setting up the memory of the VM to be launched, the virtual CPUs to be used, etcetera. QEMU (with the help from KVM) configures memory regions that would trap when the guest attempts to read or write to them; the execution workflow will return to QEMU, which emulates the behavior of memory mapped I/O devices (MMI/O), such as network interfaces, graphics controllers, and storage and user interface devices, such as keyboards. Depending on the underlying architecture QEMU may also handle injecting interrupts and emulating an interrupt controller in the same fashion.

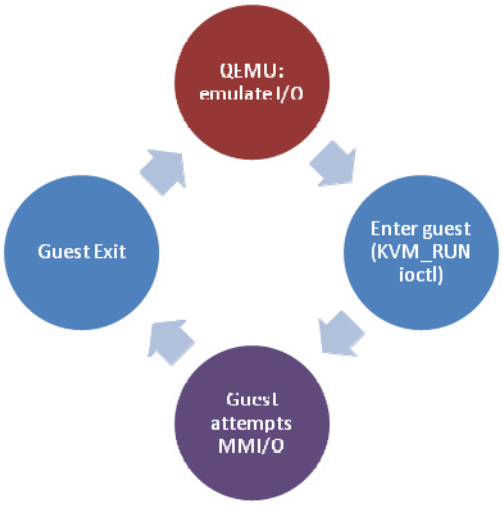

Figure 2. Abstract view of KVM and QEMU interactions.

As shown in Figure 2, synergy between QEMU and KVM is based on a standard `ioctl` system call interface which KVM exposes to user space; QEMU simply issues `ioctl` commands to setup KVM, and to enter execution inside the guest. In the case of a guest exit, QEMU is able to determine why the guest stopped executing and take appropriate action, e.g. by emulating a MM I/O operation.

3.3. Profiling Virtualized Systems: Definitions

The topic of PMU virtualization is in no way a new one. For example, an open source framework called Perfctr-Xen uses performance counter virtualization based on the older perfctr framework [24]. There is also interesting research on utilizing Oprofile within both Xen [19] and KVM-based environments [42] which leads to three different types of profiling depending on the scope of the collected samples.

Host wide profiling refers to running the profiler in the host and collecting samples only for the host. There is a limit to the kind of information we can extract from an instance of host wide profiling; in particular. most of the execution time appears to be spent in the function of KVM which calls a guest's virtualized CPU switching code, with no visibility of what happens inside the guest. This describes what is already possible without the development of additional code in the hypervisor.

Guest wide profiling refers to collecting samples only from a particular guest. To achieve that, the performance counters need to be virtualized or emulated by the host, while at the same time interrupts from performance counters overflows need to be injected synchronously into the guest. In this way the guest can run an unmodified instance of the profiler, to profile its own

processes as if it was running on own hardware. Cloud servers where the client doesn't have permission to run code in the host have been cited as an example of where this functionality can be useful. However having no visibility of the host processes and kernel within the samples collected, is an obvious disadvantage and limits the kind of conclusions that can be reached.

System wide profiling, in the end, is possible when we are running an instance of the profiler on both the host and the guest being profiled. The performance counters are being virtualized as with guest wide profiling, however in the end the data collected from the host and the guest are being merged giving a complete system wide picture of the profiled code behavior on both the host and guest level. An interface, perhaps using special hypercalls[1], which allow a guest to invoke functionality from the hypervisor, is necessary to synchronize the collection of data between the profilers, with consistent parameters (i.e. all profilers should sample the same type of counters). In the end of the data collection period, the profiler in the host needs to gather all collected samples from each guest and combine them into one system wide report.

For the case of system wide profiling, two techniques of collecting guest data have been identified; **full delegation** is the simplest approach, where interrupts from the performance counters are injected to the guest when the overflow occurred while running a guest. In this case, the guest profiler is in charge of collecting and interpreting the sample. In contrast, **interpretation delegation** allows the host to collect the sample data, including the current program counter and the process identifier currently run by the guest. For this to work, an increased degree of communication is required between host and guest profilers, so that the host profiler sends the guest profiler process the necessary sample data to be examined. Since interpretation delegation is more complicated and requires an efficient communication channel between the host and guest profiler, it is only used when full delegation is not possible; i.e. when synchronous interrupt injection is not possible in the guest.

Besides the requirements stated previously, another important question for both guest-wide and system-wide profiling is: what is the right time to save and restore PMU registers?

For guest-wide and system-wide profiling, selecting the right time to switch (save and restore) the PMU state can be based on two common strategies: **CPU switch** and **domain switch [43]**.

- The former technique saves and restores the PMU registers, when the CPU switches between running the guest code (e.g. accounted for interrupts) and running hypervisor code. In this case, when the CPU switches to execute the VMM code that emulates the effect of guest I/O operations, monitored hardware events effectively contributed by the guest are accounted to the host. For guest-wide profiling, the PMU can be turned off, while for system-wide profiling the events are accounted to the VMM.
- The latter method saves and restores the relevant PMU registers when the hypervisor switches execution from one guest to another, thus providing a more realistic picture of the virtual environment.

Perf also includes a feature to profile a guest running under KVM [42], which demonstrates a third option with regard to sample delegation. This allows us to record samples from a running guest, without running another profiler inside the guest; in this case we have **no delegation**, and the sample is both collected and interpreted by the host's instance of perf. However, perf has now knowledge of the memory mappings inside the guest, so for this to work it takes as an extra parameter the kernel map of the running guest. The resulting samples only include information from the guest kernel, and therefore this approach can be classified as neither guest wide nor system wide profiling.

3.4. Preliminary KVM Profiling on ARM Cortex-A15 using Fast Models

The target architecture we are considering is primarily the ARM Cortex-A15 processor, and hardware platforms implemented around it. This processor is based on a version of the ARM architecture which includes support for hardware virtualization; this is what allows KVM to run on this board efficiently.

[1] A hypercall is a special interface that allows the software running under a virtual machine to communicate with the host. This enables features such as paravirtualization of drivers, by batching operations and passing them to the host through hypercalls.

The latest version of the ARM architecture, which is implemented by the Cortex-A15, has been available since mid-2011. This version of the architecture includes numerous improvements, for example the Large Physical Address Extension allowing up to 1TB of physical memory space to be accessible [3], and support for up to two clusters per chip with up to four cores each. Another interesting point of the architecture is that it is implemented by both the relatively powerful ARM Cortex-A15, and also the low power Cortex-A7; a system can include clusters with either cores, offloading the workload according to changing power and performance requirements [11]. This flexibility expands the potential markets and applications the ARM processors can support.

A limit of the ARM architecture traditionally has been that it is not considered virtualizable efficiently, as it does not satisfy the requirements usually defined in the bibliography [29]. The main obstacle being, that a number of instructions behave differently when in user mode rather than in privileged mode; for the platform to be realistically virtualizable without significant slowdowns incurred by binary patching techniques, these instructions should trap. This is facilitated by a number of new features in the ARM architecture which enable efficient software virtualization through hardware acceleration, usually referred to as the ARM Virtualization Extensions [4]. These extensions include virtual-to-physical address translation, protection, partitioning and resource management of complex systems involving client and server devices and software stacks into virtual machines.

A new processor mode is introduced, called Hypervisor mode, which allows each guest to have access to its own privileged mode; the processor state can be switched between guests, allowing the processor to be virtualized without expensive binary patching techniques, and with very few traps being necessary. Virtualization Extensions also allow certain instructions to be set up to trap if that is necessary to run a guest efficiently.

In the ARM architecture, when a CPU receives an interrupt, it checks with the interrupt controller, called the Generic Interrupt Controller (GIC), which was the cause of the interrupt. The GIC functions like just another memory mapped I/O device, and is emulated the same way with the help of QEMU. This complicates implementation of asynchronous interrupt delivery, since we have to handle it within the interrupt handlers that catch the event and KVM itself which must also inject the interrupt to the guest. But also QEMU must be updated in respect to the virtual interrupt to inject. This is possible, but not ideal and adds additional context switches towards user space, where QEMU resides.

Recent versions of the ARM GIC also support exposing a Virtual GIC (VGIC) interface; this is implemented inside KVM as an in kernel interrupt controller, bypassing extra exits towards QEMU, making interrupt delivery simpler and faster.

The above described extensions are sufficient to fully virtualize the CPU and memory for any number of guests, and also trap any accesses to I/O devices so they can be emulated by software.

End user products using the A15 processor have not yet hit the market, and are not expected before the end of 2012. Also silicon implementations of the board are still very rare and not easily accessible; for this reason, the development of the KVM on ARM Cortex-A15 project has relied on the Fast Models simulation package provided by ARM. As explained below, this significantly limits our efforts towards evaluation of the KVM component based on system wide profiling using Oprofile and hardware performance counters.

The simulator processes ARM instructions in sets called quantums, and makes no effort to maintain timing accuracy of the execution of those instructions.

- The simulator processes ARM instructions in sets called quantums, and makes no effort to maintain timing accuracy of the execution of those instructions. The simulator is in no way advertised by ARM to be cycle-accurate with regards to application performance; only the programmer view of the hardware is considered accurate, and monitoring code executed on Fast Models would not produce any useful data [2].
- The way Fast Models processes instructions greatly limits the granularity of any instruction counters, since those are incremented in quantums. The simulator also makes very few guarantees with regards to the execution order of the instructions within a quantum. While the programmer view of the system is still accurate (ensuring data dependencies in the instruction flow), system behavior in terms of branch prediction, caches and TLB performance does not necessarily match the real hardware.

- Finally, a Performance Monitoring Unit is only partially implemented on the Cortex-A15 models included in Fast Models.

Another alternative could use cycle-accurate ARM Cortex-A15 models, e.g. the recent ones available from Carbon Design Systems Performance Analysis Kit. Although this method could allow the development of performance monitoring features, it has a clear shortcoming: speed; it is impossible to boot Linux with KVM hypervisor, and execute application software on a cycle accurate virtual prototype.

Another option, also pursued in a different context by TEI in Section 5, could examine KVM performance bottlenecks by focusing only on key internal micro-code that handles intensive virtualization tasks, such as system management, resource allocation, dynamic virtual machine switching and network/storage virtualization. This kernel microcode, including any virtualization processor extensions, is first compiled to ARM assembly through an open cross-compiler, such as `arm-none-linux-gnueabi-gcc`. Then, each ARM processor instruction (e.g. add, subtract and multiply, compare and branch, move and shift and local load/store) can be annotated with cycle-accurate timing information using the assembly instruction cycle delays published in the technical reference manual (e.g. for Cortex-A9). As explained in Section 5.1, when exploring different architecture configurations, certain parts of the exported assembly code, such as system library calls and memory accesses, may also be replaced by "equivalent" SystemC function calls, hence co-simulating the original micro-code on a system configured with ARM processors and custom IPs.

In conclusion, the ARM Cortex-A15 enables virtualization, and also PMU counters in a virtualized environment, however PMU specific features cannot be evaluated on the Fast Models simulator. However we present guidelines, based on the ARM Cortex-A15 Performance Monitoring Unit, to support more interesting PMU use cases in a virtualization environment.

3.5. KVM Profiling and Monitoring Methodology on ARM Cortex-A15

Host wide visibility of performance monitor counters is already feasible with existing tools; a system designer can already leverage toolkits, such as PAPI, or perf events directly, to obtain visibility from individual virtual machines and react accordingly, provided that access to the host side is provided.

However, extending the scope of performance counters enables additional capabilities. For example, guest wide visibility of the counters for monitoring and profiling allows a virtual machine to fine tune on the level of individual applications running inside the guest without requiring direct access to the host. This is useful in scenarios, such as cloud computing environments, where a client accessing a virtual machine wishes to obtain monitoring capabilities, without compromising the host.

System wide profiling, allows the same kind of granularity of monitoring activity of individual processes inside virtual machines, within an environment that leverages the integration and secure isolation capabilities of virtualization. Today this is not possible on ARM because the performance counters are not accessible from a guest running under KVM-on-ARM. In fact, the ARM Performance Monitor Unit allows PMU virtualization by switching its state in order to be usable from virtual machines [1]; however this is not currently implemented.

In order to implement guest wide, and additionally system wide profiling and monitoring, we need to switch the PMU registers when transitioning from the host to a guest or vice versa. This is not very different from handling the processor state when switching between guests; we save the state of the guest we exit from, and load the state of the guest we are loading. Moreover, for sample-based profiling, since the Performance Monitoring Unit generates interrupts when counters overflow we must ensure that interrupts are handled appropriately by the target guest.

Currently on KVM-on-ARM, when an interrupt is received on a CPU, the execution exits from the guest and is handled by the host. This means that if we allow a guest to use the performance counters, we must be able to detect an interrupt due to a counter overflow in the host, and inject it back to the guest before the later resumes execution. This can be implemented by utilizing the Virtual GIC support present in the Cortex-A15 platform, which allows for a full delegation profiling approach.

For system wide visibility of performance counters, the total counter activity in both the host and the guests must be aggregated. Although this step appears straightforward (since the order of samples does not carry any particular weight), communication between the host and guest instances of the profiler (e.g. perf) could become an issue, since KVM-on-ARM does not currently implement a standard hypercall interface.

As discussed in Section 3.4, past related work has focused only on system wide profiling using Oprofile based on hardware performance counters. Since perf can use software counters defined inside the Linux kernel, it makes sense to consider **system wide profiling and monitoring based on software counters**. This means guests would be able to monitor a number of hardware counters, as well as software counters provided by the host.

In this particular use case, the host's software counters do not usually cause overflows during execution of a guest; however, we can still implement useful features, for example, if we consider experimenting with different PMU state switching strategies during VM entry and exit (called CPU switch), we can keep the PMU state pointing towards a guest even when we are executing KVM and QEMU code on its behalf (domain switch). This way we could perform a guest wide profiling session, in which we can examine which parts of our guest are causing more strain to the underlying host, e.g. page faults.

Perf can also record the stack trace when taking samples. Combining the stack trace of the host with those of the guests, involves several synchronization challenges among host and guest profilers; a simple full delegation approach won't suffice, if we want the stack trace to span multiple levels of virtualization. However, it is very interesting to examine what kind of insight can be obtained by combining stack traces from the host and the guest side.

To implement guest and system wide profiling based on software counters, it does not suffice to use a mostly unmodified profiler running inside the guests. Instead, the PMU architecture would need to be extended to include virtual counters representing the host's software counters. Thus, if we implement CPU switching of the PMU state, then the counters would only overflow when we are executing the host. Likewise, if we profile based on a guest's software counter, overflow would occur only when inside the guest.

Therefere, we envision system wide visibility of hardware and software counters which extends monitoring options within virtual machines and supports advanced features, such as recording super stack traces and profiling a guest based on a combination of physical and virtual counters, reflecting the strain it causes to host resources.

4. Multicore SoC Virtual Platform and System-Level Monitoring

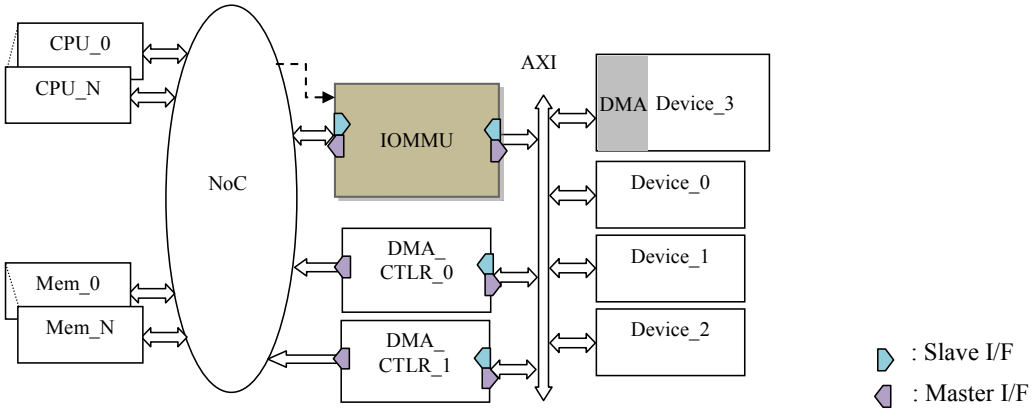

Figure 3. Virtual platform with IOMMU unit managing DMA transactions for different devices

In order to perform concept validation and design space exploration of different multicore applications, in relation to different virtualization requirements and alternative subsystem macro-architectures, such as NoC, Memory Controller and DMA architecture, a SystemC virtual platform has been developed at clock-accurate transaction-level.

Modern scalable, shared-memory, multicore SoC architectures are increasingly common as external devices implement new sophisticated features. As a result the communication mechanism used to pass data between the I/O peripheral devices and on-chip processing cores is becoming very important.

As shown in Figure 3, the proposed multicore SoC virtual platform connects together processor cores, memory tiles (through memory controllers), IOMMU and DMA controllers through a network-on-chip. IOMMU, DMA controller, and DMA-capable external devices exchange data and control information, such as source address, destination address, transaction length and data, through one or more system-on-chip buses, such as AMBA AXI. Notice that DMA-capable devices integrate a DMA controller for direct transmission to the NoC.

While our SystemC virtual platform is still under development (a fully functioning, open source version is expected in 2013), we focus on a high-level description of its architectural components and interfaces.

4.1. VP Component Models

4.1.1. Abstract Processor Model

Our abstract processor model relies on interesting concepts, such as instruction set simulators and cache-integrated processor models. It attempts to capture processor tiles at a high-level of abstraction, i.e. as high-level SystemC threads and supports:

- *fast simulation,*
- *cycle-approximate instruction processing* (cross-compiling feature),
- *a communication interface supporting synchronous blocking remote shared memory operations,*
- *an alternative user-defined network packet command interface supporting synchronous blocking and asynchronous nonblocking remote shared memory operations*,
- *barrier synchronization among arbitrary processor tiles* through a strong synchronous read-modify-write load-linked/store-conditional (LLSC) operation supported by the memory controller (see implementation for centralized lock and barrier for LLSC in Figure 4). An extension to a more powerful software synchronization library is in progress. This library will offer several weak and strong atomic (test & set, fetch & add, fetch & store, compare & swap), as well as distributed synchronization operations, such as lock, built on top of a minimal set of atomic shared memory primitives implemented by the memory controllers.

Alternative read/write barrier operations will also be considered; they are useful for many multicore programming scenarios.

- *efficient packet communication* by interfacing to a high performance NoC; however, notice that remote data exchange is much slower than local access,

Shared Memory Lock via LLSC

```
Lock() {
TryLock:
    lock = LL (lockAddr);
    if (lock) // Lock
              // locked elsewhere
       goto TryLock;
    lock = 1; // Try to acquire
              // lock
    result = SC (lockAddr, lock);
    if (!result) { // Contention
                   // for lock
       ExpBackOff(); // Wait
                     //random
time
       goto TryLock;
    } // now hold the lock
}
```

Shared Memory Barrier via LLSC

```
typedef struct {
    int count;
    int padding[]; // Pad to new
                   // cache line
    int generation;
} barrier_t;

Barrier(barrier_t      *barrier,      int
num_procs) {
   int gen = barrier->generation;
   loop:
        count = LL(&barrier->count);
        count++;
        if (count==num_procs) count=0;
        result=       SC(&barrier->count,
count);
        if  (!result) goto loop;
        if  (count == 0) {
          barrier->generation = gen+1;
          return;
        }
     while (gen == barrier->generation)
              continue;
}
```

Figure 4. Implementation of shared memory lock/unlock and barrier using LLSC.

We have already focused on encapsulating within our SystemC processor model existing processor architectures, such as ARM Cortex A9, and we consider useful software extensions, such as real-time operating system schedulers and KVM-like virtualization models.

4.1.2. Memory and Memory Controller

The default supported physical address space allows memory tiles as large as 1 tera words (2^{40} words); alternative implementations, e.g. 32 or 64-bit addressing mode are also possible. If any operation attempts to access memory beyond the module's range, it fails, the memory is not altered and the acnowledgment is marked with an error flag. Memory is single-port, word-addressable with a standard word size of 64 bits, however, two consecutive words are accessed by the memory controller during DMA operations. Since the data field during an (encapsulated) DMA transaction is 128 bits long, two consecutive read/write accesses are performed by the memory controller. The read/write latency is configurable (default 10 nsec). Memory endianess currently depends on the operating system. Memory initialization can be performed from a user-provided file or based on a single value provided which is copied to all memory locations.

The memory can only be used in conjunction with a compatible memory controller which is attached to each memory tile interfaces to the NoC which converts transport- to/from network-layer packets. The controller interfaces to the NoC (configurable buffer size) and the memory array, i.e. it is made sensitive to a memory ready event to handle memory accesses and also performs atomic operations by supporting the required synchronization data structures, e.g. for LLSC. In case of controller saturation, back pressure is implemented at the router and eventually the processor to avoid dropping packets at the memory controller. The memory controller implements both asynchronous nonblocking and synchronous blocking shared memory accesses through a request acknowledgment flag within the network packet. In the former case, the processor returns immediately after the transmission phase allowing the calling process to perform communication and computation overlap; as an extension, a probe function could be called later to check for completion status. In the latter case, execution is suspended until an acknowledgment is sent back to the caller through the NoC (after a waiting phase).

Depending on the nature of the request, this reply, may be a simple acknowledgment (e.g. in synchronous write and LLSC atomic operations) or it could involve sending a single value (e.g. most atomic operations) or memory data (e.g. in synchronous read).

LLSC operates as follows.

In the *load-linked* part:
`bool cpu_ll_cmd(uint64 addr, data_t *data);`
the value of the target address `data` is returned to the caller and the target address `addr` is marked as a monitored address. If the target address is already in the list, then the operation is unsuccessful. Monitored addresses are flagged when read or written by any node.

In the second *store-conditional* part (used in conjunction with load-linked):
`bool cpu_sc_cmd(uint64 addr, data_t data);`
an address `addr` previously load-linked by anyone is set to a (new) value `data` if and only if it is not flagged. Otherwise, the operation is unsuccessful. When the operation is successful, the address is removed from the list of the monitored addresses.

Furthermore, different memory flags related to specialized QoS, cache, security or fault tolerance operations are also identified by the memory controller for appropriate action. Thus, the memory controller presents the user with a rich, user-friendly multicore programming API based on shared memory paradigm.

4.1.3. Network-on-Chip

The NoC is decomposed into a number of interconnected modular unicast or multicast routers. Due to its symmetry, multiplicity of paths and nice embedding properties, we have implemented a custom hypercube NoC topology;other topologies, such as traditional NoC topologies, e.g. mesh and torus, and chordal point-to-point topologies, such as Spidergon, can also be examined. In order to reduce router complexity, we assume deterministic shortest path routing which avoids expensive buffering schemes and packet reassembly unit and improves performance. We use two types of `nxn` routers. While the *unicast router* is a simple input-output buffered switch, the *helix router* (generalized from an open source router provided by Synopsys) provides internally an array of shift registers to implement multicast; this is very useful for cache protocol design. Internal router architecture is beyond the scope of this document. In order to avoid protocol deadlock, two virtual channels (VCs) with request and reply packets are supported for each standard, priority (e.g. offering QoS by avoiding head-of-line blocking effects) and system monitoring packet class. Three different VCs are used by cache coherence packets (request, reply and acknowledgment).

The NoC packet size is currently 256 bits in order to encapsulate efficiently configuration and control information within a single packet transfer. This choice supports the shared memory communication paradigm, as well as efficient DMA (and possibly distributed direct memory-to-memory DMA) operations. It also provides a relatively small enough packet size to allow for on-chip packet transmission in a single step, while offering a good ratio of header vs. payload size, thereby reducing header overhead.

The structure of the NoC packet contains an optional 64-bit data; this is extended to 128-bits for DMA, i.e. always mapping two 64-bit burst AMBA AXI read operations from a device (assuming that the DMA length is always a multiple of two packets). It also includes 8 bits for addressing 256 possible sources, 8 bits for addressing 256 possible network node destinations, usually 40 bits for a device address space, and a number of different flags (not yet exhaustively specified), such as request/response packet, end-of-packet and status indicators. Finally, the NoC packet contains *system- and virtualization-specific information used for IOMMU monitoring*, synchronization, QoS and monitoring, security/protection and fault tolerance issues.

4.1.4. DMA Controller with Integrated Network Interface

The DMA controller incorporates standard techniques (also used in ARM's A15) to support single remote read/write DMA operations by accessing data from device memory through standard AMBA AXI read/write operations. Each DMA controller is also attached to a custom network interface (NI) that provides access to the NoC. The NI is responsible for packetization (or de-packetization) of data to/from network packets directed to/from memory, as well as size conversions; frequency conversions are currently not needed in our current VP, since our single

port memory uses an asynchronous event-based interface. Other NI characteristics related to fault tolerance, security and power-efficiency can also be specified in the future. The implemented DMA controller uses a flit-by-flit fetch and deposit scheme with small buffers to store data before transmitting to the network via the attached network interface. Recently proposed fly-by schemes are also interesting: they are faster due to direct transmission of data from the device to the NoC but have a synchronization overhead since an array of semaphores is required. Atomic DMA transactions and direct memory-to-memory copy (so called distributed DMAs) are also interesting.

For DMA operations, *configuration and control information* is transferred through the system bus and includes source address (`SRC` - 40 bits), destination address (`DST` - 40 bits) and length in bytes (`DST` - 16 bits). This information is processed and encapsulated within the NoC packet.

4.1.5. Peripheral Devices

Devices attached to the AMBA bus currently include two generic models: a wireless network and a digital camera which integrates a local memory tile for data storage. A future target is to connect different camera flows in order to model a parallel surveillance application as a realistic multicore benchmark performing coherent shared memory accesses and DMA operations. Additional work would consider modeling different peripheral devices, such as PCI express, Ethernet bus and USB.

4.1.6. Amba AXI3 Bus

Since we are mainly interested in general IOMMU functionality rather than performance issues, a SystemC implementation of an AMBA AXI-3 system bus is used for connecting the devices, the DMA controllers and the IOMMUs. An AMBA AXI-4 bus implementation is also useful.

4.1.7. I/O Memory Management Unit (IOMMU) Architecture

Multiple IOMMUs within the virtual platform are able to support a mutually exclusive subset of one or more integrated devices (only one such example is shown in Figure 3). IOMMU functionality, including support for virtualization and hardware/software interfaces, is being developed from scratch. A preliminary description of the functionality of this module is beyond the scope of this work, e.g. see architecture specifications in deliverable D1.2.

4.2. System-Level Monitoring API

System-level monitoring based on fast and accurate evaluation of different system metrics is an essential part in multicore SoC design flow. The collected monitoring information enhances system flexibility during runtime by adapting machine resources to dynamic workloads, thus improving performance, enhancing power-efficiency and providing assurance for early detection and efficient recovery from transient, intermittent or permanent faults or other hazards, e.g. data race, deadlock, starvation or livelock. Thus, high-level system performance modeling is an essential ingredient in SoC design flow.

This section considers the development of a generic system-level statistics library of communication/synchronization facilities and hierarchical protocols to monitor the performance characteristics of NoC-based heterogeneous multicore SoC architectures at the system, hypervisor or application layer. While the user may utilize this statistics library to perform measurements for software components, hardware components and interfaces equipped with control, computation, communication and synchronization mechanisms may directly integrate monitoring functions to evaluate SoC performance characteristics, thus entirely hiding internal access to the statistical API of the modeling object from the user.

System performance metrics identify system bottlenecks and involve recording of *instant values* commonly in time-driven simulation and *time duration values* usually in event-driven simulation. Thus, we have developed executable specifications of a monitoring library based on two general monitoring classes for collecting instant and duration values from system components, such as processor cores, memory tiles, on-chip interconnect, DMA controller, IOMMU and virtualized peripheral devices.

More specifically, in *time-driven* statistics, attributes have instantaneous values, such as the length of a FIFO at a given time. These values are simply recorded in the `StatInstant` class using the function
`void stat_write (double time, double value).`

In *event-driven* statistics, arrival and departure time (or duration) is recorded. Since departure time is later in time, the `StatDuration` class relies on two member functions. At first, the user invokes
`index = (uint64) stat_event_start(double arrival_time)`
to record the arrival time `arrival_time` and save in integer variable `index` the unique location of this event within the internal table of values. Then, when the event's departure time is known, this time is recorded within the internal table of values at the correct location and the time difference is returned by invoking a call to
`double stat_event_end(uint64 index, double departure_time)`

Based on the previous statistics classes (`StatInstant` and `StatDuration`), we may also define:

- *derived classes* for expressing statistics for various system-specific metrics, such as average throughput, average hit ratio, latency, average size, and packet loss. Thus, `StatInstantAvg` and `StatInstantCAvg` inherit from the top-level `StatInstant` class and provide the functionality required for expressing throughput and average size within a user-defined time window.
- *joint or merged statistic classes* that combine statistical data from different modeling objects, e.g. from various hierarchical memory units in order to compare average read vs. write access times.

Collection of statistical information is enabled using the primitive,
```
void enable_stat(sc_time start=0, sc_time end= DBL_MAX,
                 long int time_window=1,
                 char* x_axis = "X Axis", char* y_axis = "Y Axis");
```
where the function parameters are:

- the starting time (`start`) for statistical collection,
- the end time (`end`) for statistical data collection
- the time window (`time_window`, default 1) for cumulative statistics, i.e. the number of consecutive points which are averaged in the statistics,
- the label for the axis x (`x_axis`), and
- the label for the axis y (`y_axis`).

As explained before, the statistics based on `stat_write` and `stat_event_start/end` are either performed by the user, or possibly by the hardware modeling objects (SystemC-based classes of device subsystems) using library-internal object pointers. In the latter case, software probes can be inserted into *the source code of library routines*, either manually by setting sensors and actuators, or more efficiently through a monitoring segment which automatically compiles the necessary probes (using standard non-intrusive C/C++ profiling mechanics). Software probes share resources with the system model, thus offering small cost, simplicity, flexibility, portability, and precise measurement in a timely, frictionless manner.

Notice that differentiation between various types of latencies (e.g. for different virtual machines, priority classes or virtual circuits) can be obtained by saving the value of `index` in internal arrays to the device class and recalling it later, possibly through appropriate hash functions.

We have already parameterized the above statistics for certain VP library objects, especially processor cores, routers and DMA controller with associated network interface. We can also easily statistically parameterize shared memory tiles, registers/buffers (FIFO, LIFO, Circular FIFO), and IOMMU. In addition to throughput for read/write access and delay statistics for consecutive read/write operations that are normally provided for all these objects, we may also provide average and instant size for register/buffer objects and Cache hit ratios for memory

objects (e.g. for IOMMU TLB). If necessary, specialized classes can also be derived for advanced statistics, such as cell loss probability.

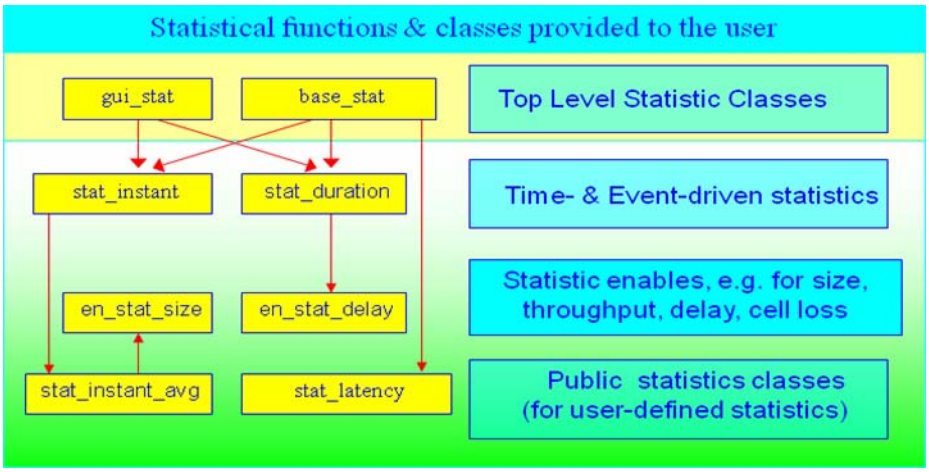

Figure 5. Statistical API for system-level monitoring.

Figure 5 shows an outline of the statistics classes. Statistical data including results from `sc_trace` functionality can be further processed using visualization software, e.g. the open source *Grace* tool or dumped to a file for offline processing, e.g. via an electronic spreadsheet. With Grace alone, it is possible to perform:

- XY graph, XY chart, pie chart, polar and fixed graph layout types; A typical example of *Grace* printout is shown in Figure 6 below.
- user-defined title scaling, ticks, labels, symbols, line styles, fonts, colors composed from enable_stat_ parameters. Notice that units and legends listing the object name and time_window are computed automatically. Figure numbers are also included in the file names; this helps in organizing multiple graphs.
- different post-analysis functions, such as merging, validation, cumulative average, curve fitting, regression, filtering, DFT/FFT, cross/auto-correlation, sorting, interpolation, integration and differentiation,
- custom analysis using its internal language and/or through dynamic module loading in general purpose languages, such as C or Fortran,
- export to PS, PDF, GIF and PNM formats.

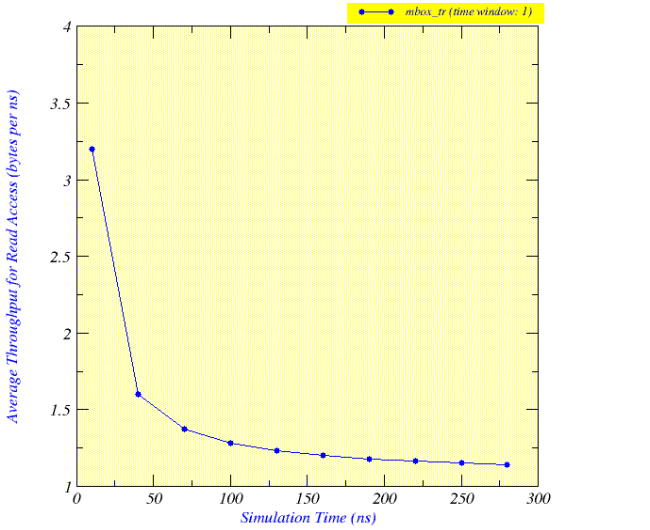

Figure 1. Time-driven statistics: Average Throughput for Read Access vs Simulation Time

Figure 6. Typical Grace printout from statistical library.

5. System Monitoring and Adaptation Scenarios

The rapid evolution of electronic system-level (ESL) methodology focuses on the functionality and relationships of the primary system components, separating system design from implementation. This greatly decreases the number of parameters and constraints in the design space, thus extremely simplifying optimal design selection and verification efforts. Similar to near-optimal combinatorial algorithms, e.g. travelling salesman heuristics, system-level models effectively prune away poor design choices by identifying bottlenecks, and focus on closely examining feasible options. Thus, for the design of multicore SoC platforms, system-level modeling provides rapid, high quality, cost-effective design in a time-critical fashion by evaluating a vast number of communication configurations.

Within this context, our research and development efforts have partly focused on innovative system-level monitoring techniques of embedded SoC and NoC-based multicore SoCs, including cost-efficient, non-intrusive system-level design primitives, protocols and supporting libraries.

The collected monitoring information can be exploited at different system levels by distributed system controllers, operating system and hypervisor managers to dynamically optimize operating system, hypervisor and application performance, scalability, power consumption, reliability and overall quality of experience with a small relative cost. It is also possible to improve quality of service through fundamental network control schemes (limiting bandwidth, latency, buffer management) or more sophisticated latency hiding, mapping or task migration and power (and thermal) management mechanisms. Similar monitoring mechanisms can also be used to provide assurance for early detection and recovery of temporary or permanent faults or hazards (e.g. deadlocks or livelocks).

In order to illustrate the effect of multicore SoC monitoring at operating system and hypervisor level, we have appropriately configured our SystemC-based clock-accurate, transaction-level virtual platform (VP) prototype equipped with a unified, generic and flexible time- and event-driven monitoring API. More specifically, using this VP, we have considered two distinct testbenches that exploit virtualization-specific monitoring information and adapt SystemC simulation and/or partitioning tools to efficiently manage resource allocation to static and dynamic virtual machine workloads, enhancing runtime system flexibility and improving performance and power-efficiency. More specifically:

- The first case study concentrates on interactions between an intelligent runtime monitoring scheme, a distributed hypervisor and two virtual machines (`VM0` and `VM1`). `VM1` runs on processor `CPU2` and performs dummy shared memory accesses, while `VM0` executes a concurrent array processing application (matrix multiplication) on `CPU0` and `CPU1` by invoking a physical shared memory. Hypervisor threads not only implement a specific memory bandwidth-sharing service-level policy, but also help perform application-level load balancing, thus minimizing `VM0` execution time.
- The second scenario considers the effect of static and dynamic frequency scaling by relating the efficiency of relative interconnect power dissipation to the granularity of a homogeneous system of distributed dynamic power management (DPM) controllers.

5.1. Hypervisor SLA & Load Balancing for Efficient Scientific Applications

While the obvious way of improving multicore performance is recompiling the application to execute on more cores, this alone is often not enough. In the context of enforcing service-level agreements (SLA) among VMs and improving application concurrency of VMs, we examine interactions among an intelligent runtime monitoring scheme, a distributed hypervisor and two virtual machines (`VM0` and `VM1`), all residing on the same SystemC VP, and evaluate the performance of distributed hypervisor processes that perform system and application monitoring, dynamic allocation and load balancing.

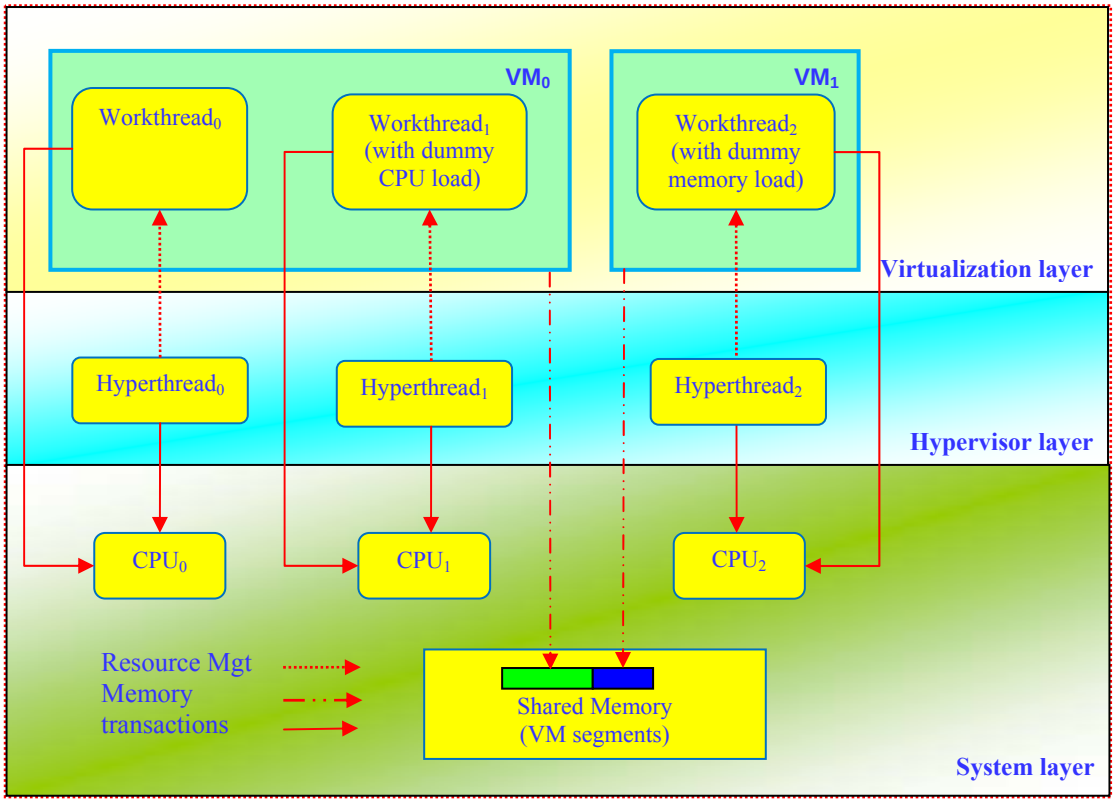

Figure 7. Configuration showing system layers and their interactions.

System configuration (see Figure 7) assigns one hypervisor thread and virtual machine `VM1` to run on `CPU2`. The hypervisor thread `Hyperthread2` on `CPU2` implements a system-specific control thread which performs scheduling tasks, such as initializing globally shared and local data (e.g. for barrier instruction) and initializing `VM0` by loading the parallel application instructions onto processors `CPU0` and `CPU1`, and initiating the execution. `VM1` represents an application thread which performs dummy shared memory read/write accesses on remote shared memory.

As shown in Figure 7, `VM0` implements two hypervisor threads: `Hyperthread0` on `CPU0` and `Hyperthread1` on `CPU1`, and executes a compute- and memory-bound concurrent application micro-kernel (matrix multiplication) by implementing two application threads: `Workthread0` on `CPU0` and `Workthread0 CPU1` by performing simultaneous array accesses to a physical shared memory. The two hypervisor threads on `CPU0` and `CPU1` not only implement a specific memory bandwidth-sharing service-level policy, but also help perform application-level load balancing with the objective of minimizing `VM0` execution time.

In particular, the distributed hypervisor threads at `CPU0` and `CPU1`, which coexist with corresponding application work threads on `CPU0` and `CPU1`, use runtime hardware monitoring support (memory controller statistical counter units) to measure and assess statistic metrics related to the number of memory accesses by each VM in a time window. In this manner, it is possible to dynamically adjust available memory bandwidth and interconnect capacity among the two virtual machines by spreading `VM0` accesses in a time window (corresponding to processing an array slice) in order to meet constraints from a specific, possibly budget-based service-level agreement (SLA).

At the same time, the distributed hypervisor threads at `CPU0` and `CPU1` monitor the relative processor load by examining the latency for completing the previous assigned operations, thus enabling application-level load balancing. Symmetry in the number of hypervisor and application work threads is required to enable distributed non-intrusive monitoring. More specifically, in between array computations corresponding to a given *slice of computations*, hypervisor threads help perform dynamic workload reassignment by allowing the application to adjust the number of local and shared memory operations assigned to each application work thread. This ultimately

optimizes operational characteristics of the scientific application, e.g. minimizing the total VM0 execution delay by reducing the time work threads wait for each other.

In this context, we assume a controlled simulation study, where system components (compiler, processor, memory and interconnection network) remain fixed. In fact, the following VP architecture and application parameters are appropriately specified.

- Queue size at network router and memory controller: 16 packets
- Period time for router clock: T_{router} = 25 or 50 ns,
- Period time for CPU clock: T_{cpu} = 25 or 50 ns,
- Period time for memory controller clock: T_{memory} = 100 ns,

In respect to the application and load balancing strategy, we use the following parameters:

- Array size: N = 128 and number of slices: Slices = 8 or 16; the size of each slice is determined from Slice_Size = N/Slices.
- External load (called dummy load) applied only on CPU1 of VM0; this extra load is applied to each instruction run by the application work thread on CPU1 and causes an extra delay from 50 to 350 ns. This dummy load generates imbalance in the relative cpu loads, i.e. besides the parallel application, thus justifying the role of load balancing
- Memory transaction load (called memory load) applied on the shared memory block by CPU2 of VM1; more specifically, CPU2 sends a remote read/write packet every 60 ns (or 240 ns) to model high (resp. low) memory traffic for VM1. In both cases, VM0 memory transaction rate is initially higher than VM1. This dummy load generates imbalance in the memory traffic, thus validating how our approach is able to meet SLA constraints.
- Load balancing correction step (called Step); this parameter (selected as 2 or 4) affects the sensitivity of the load balancing, triggering reassignment if and only if |CPU1_load – CPU0_load |> Step; the reassignment adjusts the application workload by ±Step/2 in the *next computing phase*. Notice that at any time the sum of the two scientific application workloads equals the Slice_Size.

In our implementation, each hypervisor thread in VM0 is a generic SystemC thread component that receives on-the-fly feedback on the processor load through appropriately enabled system or application monitors embedded in the code of the work threads. Then, the hypervisor thread performs the necessary shared memory barrier synchronization and read/write operations to dynamically balance the workloads assigned to work threads.

Processor CPU_i ($0 \le i \le 1$)	

Hypervisor Thread;
start_Work_Thread(tid, cpuMode);
for(i=0; i<NO_SLICES; i++) {
  start_timer;
  // wait for work threads
  // to complete current slice
  barrier_wait(barrier2, 4);
  end_timer; //workthread delay
  read_cpu_loads_from_mem();
  load_correction();
  set_next_loads();
  if (mem_trans_ratio_enabled){
  read_mem_load();
  adjust_mem_bandwidth();
  }
  barrier_wait(barrier3, 4);
}
	WorkThread;
	read(tid, cpuMode);
	for(i=0; i<NO_SLICES; i++) {
	  read_next_load();
	  matrix_multiply(Slice,
	  start_index,
	  next_load);
	  start_timer;
	  // synchronize work threads
	  barrier_wait(barrier1, 2);
	  write_cpu_load_to_mem();
	  barrier_wait(barrier2, 4);
	  // Work threads wait for
	  // hypervisor threads to read
	  // compute & write next loads
	  barrier_wait(barrier3, 4);
	  end_timer; // sync overhead
	}

Figure 8. Symmetric hypervisor and work threads running parallel scientific application; initializations by control processor (threads of `CPU2` in `VM1`) are omitted.

Figure 8 outlines the system software framework consisting of a single program multiple data (SPMD) processor running one hypervisor (for monitoring and load balancing) and one application thread (running the scientific code, such as array multiplication, and another "dummy" load). In fact, application threads run the assembly code of the parallel (multithreaded) scientific application extracted through an ARM cross compiler, i.e. add, subtract and multiply, compare and branch, move and shift, local load/store, while remote shared memory accesses and synchronization primitives (e.g. barriers based on LLSC atomic operation) are all mapped to the corresponding macro-operations supported by the memory subsystem of the VP. Notice that application processing is performed in consecutive computing phases, whereas during each phase, the distributed hypervisor assigns to processors (work threads) a given workload corresponding to processing a fixed size slice of the shared memory array.

- Initially, at the start of the algorithm, the first workload may be equally shared among all work threads or split according to a static system metric, e.g. relative processor performance.
- At the end of each computing phase, when all processor cores have completed their allocated workloads on the given slice, the distributed hypervisor threads monitor relative application or system (NoC, memory) metrics, such as latency or throughput, cache hit ratio or power dissipation. These metrics reflect the relative load of each processor during the previous slice processing phase.
- By calculating and utilizing these dynamic metrics, the hypervisor threads can better infer how the new fixed size slice should be shared among the cores. This load balancing action performed by the hypervisor enables improving application performance (latency and bandwidth) and/or other system metrics, such as dynamic power consumption or system utilization.

In the ideal case, all work threads complete processing of the current slice at the same time and the hypervisor threads do not make any slice adjustments to the work threads when allocating the slice for the next computing phase. This very unusual case clearly presents an overhead to both system and application performance.

Accurate and frequent workload reassignment of the fixed size slice to the processor cores may be necessary in use cases with a high dynamic load variation, e.g. in a heterogeneous multicore SoC environment performing multitasking whereas new processes can be started at any time. This can be accomplished by dynamically adjusting the slice size during runtime. In this case, the relative effectiveness of the synchronization algorithms (e.g. barriers, locks or atomic operations) compared to the parallel scientific computation time determines the optimal choice for the slice size.

In this context, we perform extensive design space exploration to accurately and effectively evaluate the performance of shared memory scientific processing kernels. As a representative simple example of this methodology, in this deliverable report we focus on matrix multiplication with two application processors.

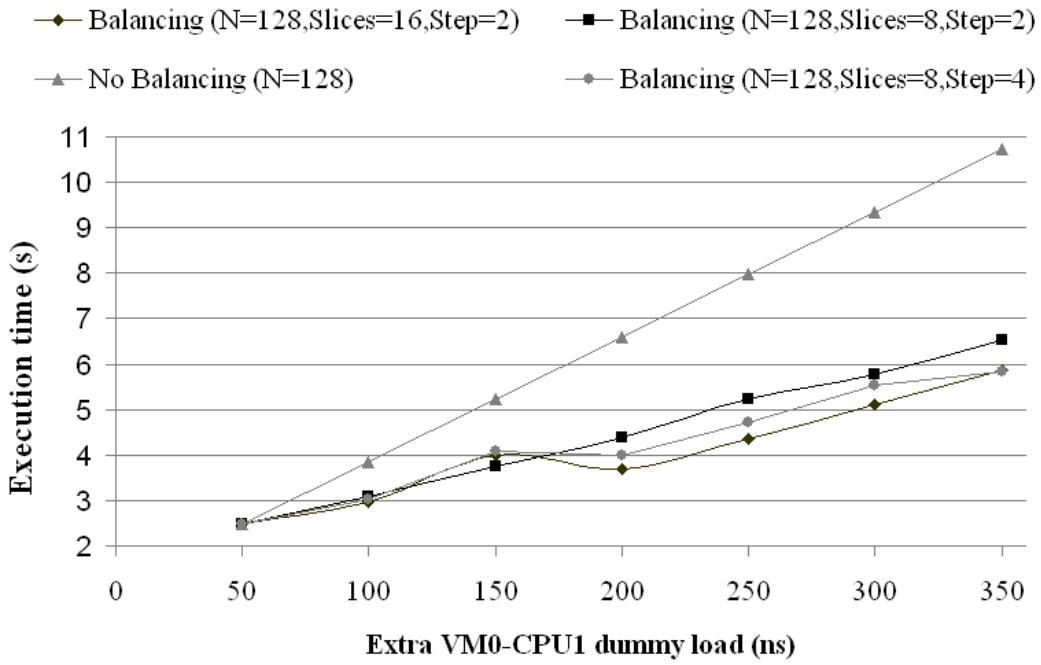

Figure 9. Execution time for parallel matrix multiplication (balancing vs. no balancing)

Figure 9 shows the execution time of the matrix multiplication scientific application vs. the `dummy load` (from `50` to `350` ns) on the `VM0 CPU1` for different combinations of number of `Slices` and `Step`. Notice that for all load balancing scenarios, the execution time improves by `35-45%` compared to the case where no load balancing is used. Moreover, with a higher dummy load, improved relative execution time for load balancing is achieved for a larger number of `Slices` (smaller `Slice_Size`), since balancing is achieved earlier and the extra load is constant.

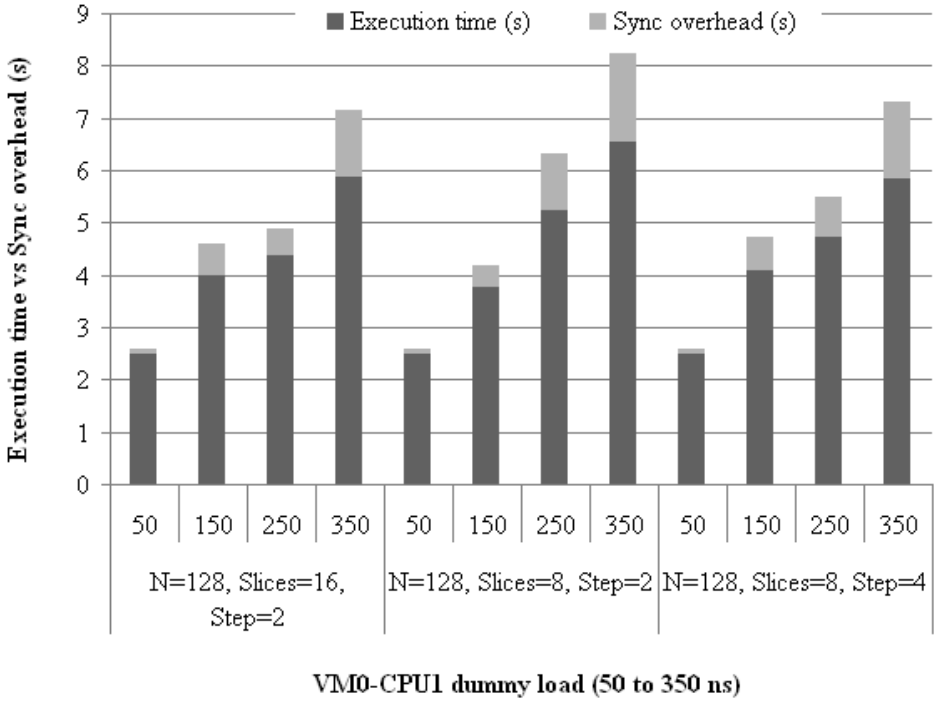

Figure 10. Execution time vs. synchronization overhead for various application configurations

CPU load balancing requires synchronization between the two application threads. This synchronization is supervised by the two hypervisor threads which monitor the actual load on each CPU after each slice and decide on the load assignment for the next computing stage. Since a work thread cannot proceed to the next stage until all hypervisors have calculated and written the next CPU loads to memory, application load balancing introduces thread synchronization, including a small part of local computations, remote write operations for storing the cpu load and barrier. This overhead is estimated by averaging the corresponding execution time from `start_timer` to `end_timer` in the right column of Figure 8 for each participating application thread.

In Figure 10, we show the actual synchronization overhead during the previous execution scenarios, i.e. the extra dummy load on `CPU1` varies from `50` to `350ns`. We observe that for this range, absolute synchronization time deteriorates when the dummy load increases, although in percentage terms, relative synchronization to execution time only slightly increases. This causes relative execution time to scale linearly when load balancing is used (see Figure 9). Finally, notice that a large dummy load may cause balancing to shift the entire workload to `CPU0`. This extreme case happens with a dummy load greater than or equal to `450ns`, in which case the estimated synchronization overhead exceeds `30%`.

Another interesting case on our VP, concerns monitoring and adjustment of the relative memory utilization ratio of virtual machines operating within a multicore SoC environment. This scenario requires maintaining a given memory access ratio between `VM0` and `VM1`, e.g. as specified in a service level agreement. Our proposed memory utilization balancing (MUB) methodology involves adding an extra "sleep" delay time to packets of `VM0` or `VM1`, if the relative `VM0`/`VM1` memory utilization ratio (measuring the number of memory transactions) exceeds the desired ratio during the latest time window.

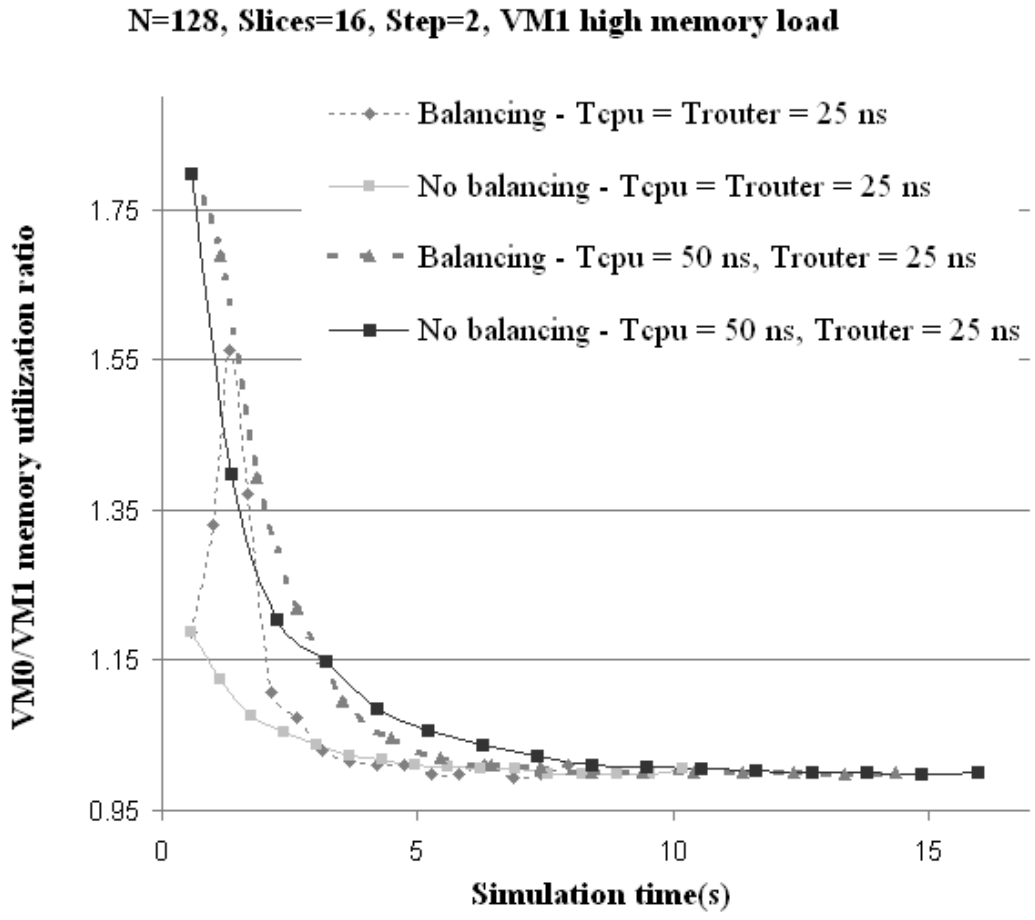

Figure 11. Relative memory utilization ratio of `VM0` vs. `VM1` for high dummy load, high memory load and two different relative CPU vs. router frequencies.

In Figure 11, we consider four scenarios that correspond to varying the relative CPU and router frequencies (T_{cpu} = 25/50 ns and T_{router} = 25 ns), while optionally examining the case of load balancing. We assume a high memory load (memory transaction every 60 ns) and VM0-CPU1 dummy load (300ns extra delay) and a desired VM0/VM1 memory utilization ratio of 1. Moreover, the application starts with its computation load initially shared equally by both application threads. From this figure, we see that without load balancing, the VM0/VM1 memory utilization ratio can be achieved smoothly, quickly and with a smaller number of steps. This is also true if the router is faster than the CPU (T_{cpu} = 50ns and T_{router} = 25ns).

In contrast, when the relative speed of CPU and router is equivalent (T_{cpu} = T_{router} = 25 ns), the CPU load balancing competes against the MUB strategy. After the first slice is completed, the VM0/VM1 memory utilization ratio increases to 1.18 which exceeds a fixed threshold (set to ±epsilon of the desired ratio). Thus, the activated MUB mechanism introduces a small "sleep" delay time for all packets directed from the CPUs of VM0 to shared memory. However, due to the extra load on VM0-CPU1, load balancing is also activated to perform task reassignment. Due to this secondary correction, memory utilization of VM0 fluctuates, since the execution time of this computing stage changes. Therefore, in the initial algorithm stages, correction performed by the previous MUB mechanism is generally not enough. In later stages, when load balancing is achieved, the MUB mechanism is able to bring the ratio of memory transactions quickly down to the desired ratio.

Furthermore, notice that load balancing achieves significantly shorter execution time than when no load balancing is used. More specifically, load balancing reduces the execution time by 10% if the relative speed of CPU and router is equivalent (T_{cpu} = T_{router} = 25 ns); this improvement reaches 20% if the router is faster than the CPU (T_{cpu} = 50ns and T_{router} = 25ns). Notice that the desired ratio affects the actual execution time of the scientific application (with or without balancing), since the MUB policy must appropriately reduce the VM0 memory transaction rate to lower the VM0/VM1 memory utilization ratio.

5.2. Optimizing MPEG4 Power-Efficiency through DFS at NoC-level

Power dissipation is a major concern for efficient multicore SoC architectures, especially for portable embedded platforms with critical power constraints. Although power estimation at RT-level is considered accurate (e.g. [5, 38, 39]), these models rely on time consuming simulation and are not easy to use for optimization purposes due to the huge solution space that must be evaluated.

An interesting system-level power estimation model relates NoC power dissipation to the dynamic power consumed when a single bit of data is transported across two neighboring network routers, taking into account the router's switching fabric, the internal buffers and registers, the wires and the communication link [41]. In fact, when these parameters are constant, the expected system-level dynamic power for transferring one bit of data between two tiles is proportional to their relative routing distance (number of hops). This approximation is considered accurate for NoC topologies and has been used extensively in system-level NoC research, e.g. [9, 38, 40]. For example, experimental NoC power estimation metrics derived from this model are very similar to results obtained from the Orion micro-architecture-level power-performance interconnection network simulator [38], with a relative error less than 10% and an average error of only 4.2% for a series of benchmarks [9].

Automated power-aware embedding of application IPs (specified as communication and computation task graphs) onto NoC-based multicore SoC architectures is critical to performance and power-efficiency tradeoffs. In this context, we propose a multi-tiered system-level power management approach able to improve power-efficiency depending on multicore application requirements and system constraints.

Power management can be applied to processor cores, routers and communication links either offline or online in a static or dynamic manner. It aims to reduce static power by shutting down either totally or partially unutilized or idle resources and also dynamic power by performing DVFS. Notice that static frequency scheduling suffers from inaccurate compile-time estimation of execution time, communication delays and workload variations.

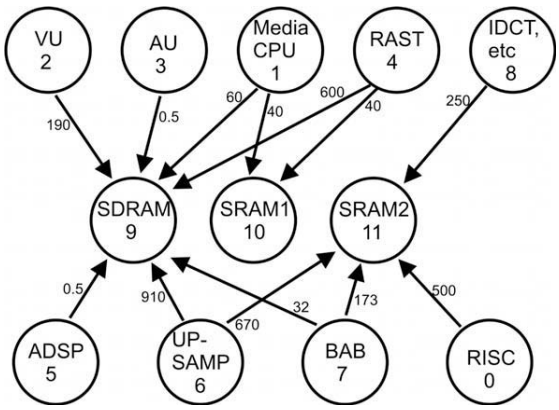

Figure 12. The MPEG4 application task graph.

Within this section, we mainly focus on dynamic frequency scaling for the MPEG4 decoding application under soft real-time application constraints shown in Figure 12; notice that SDRAM, SRAM1 and SRAM2 are passive storage elements, while other nodes are active processor cores, generating packets at highly different rates (e.g. compare `1580 MB/s` bandwidth for UP-SAMP to `0.5 MB/s` for ADSP blocks).

MPEG4 was first examined as an application graph by van der Tol and Jaspers [35] and has recently been used by Murali and De Micheli as an interesting application for testing NoC simulation environments [13, 22, 23]. The original chip was designed to accomplish multimedia operations more efficiently by using dedicated processor cores. In our setting, processor cores use single-beat 128-bit master/slave data interfaces to connect to the hypercube NoC interconnect. The NoC (router and memory controller) hides some of the network latency by using similar high throughput interfaces, while the slower memory array uses a 64-bit interface with a 2-beat burst transfer mode. Similar to other common multimedia applications, such as Video Object Plane Decoder (VOPD), DVD playback, audio player, music synthesizer and video capture, MPEG4 communication-intensive traffic requirements provide soft real-time (latency or packet rate) constraints, i.e. failing to meet them results in degraded quality of the user experience.

Multimedia application models have been widely used in computing performance and power efficiency metrics of NoC-based multicore SoC topologies. For example, Hu and Marculescu examined mapping of a heterogeneous 16-core task graph representing an audio-video application, onto a Mesh NoC topology [13]. Murali and De Micheli used a customized tool (called Sunmap) to map a 12-core heterogeneous task graph representing a video object plane decoder (VOPD) and a 6-core DSP filter application onto a 2d-Mesh or 2d-torus NoC topology using different routing algorithms [22, 23]. The proprietary Sunmap tool, proposed at Stanford and Bologna Universities, performs RTL-level NoC topology exploration by minimizing area and power consumption requirements for different application models (e.g. MPEG4), maximizing performance for various routing algorithms. Another study, supported by SystemC simulations, utilized open source graph-theoretic partitioning and visualization tools, such as Neato, Nauty, METIS and Scotch, to study near-optimal static mappings of synthetic and realistic application task graphs onto different NoC topologies [6]. Virtualization studies related to system monitoring has only recently been considered, e.g. using KVM [43].

Power management can be applied to processor cores, routers and communication links either offline or online in a static or dynamic manner. It aims to reduce static power by shutting down either totally or partially unutilized or idle resources and also dynamic power by performing DVFS. Notice that static frequency scheduling suffers from unpredictable compile-time estimation of execution time, lack of efficient and accurate methods for estimating task execution times and communication delays.We configure our SystemC virtual platform architecture to run a transfer speed test where processor cores transmit a fixed amount of request packets to memory tiles proportional to the rates in the MPEG4 decoder task graph. In order to maintain the very high packet rates required by MPEG4, NoC buffer sizes at the sender/router modules have been assumed to be large enough to avoid packet loss. Sender clock is at the highest frequency (`1ns`). Memory tiles respond to each request packet by

instantaneously issuing a reply packet for each received request. Request and reply packets are 256-bit long, carrying a 128-bit data payload. In the transfer speed test, we also assume that each processor core uses a packet injection rate proportional to the highest demanding PE (UP-SAMP in Figure 12). Notice that the UP-SAMP (core 6) communication requirements exceed a rate of 1580MB/s.

In order to evaluate the effect of static IP mapping when using dynamic frequency scaling, we set up a transfer speed test (with the same architectural parameters), whereas a Scotch-based near-optimal embedding of the MPEG4 task graph resources (IPs) onto the 4-cube NoC topology of 16-nodes obtained with offline partitioning is implemented. Scotch has been developed at the University of Bordeaux I and provides efficient libraries based on recursive bi-partitioning for statically mapping any possibly-weighted source graph (G_S) onto any possibly-weighted target graph (G_T) [27, 32]. The cost function used in Scotch minimizes the *maximum edge expansion over all edges of* G_S, where the *edge expansion of an edge of* G_S multiplies the length of the path in G_T onto which an edge of G_S is mapped (called edge dilation) with its corresponding edge weight. This embedding quality metric minimizes the average latency overhead of a weighted task graph.

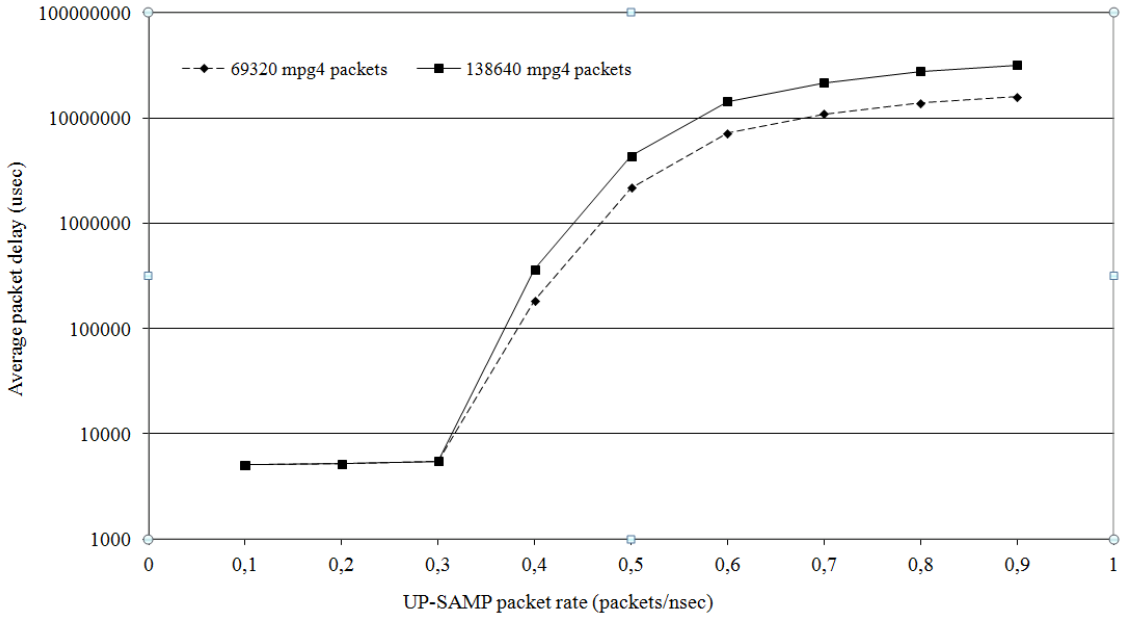

Figure 13. Average packet delay vs. packet injection rate of the UP-SAMP.

In Figure 13, we compare the average packet delay (for requests and replies of all IPs) versus an increasing packet injection rate applied to the UP-SAMP block (or equivalently a packet interarrival time) for two cases: 69320 and 138640 total MPEG4 packets; notice that very similar behavior would have been observed if we had considered average network round trip time, reply packet rate, or average router buffer size for all initiators. The minimum (required) packet injection rate for the UP-SAMP (shown as the first x-axis point in Figure 13) must satisfy the MPEG4 application traffic requirements shown in Figure 12. More specifically, since useful data in a NoC request or reply packet is 128 bits, the first x-axis point for UP-SAMP is computed as: 1580MBytes/s * 8 Bits/128 Bits = 98.75M packets/s=0.09875 packets/ns. Moreover, from Figure 13, we observe that the average packet delay is initially stable, but starts increasing exponentially for a packet injection rate between 0.3 and 0.4 packets/ns when the NoC saturates; if small buffer sizes were used, this would have caused an exponentially increasing number of dropped packets, or an equivalent number of retransmissions if memory/NoC back pressure was enabled. After this saturation point, the router becomes insensitive to the offered load, but continues to work at the maximum possible rate. Thus, average packet delay stays constant (at a maximum point), while the initiators' output buffer size diverges to infinity. Based on this discussion, we observe that a stable injection rate for operating our MPEG4 speed test is set to 0.2 packets/ns for the UP-SAMP and proportionally for the remaining initiator blocks. With the asynchronous memory controller

and a router frequency of 1ns, this rate is enough to sustain the required MPEG4 bandwidth, while also allowing for power optimization. Moreover, system behavior at this injection rate is similar for both smaller (69320) and larger (138640) number of packets; this means that we can utilize the smaller number of packets for our simulations at this rate.

Static frequency scheduling refers to (usually offline) allocation of single or multi-frequency levels to certain frequency scalable resources regardless of its utilization during runtime. Dynamic frequency scaling (DFS) refers to allocation of single or multiple frequency levels for running tasks on frequency scalable resources and can also be performed offline or online. By assigning a lower operating frequency to certain routers mapped on frequency scalable resources, we effectively slow them down, exploiting the available slack; however notice that frequency switching overhead is not always negligible.

Reducing dynamic power consumption on the NoC relies on adjusting the frequency (and/or voltage) level of frequency (and voltage) scalable hardware resources, such as routers and links. For the case of frequency scaling, we can adjust the frequency level of a router based on current router performance or buffer utilization. In this context, we propose two interesting monitoring metrics that enable corresponding dynamic frequency scaling policies.

- *Router bandwidth within a time window* (called PacketOut policy); this frequency scaling mechanism relies on comparing the total packet rate metric from all output queues of the router within a time window (specified through a sampling rate or from simulation start) with the *harmonic average packet rate* for all routers; harmonic average corresponds to simple average computation on the time scale. Divergence from this average (by a certain threshold) is used to define frequency up/down scaling (router power transitions: DO_UP and DO_DOWN), where DO_UP moves to the next higher and DO_DOWN moves to the next lower frequency.
- *Cummulative router buffer size within a time window* (called BufferSize policy); for this policy, frequency scaling is based on comparing the sum of the sizes of all router queues within a time window (specified through a sampling rate or from simulation start) with the *average sum for all router queues*. Assuming fairness in packet handling, this comparison serves as an indicator for remaining workload to be performed, and therefore, alike the previous case, divergence from the average (by a certain threshold) can be used to perform frequency up/down scaling (corresponding DO_UP and DO_DOWN frequency transitions).

With both DFS policies, a threshold value for DPM enable (simply called DPM_Threshold) introduces the possibility of no frequency scaling (DO_NOTHING decision). Assuming the threshold is defined as a percentage of the average value (e.g. 10% or 0.1), both DFS policies operate as follows. If the current metric is above (resp. below) the average by at least 1+DPM_Threshold (or 1- DPM_Threshold), then frequency up-scaling (resp. down-scaling) is implemented at the corresponding router via a DO_UP (respectively, DO_DOWN) power state transition. Otherwise, DO_NOTHING is implemented (no frequency scaling).

Scaling the clock frequency relies on automatically adjusting the phase-locked loop (PLL) frequency. Thishis has been effectively modeled at behavioral level in SystemC-AMS (time-annotating from low-level electrical circuits); however, this part referring to clock skew minimization falls beyond the scope of this report.

In order to examine the merits of the above DPM policies and the effect of the sampling rate (SR) and threshold (DPM_Threshold), we have considered the same MPEG4 application scenario, assuming that the packet rate of the UP-SAMP is initially set to 0.2 packets/ns; the remaining initiator IPs are proportionally set. Based on our simulations, we have found that this injection rate is below saturation which occurs above 0.3 packets/ns. Moreover, it is sufficient to sustain the required MPEG4 bandwidth with an asynchronous memory controller that consumes data and returns a reply packet or acknowledgment immediately at the following router cycle and a router frequency of 1ns, while also allowing for power optimization. Moreover, router arbitration latency is variable from 0 to 4 cycles (we use the open source *helix* router), while all twelve active routers, which are organized in a 4-cube, initially operate at 1GHz and use alternate frequencies: 0.5 and 0.25 GHz. System behavior at this injection

rate is similar for smaller (69320) or larger (138640, etc) number of packets; hence, we use the smaller value in our simulations at this rate.

Figure 14 shows the relative NoC power when DPM is enabled with DPM_Threshold = 0.4. As discussed previously, NoC power is estimated by multiplying the number of packet hops with the corresponding frequencies. Notice that for the PacketOut policy, the execution time remains relatively constant (176us vs. 158us), although individual routers scale their frequencies much more often than the BufferSize policy. For the BufferSize policy, the execution time increases by 60% to 254us. This extra slack time provides the opportunity to reduce the average relative NoC power by 78% for PacketOut and 55% for BufferSize policy. Moreover, if the sampling rate increases, the maximum power fluctuates more (sometimes exceeding maximum power without DPM, since router-specific frequency scaling decisions become more chaotic. Finally, for larger values of DPM_Threshold (above 0.6), no further slack in the execution time can be obtained (graphs omitted due to space restrictions).

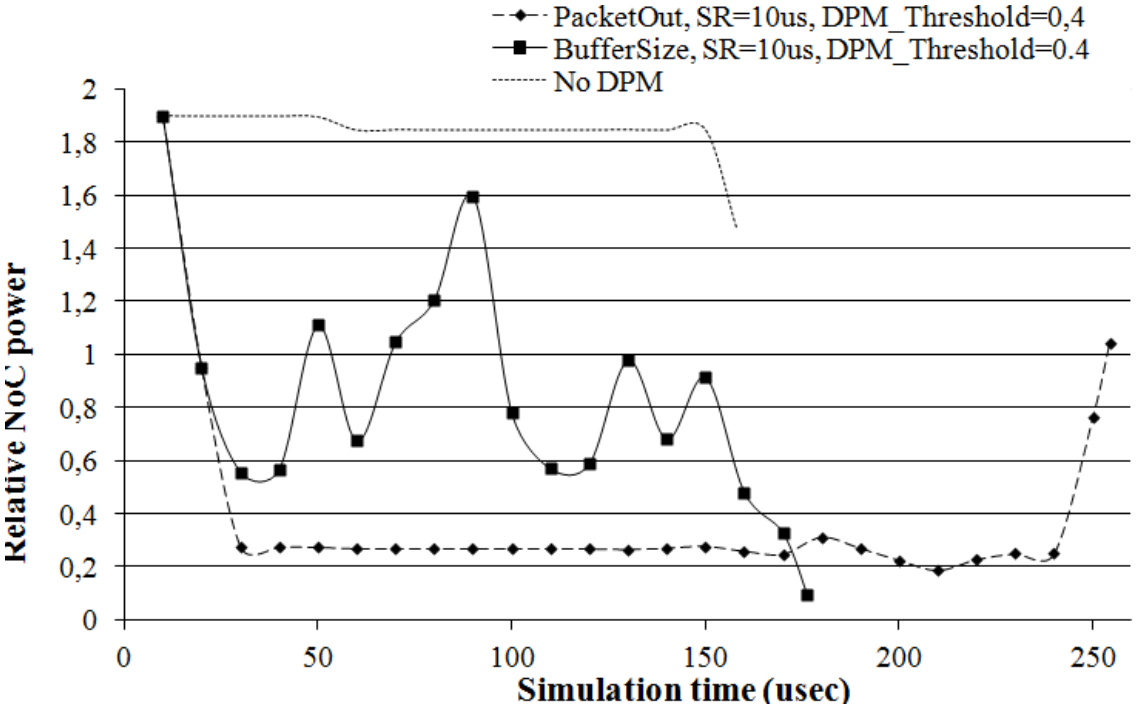

Figure 14. Relative NoC power vs simulation time (us)

We next consider an extension of our DPM policy to soft real-time. In this case, the DPM module predicts the *expected finish time* and compares it with the required deadline before issuing a router policy decision; this prediction only slightly increases the DPM protocol complexity.

In order to forecast the finish time, we monitor the cummulative rate of acknowledgment packets at the outgoing port (0) of all hypercube routers connected to initiator IPs, i.e. we examine (Port_0_PacketOut[i], where i: Initiator). By accumulating this monitoring information together with the application start time (start_time), the remaining packets from initiator MPEG4 IPs still waiting to be sent (remaining_packets[i], i : Initiator) and the current frequency level, the DPM real-time module can estimate the expected finish time of each initiator IP at regular sampling intervals (SR). The maximum of all predictions defined by each of the nine hypercube NoC initiator IPs, marks the expected finish time. Then, by characterizing the relation between the expected finish time and the required soft real-time deadline, the DPM real-time module is able to make smart decisions based on the current status of the MPEG4 application.

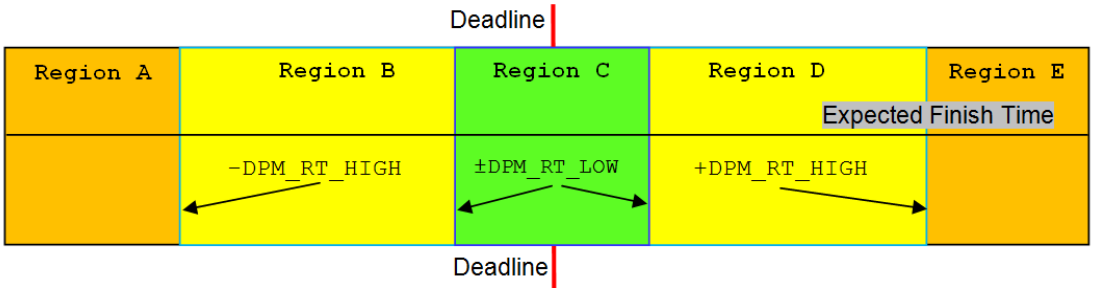

Figure 15. The proposed real-time DPM model.

More specifically, our DPM real-time policy implements two additional thresholds (`DPM_RT_LOW` and `DPM_RT_HIGH`, where `DPM_RT_LOW` < `DPM_RT_HIGH`). This allows a hierarchy of DPM real-time decisions that take into account the relation between the expected finish time and the required deadline and follow a well-defined five state power characterization: `{DO_ALL_DOWN, DO_DOWN_NOTHING, DO_ALL_NOTHING, DO_UP_DOWN_NOTHING, DO_ALL_UP}`.

- `DO_ALL_DOWN` is a basic power state entered when the expected finish *time* is smaller than the given deadline by at least a `DPM_RT_HIGH` percentage (`Region A` in Figure 15); this state is also entered if the current time is before the deadline and the block has no packets to send (i.e. the soft deadline has been met).
- `DO_DOWN_NOTHING` is a hierarchical superstate with two router-specific decisions `{DO_DOWN, DO_NOTHING}`; this superstate essentially uses the same principles as previously discussed for the DPM case (no real-time case), although the `DO_UP` state is actually merged with `DO_NOTHING`, since it is not required to perform frequency up-scaling. Thus, the `DO_DOWN_NOTHING` superstate makes **router-specific decisions** according to the harmonic averages using the `DPM_Threshold`. This decision corresponds to the closed `Region B` in Figure 15.
- `DO_ALL_NOTHING` is a basic power state entered when the expected finish time is smaller than the given deadline by at least a `DPM_RT_HIGH` percentage (`Region C` in Figure 15); this state is also entered if the expected finish time cannot be dtermined, e.g. when packet rate (`PacketOut`) monitoring info is not available; an alternative, would have been to perform the same decision as before.
- `DO_UP_DOWN_`NOTHING is a hierarchical superstate with three router-specific decisions `{DO_DOWN, DO_NOTHING, DO_UP}`; this superstate uses the same principles as previously discussed for DPM case (without real-time) and makes *router-specific decisions* based on the harmonic averages using the `DPM_Threshold`. This decision corresponds to the closed `Region D` in Figure 15.
- `DO_ALL_UP` is a basic power state entered only when the expected finish time is larger than the given deadline by at least a `DPM_RT_HIGH` percentage (`Region E` in Figure 15); this state is also entered if the current time is past the deadline and the block has remaining packets to send.

In order to monitor the packet rate at the outgoing port of all hypercube routers we require hardware counters with an average buffer size that depends on the DPM policy. However, this overhead is not significant in terms of circuitry area. In addition, since the DPM module needs to collect all measured statistics before making a decision we use a separate virtual circuit. We preferred this solution against adding extra communication links due to reduced cost and minimal perturbation. Moreover, instant monitoring (i.e. within a time window controlled by the sampling rate) may cause variation in traffic and consequently false DPM decisions. In this case, which is not explored in this report, the hardware DPM module may intelligently attempt to skip false indications by maintaining two past decisions to avoid ping-pong effects.

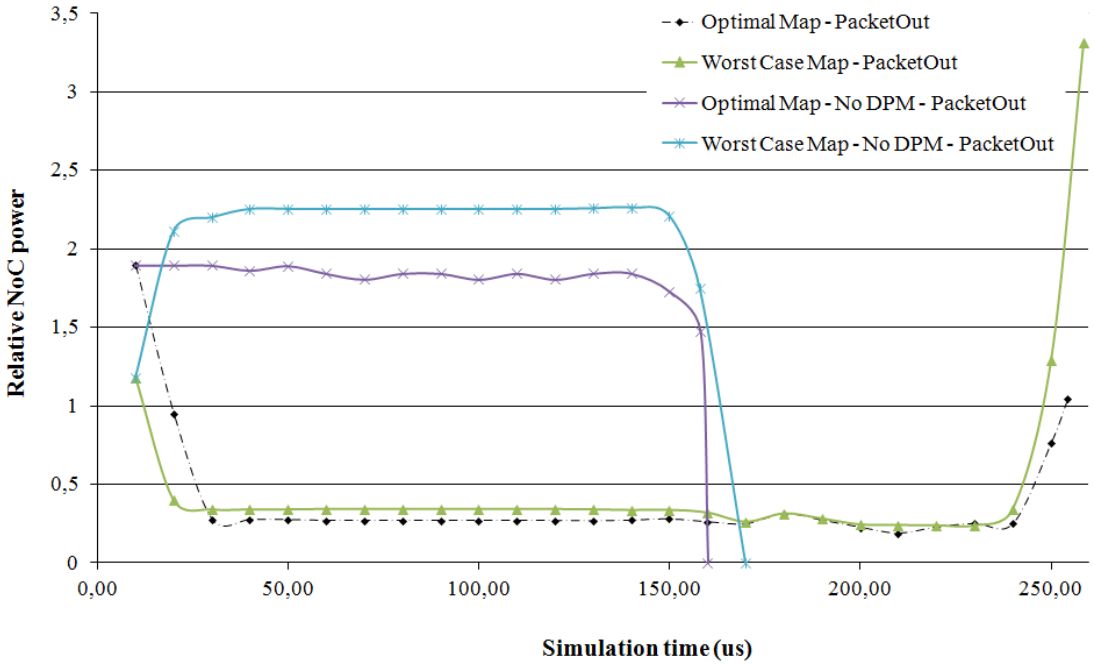

Figure 16. Performance of the proposed soft real-time DPM policy.

In Figure 16, we compare the relative NoC power for the soft real-time DPM policy using the `PacketOut` monitoring policy (results from `BufferSize` policy are similar) to the case when there is no DPM, considering both optimal and worst-case mapping of the MPEG4 IPs. We assume a DPM policy with these parameters: `DPM_Threshold = 0.1`, `DPM_RT_LOW = 0.08`, `DPM_RT_HIGH = 0.16`, a soft deadline of `240us` and (as before) sampling rate `SR=10us`, with an initial operating clock frequency at `1GHz` (alternative frequencies at `0.5` and `0.25 GHz`). Prediction of the estimated finish time by the MPEG4 IPs is based on cumulative acknowledgment traffic statistics on their communicating port (`0`). From Figure 16, we observe that our real-time DPM policy is sensitive to the deadline, achieving a fine balance by dynamically reorganizing IP accesses in time to achieve power-efficiency and meet the soft real-time deadline. Our DPM real-time policy adjusts the frequency rate, reducing average relative NoC power compared to no DPM case by almost `80%` (for an optimal Scotch-based mapping). Moreover, Figure 16 shows that the effect of embedding of the IPs on the NoC topology is significant: worst-case mapping of IPs results in reducing the relative maximum NoC power-efficiency by `16%` and the average power efficiency by more than `80%`. Hence, using near-optimal static embedding of the MPEG4 application IPs onto the hypercube NoC enables extremely better power savings than resorting to real-time DPM policy with a random mapping. Further improvements to DPM power are expected from changing the MPEG4 initiator speed; this issue is beyond the scope of this section which concentrates strictly on relative NoC power-efficiency.

6. Conclusion and Future Extensions

VOSYS has proposed a generic methodology for system wide profiling on virtualized systems based on hardware and also software counters, detailing its innovative aspects and identifying related complexity issues. It is particularly interesting to consolidate and standardize these extensions in the context of standardizing interfaces for virtualization management. Moreover, clarifying the association between the generic methodology and the ARM Performance Monitor Unit (in particular its register set) is an open question.

TEI has developed a SystemC virtual platform of multicore SoCs (to be released as an open source software package in 2013) which enables the development of innovative system-level monitoring strategies for designing high performance, power-efficient and reliable adaptive NoC-based multicore SoCs, including hypervisor and application models. Within the context of WP4 deliverables D4.1 and D4.4, the virtual platform will be used to simulate, validate and analyze innovative macro-architectural features which deal with IOMMU design. Furthermore, several interesting extensions are open for further investigation.

- By monitoring virtual machine and process identifiers, we can optimize hypervisor performance, scalability, reliability and overall quality of experience with a small relative cost.
- Standardizing the API of the system-level monitoring library is essential to providing smooth platform-wide interoperability with external tools. In this context, to ensure optimized quality in terms of performance and power consumption, the adoption of specific SystemC classes saving, extracting and visualizing monitoring information for dynamic (real-time) system management is challenging. Existing standards and initiatives that express generic dynamic power management and monitoring data structures and actions, such as Advanced Configuration and Power Interface (ACPI) for PCs, JTAG (IEEE 1149.1) and especially Unified Power Format (IEEE standard P1801), are generally not detailed enough to handle system-level NoC-based multicore SoCs.
- This virtual platform allows calibration of a soft real-time DPM module by appropriately characterizing its operating frequencies, threshold values and sampling rate. Extensions to low-complexity DPM modules may concentrate on fine vs coarse granularity in the organization of the DPM controllers, e.g. by considering independent controller modules assigned to each NoC topology subgraph. In addition, for homogeneous NoC topologies, more sophisticated prediction models can be based on existing network calculus or similar queueing models. Finally, the DPM technique is also useful for selecting the best candidate NoC topology that satisfies certain application constraints even before the target NoC architecture is actually defined, i.e. during virtual prototyping, system-level simulation and design space exploration.

References

1. ARM, A15 Architecture Reference Manual, 2011, available from http://infocenter.arm.com/help/topic/com.arm.doc.ddi0438e/DDI0438E_cortex_a15_r3p0_trm.pdf
2. ARM, Fast Models Reference Manual, November 2011, available from http://www.arm.com/products/tools/models/fast-models.php
3. ARM, LPAE in A15 Architecture Reference Manual, 2011, available from http://infocenter.arm.com/help/topic/com.arm.doc.ddi0438e/DDI0438E_cortex_a15_r3p0_trm.pdf
4. ARM, "Virtualization Extensions Architecture", 2011, available from http://www.arm.com/products/processors/technologies/virtualization-extensions.php
5. N. Banerjee, P. Vellanki, and K.S. Chatha, "A power and performance model for network-on-chip architectures", in Proc. Design Automation and Test in Europe Conf., 2004, pp. 1250--1255.
6. L. Bononi, N. Concer, and M. Grammatikakis, "System-Level Tools for NoC-based multicore design", in Multicore Embedded Systems. Ed. G. Kornaros, Chapter 6, CRC Press, Taylor and Francis Group, 2010.
7. L.L. Chan, "Modeling virtualized application performance from hypervisor counters", MS Thesis, Massachusetts Institute of Technology, 2011.
8. A. C. de Melo, "The New Linux 'perf' tools", 2010, available from http://vger.kernel.org/~acme/perf/lk2010-perf-paper.pdf
9. N. Eisley and L. Peh, "High-level power analysis for on-chip networks", in Proc. Compilers, Arch. & Synthesis for Emb. Syst., 2004.
10. T. Gleixner, "Performance counters for linux", 2008, available from http://lwn.net/Articles/310176
11. P. Greenhalgh, "Big.LITTLE Processing with ARM CortexTM-A15 & Cortex-A7: Improving Energy Efficiency in High-Performance Mobile Platforms", ARM White Paper, 2011.
12. HPCToolkit /HPCView, available from http://hpctoolkit.org/
13. J. Hu and R. Marculescu, "Energy-performance aware mapping for regular NoC architectures", IEEE Trans. on CAD of Integr. Circ. and Syst., 24(4), 2005, pp. 551--562.
14. U.J. Kapasi, S. Rixner, W.J. Dally, et al., "Programmable Stream Processors", IEEE Computer, 36(8), 2003, pp. 54--62.
15. A. Kivity, U. Lublin, and A. Liguori, "Kvm: the linux virtual machine monitor," *in Proc. Linux Symp.*, 2007, vol. 1, pp. 225–--230.
16. Linux containers project, http://lxc.sourceforge.net/
17. Linux kernel virtual machine project, http://www.linux-kvm.org
18. LXC Linux Container, available from http://lxc.sourceforge.net/
19. A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, "Diagnosing performance overheads in the Xen virtual machine environment," in Proc. Conf. on Virtual Exec. Env., 2005.
20. P. J. Mucci, S. Browne, C. Deane, et al., "Papi: An interface to hardware performance counters," 1999, available from http://web.eecs.utk.edu/~mucci/latest/pubs/dodugc99-papi.pdf
21. MUMMI, available from http://www.mummi.org
22. Murali, S. and De Micheli, G., "Bandwidth-constrained mapping of cores onto NoC architectures", in Proc. Design, Automation & Test in Europe Conf., 2004.
23. Murali, S. and De Micheli, G., "SUNMAP: A tool for automatic topology selection and generation for NoC", in Proc. Design Automation Conf., 2004.
24. R. Nikolaev and G. Back, "Perfctr-xen: a framework for performance counter virtualization," in Proc. Conf. on Virtual Exec. Env., 2011.
25. OProfiler, available from http://oprofile.sourceforge.net
26. PAPI, available from http://icl.cs.utk.edu/papi
27. F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs", in Proc. High Perf. Comp.. & Networking, 1996, pp. 493--498.
28. Perfsuite, available from http://sourceforge.net/projects/perfsuite

29. G. J. Popek and R. P. Goldberg, “Formal requirements for virtualizable third generation architectures,” *Comm. ACM*, **17**, 1974, pp. 412--421.
30. PPW, available from http://ppw.hcs.ufl.edu/
31. Scalea, available from http://www.dps.uibk.ac.at/projects/scalea/
32. Scotch, available from http://www.labri.fr/perso/pelegrin/scotch/
33. J. Smith and R. Nair, “*Virtual machines: versatile platforms for systems and processes*”, Morgan Kaufmann Publishers Inc., 2005.
34. TAU, available from Tuning and Analysis Utilities http://tau.uoregon.edu
35. E. van der Tol and E. Jaspers, "Mapping of mpeg-4 decoding on a flexible architecture platform", in Proc. SPIE Int. Soc. Opt. Eng., 2002, vol. 4674, pp. 1--13.
36. VOSYS, “Kvm Cortex-A15 Implementation”, project repository, available from https://github.com/virtualopensystems/linux-kvm-arm
37. VProf, available from http://sourceforge.net/projects/vprof/
38. H. Wang, X. Zhu, L. Peh and S. Malik, "Orion: a power-performance simulator for interconnection networks", in Proc. Int. Symp. Microarchitecture, 2002, pp. 294--305.
39. H. Wang, L. Peh, and S. Malik. Power model for routers: Alpha 21364 and infiniband routers, IEEE Micro, 24(1), 2003, pp. 26--35.
40. T. Ye, L. Benini, and G. De Micheli, "Packetization and routing analysis of on-chip multiprocessor networks", J. Syst. Arch., 50, 2004, pp. 50--81.
41. T. Ye, L. Benini, and G. De Micheli, "Analysis of power consumption on switch fabrics in network routers", in Proc. Design Automation Conf., 2002.
42. Y. Zhang, “Enhance perf to collect kvm guest os statistics from host side”, 2010,available from http://lwn.net/Articles/378778/
43. W. Zwaenepoel, J. Du and N. Sehrawat, “Performance Profiling of Virtual Machines”, in Proc. Virtual Execution Env., 2011.