Composable Processor Virtualization for Embedded Systems

Anca Molnos[1], Aleksandar Milutinovic[2], Dongrui She[3], and Kees Goossens[3]

[1] Technische Universiteit Delft, The Netherlands
[2] Universiteit Twente, The Netherlands
[3] Technische Universiteit Eindhoven, The Netherlands

Abstract. Processor virtualization divides a physical processor's time among a set of virtual machines, enabling efficient hardware utilization, application security and allowing co-existence of different operating systems on the same processor. Though initially intended for the server domain, virtualization gains popularity also in embedded systems, where it is mainly meant to isolate real-time from best effort applications, as they require different types of schedulers and performance guarantees. Moreover, for design productivity and cost reasons, one would like to independently design, test, and verify applications, then easily integrate them on a shared hardware platform. For best effort, this requires composable capacity, i.e. an application's number of processor cycles is independent of other applications. In real-time, performance prediction and guarantees preservation is crucial, thus the timing of an application's cycles should also be independent, i.e. composable performance/timing. Virtualization is a good first step towards these, however existing approaches do not ensure timing composability. In this paper we propose a processor virtualization scheme with: (1) composable timing and capacity, and (2) different task schedulers per application. This scheme is based on: (1) a budget-based scheduler for composable capacity, plus (2) the ability to program application switching at fixed points in time, for composable timing. We implemented this scheme on an FPGA multiprocessor running a synthetic application and experiments show composability.

1 Introduction

Virtualization is a generic technique that logically divides (in time or in space) a physical resource in fractions, creating the illusion of multiple, virtual, resources [1–3]. For a clean separation of concerns and high portability, this partitioning should be user transparent, i.e. each user application utilizes a virtual resource, as if it would be a physical one. Virtualization can be applied at various levels of abstraction and resources. Most servers employ Operating System (OS) virtualization that allows running multiple OSes simultaneously on a computer (or cluster of communicating computers). OS virtualization requires the partitioning of all physical resources (e.g. processors, memories, interrupts) achieving: efficient resource utilization, user transparent load balancing, security, flexibility.

In embedded systems the accent is on low cost, low overhead, low power, making OS virtualization less common. However, embedded systems become more complex, and comprise functionality with various timing constraints, i.e. firm-, soft- real time (RT), and non real time (best effort, BE) applications run concomitantly on a platform. The tasks of these applications are scheduled using various algorithms, depending on their timing constraints (e.g. TDM or priority based for real-time, and Round Robin for best effort). Scheduling a real-time application using Round Robin, for example, might lead to disastrous timing violations, thus some incipient form of low cost OS virtualization is required to be able to safely execute an application mix.

Composability [4–6] increases system reliability and decreases the cost involved in platform integration and verification, enabling a divide-and-conquer design approach. On a composable platform the functional and temporal behaviour of an application is independent of other applications, thus they can be designed, tested and verified in isolation, and then easily integrated.

To verify and integrate best effort applications, the only requirement is that their functionality is independent. However, if an application crashes or attempts to monopolize resources, other applications' functionality (both in case of BE and RT) and timing (in case of RT) should be undisturbed (system reliability). Thus a BE application should have a constant fraction of the total number of cycles on a processor (*capacity virtualization*), irrespective of other applications, i.e. *composable capacity* on each resource.

Real-time applications have strong timing (i.e. response time) constraints and have to be predictable. In RT the maximum duration from the moment an input stimuli is presented to the system (application enabled) till the system outputs its response (application finish) should be bounded by design, and should never be exceeded during execution. An application's response time is given by the its number of processor cycles, and their distribution in time [16, 17] (i.e. the application budget, and the precise moments in time when this budget is granted). If one of these fluctuates due to the behavior, presence or absence of other applications, the response time varies unpredictably. Thus, in RT, the platform has to ensure that not only an application's number of processor cycles, but also their timing is independent of other applications, i.e. *composable timing* (*timing virtualization*). Timing virtualization requires capacity virtualization plus a fixed distribution in time for the application's allocated capacity. Note that composability (applications non-interference) is orthogonal to predictability (the ability to estimate at design-time a bound of the worst case timing of an application). Predicting application's timing requires predictable hardware and a predictable intra-application scheduler, and composable timing preserves the design-time bounds during execution, by eliminating the interference at shared resources.

The contribution of this paper is a processor virtualization technique that (1) offers composable timing among streaming applications consisting of communicating tasks, and (2) allows each application to have its own task scheduler. To achieve composability at cycle resolution, virtualization is assumed for all other shared platform resources, e.g. NoC [7, 8], memory controllers [9], etc. We ensure processor composable capacity by assigning fixed cycle budgets to applications, via a preemptable budget scheduler for applications (i.e. Time Division Multiplex (TDM)). Furthermore, for composable timing we enforce that these

budgets are equidistantly scheduled via a mechanism able to delay an action (the application swap-in moment) until a fixed, future point in time. We denote timing composability shortly, as composability. Last but not least, inside an application we allow different task schedulers (e.g. real-time TDM, or best effort Round Robin, etc.). The only requirement is that these schedulers have to execute in a finite, known time.

We implement these mechanisms on a multiprocessor platform on FPGA. Experiments suggest that application capacity and timing composability is achieved at processor cycle resolution. In the following section we present related work, the targeted architecture and applications, followed by the virtualization mechanism, and ending with experimental results and conclusions.

2 Related Work

The related work falls in two categories: hypervisors and composable scheduling. Typically, a hypervisor is the lowest level firmware that runs on hardware and implements scheduling of virtual machines (VM) on real machine(s), while on a VM there might be a guest OS implementing the suitable intra-application scheduling. Hypervisor allocates the hardware resources dynamically, and transparently to VMs. Usually, it virtualizes memory accesses, file systems, interrupts, I/O, etc. Common hypervisors are, among others, IBM's AIX, VirtualLogix VLX Hypervisor and the L4 micro-kernel. IBM's AIX operating system implements hypervisor functionality [2], with priority-based, or non-preemptive VM schedulers. VMware's with former Trango Virtual processors offers a scheduler in AIX operating system [1] with dynamic priorities and on-line recalculation of CPU time assigned to a VM. VirtualLogix VLX Hypervisor [3] has flavours for mobile handsets and some other specialized embedded systems. Its separate VMs can run an RTOS and a general OS (with rich functionalities, like Linux) simultaneously with their real-time virtualization technology. It mainly runs on single ARM platforms with some hardware accelerators in difference of our scope. The L4 μ-kernel [13] uses hard priorities with round-robin scheduling per priority. Many different flavours are added to the basic L4 implementation resulting in big family of its derivatives. Open Kernel L4 [11] has an OS and virtualization technology, however based on a global scheduling policy interleaving priorities of threads from different subsystems/applications. In these approaches, the number and timing of cycles an application can get depend on higher priority applications', hence they are not composable.

In general, schedulers in a real-time OS (RTOS) can be of different types (priority- and/or budget- based) providing guaranties that deadlines are met. Here we discuss only the ones close to our work, namely budget schedulers. Bekooij et al. [10] propose and motivate the utilization of TDM, laying the foundations of (timing) composable multi-processors scheduling. However, [10] makes no distinction between intra- and inter- application scheduling, and does not discuss the details of processor sharing. Their work is continued in [14] with a priority-based budget scheduler with composable capacity, but no composable timing. Another interesting approach is the composable MPSoC in [6], where all resources are virtualized, except for the processors that are not shared. We assume a similar composable platform, and offer composable processor sharing.

3 Background

We consider the base architecture in [6], consisting of a set of computation tiles, an external memory, all connected by a NoC, as visible in Figure 1, thus the memory [9] and communication [7] resources are composable.

On this platform, we run a mix of real-time and best effort applications. An application consists of a set of tasks that may communicate, while there is no communication between different applications. The code and data of tasks that run on a tile are assumed to fit in the local tile memory. In streaming applications a task executes infinitely: consumes its input data produced by other tasks, executes an iteration then produces its output (to be processed by successor tasks). The inter-task data communication is implemented via uni-directional, finite FIFO buffers located in a memory tile. A producer task sends a token to the external memory (posted write). When the consumer task needs it, it fetches it in its local memory. Inside a token data access can be performed in any order. FIFOs are used through the C-HEAP protocol [12] with blocking read and write routines. We choose this programming model because it perfectly fits the streaming domain, it allows overlapping computation with communication, and - crucial for real-time applications - allows modelling an application as a dataflow graph. Moreover, the architecture, and the binding of applications to the platform can be also modelled with it. Thus dataflow analysis [15–17] is possible, allowing end-to-end guarantees. For the best effort applications the programming model is theoretically not restricted to the one above. However, for practical reasons, we assume an uniform programming model for the entire platform, leaving the relaxation of this assumption for future research.

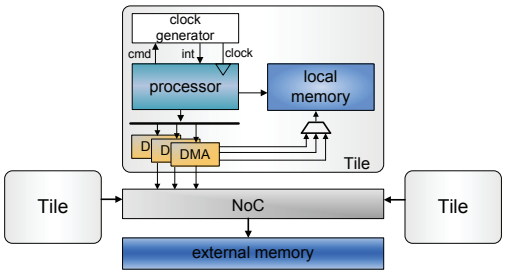

Fig. 1. Composable and predictable platform

Unlike the hypervisors in server domain, we do not support any interrupts except one timer interrupt required for scheduling applications. Virtualization of interrupts is complex and costly; it would increase the OS overhead by the worst-case of the interrupt service routine and its management. In the absence of interrupts, I/O from/to outside of our platform can be memory mapped, and it is supported as long as it does not prevent task preemption (see next section).

4 Composable processor virtualization

As presented in the introduction, composability demands allocating *fixed budgets* for each *application*, in a *fixed period* of time. To achieve composable processor timing, we basically use the same technique as for other resources [6, 10], i.e. *Time*

Division Multiplex scheduling. Though at the first glance composability might seem easy to implement, i.e. preemptable TDM, the required cycle accurate precision combined with the scheduling details make it non-trivial.

Figure 2 presents the view on time at different abstraction levels: application, task, resource (i.e. processor). Each application (consisting of a task set) has a constant time slice allocated each constant TDM period. In an application slice (some of) its tasks execute (in sub-slices). Each application may have a different task scheduler, and the task order inside a slice does not have to be the same each TDM period. Ultimately, the processor time is divided in basic allocable quanta called slots (approach common to many existing OSes). Potentially, in whatever two consecutive slots, two tasks from two different applications can execute. Thus, when having to schedule the next slot, the OS determines the application owning it, and it calls its task scheduler. Hence, tasks execution is interleaved with OS execution. In the following we refer to a slot as *system slot*, composed by an *OS slot*, followed by a *task slot*, as detailed in Figure 3.

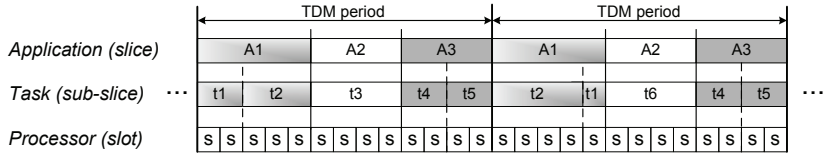

Fig. 2. Time view, at application, task and processor level

Summarizing, composable timing requires: (1) a constant schedule repetition period. This cannot be ensured unless all the system slots have a constant duration (thus the task and OS slices are constant). (2) constant application slices in a TDM period. This cannot be ensured unless the task slots have a constant duration. And (3) application slices in the same order each TDM period. Note, composable capacity only needs the first two requirements to be met.

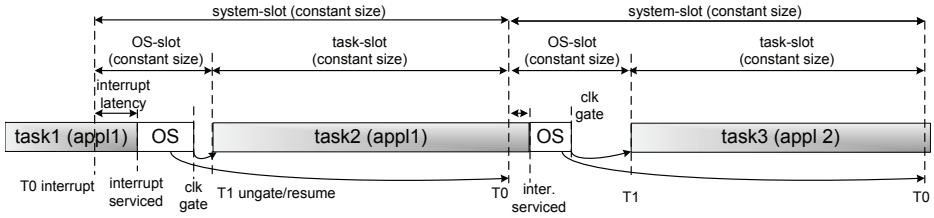

Fig. 3. System slot (OS and task slot)

We 'force' the OS and task slots to have always the same duration, namely their worst case duration, via a mechanism able to delay an action (the task swap in) until a point in the future. This allows us to remove the variation in time of what happened before, as detailed in the next subsection.

Note that composability removes inter-application interference, however an application's execution time might still be variable, due to the following:

- architecture variations
  - undeterministic (though boundable) variations, originating in the conversion from the analog to the digital domain (and vice versa), e.g. clock domain crossing, analog-to-digital converters (ADC) at inputs. Besides

the direct impact in latency (e.g. of memory accesses), it causes arbiters phase drifting, resulting in further variations.
    - deterministic, but difficult to accurately bound; architecture optimizations: out-of-order execution, caches, speculation.
- environment variations: input data dependencies.
- initialization time variations. In embedded systems the workload is known at design-time, thus we initialize the platform once (no dynamic reconfiguration). The initialization time varies with the number of tasks, FIFOs, etc., and results in different application start times, hence in different arbiters phasing from a run to the other (same as the undeterministic variations).

These variations are not dependent on other applications; they occur even when an application executes in isolation.

Clock generation block. The clock generation block (CGB) is the hardware that supports tasks switching at fixed points in time and ensures that system slots have constant duration. We assume each tile is in a different clock domain. Internally, the clock generation block consists of a programmable timer and clock gating functionality. As visible in Figure 1, CGB provides a (programmable) clock signal to the processor, can raise (timer) interrupts, and it can be programmed via the 'cmd' interface. The following can be programmed:

1. *clock* immediate *suspend* and *restart* at a given, future, point in time (similar with clock gating, followed by clock ungating at a programmed moment).
2. *interrupt* generation at a given point in the future.

Figure 3 details the task switching timing. Assume *task 1* is executing on the processor. When *task 1*'s slot ends, CGB raises an interrupt at $T0$. The processor first finishes the in-flight instructions in the pipeline, then the interrupt can be served. We denote the time to finish all in-flight instructions with 'interrupt latency' and its maximum length is typically few cycles. The OS slot starts with the first instruction of the interrupt service routine (a jump to the OS code). Then the OS finds a new task to schedule, as described. Then it programs the next timer interrupt exactly after a (constant) system slot (next *T0*). Then it programs the clock to immediately suspend for the rest of the OS slot, and to restart exactly at $T1 = T0 + OS\_slot$. The OS slot length is a parameter calculated a priori, at design time, and it should be at least the sum of (1) the longest interrupt latency, (2) the worst case number of cycles needed by the OS (for management operations and task scheduling) and (3) the time needed to program the clock gating/ungating and the interrupt for the next slot. By using this mechanism we enforce fixed size tasks and OS slots. The timer mechanism presented above works correctly only if (1) the OS execution time is bounded (we assume so because we allow only application schedulers that finish in predictable time, i.e. the application scheduler has to be analysed, and verified together with our OS) (2) the interrupt latency is bounded (discussed in the following).

Direct Memory Access (DMA) block. A bounded interrupt latency requires interruptible instructions in the processor's pipeline (typically not the case) or the bounded, preferably short, maximum time to finish a in-flight instruction. Thus in general, the maximum delay till serving an interrupt is the maximum time it takes to execute an instruction, which is in the order of magnitude of cycles, except for synchronization operations and load/store operations to remote memories (i.e. other tile's memory or external memory).

We do not support conventional synchronization operations, to exclude the possibility of an interrupt being raised during those. Blocking at FIFOs (if data or space are not available) is implemented by OS checks for data and space in its slot (not in the task slot). If no space or data, a task is simply not started in its current slot (and the slot is idled). The resulted spare capacity can be used to save energy [18], or increase QoS, however this is not the topic of this paper.

When having composable NoC and memory, we can assume that the worst case load/store operation delay is bounded and it can be calculated at design time. However, such a delay can be significantly large, depending on the allocated QoS in the NoC and memory controller. Moreover, for a best effort task with no latency guarantees from the NoC and/or the external memory, a remote access may take an arbitrary time. In order to reduce and bound the interrupt latency we use a DMA block. Hence, instead of a potentially long load/store operation, the processor initiates a DMA transfer between the local and external memories, and pools until the DMA finishes the transfer (operation equivalent with reading a local memory location). The processor is interruptible after each pool operation, thus the interrupt latency is kept short.

To utilize the DMA, the task accesses to remote memories have to be explicit (i.e. FIFO library implemented using DMA, or the DMA requests inserted in the code either by the programmer, or by the compiler). This is a realistic assumption for embedded systems, as they mostly use scratch-pad memories which anyway require explicit data placement and transport. We assume that the local task data and code fits in the local memory (future work will relax on this assumption), thus the remote accesses occur only at FIFO accesses. The DMA requests are transparent to the application programmer, as they are implemented inside FIFO read/write Application Programmer Interface (API).

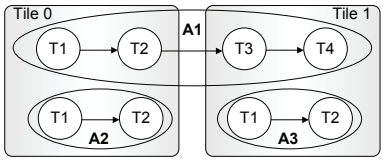

Fig. 4. Experimental applications and mapping

5 Experimental results

We implemented the virtualization mechanism on an FPGA multiprocessor platform. The FPGA platform has two tiles with MicroBlaze cores, local memories, DMAs, and one memory tile (SRAM), connected by an Æthereal NoC [7]. The memories and NoC are (timing) composable as in [6]. The entire platform is predictable (e.g. no caches, no speculation, etc.) and placed in the same clock domain; we have no undeterministic time variations, thus the only variability source in the arbiters alignment/phasing is the variable initialization time.

We use three synthetic applications (Figure 4), with inter-task data dependencies. Each task executes 100 iterations, an iteration being: read input, process, write output, as described in Section 3. We run the experiments with multiple, variable iteration execution times, leading to the same conclusions. For clarity (to be able to visualize even small variations), we present here only the results

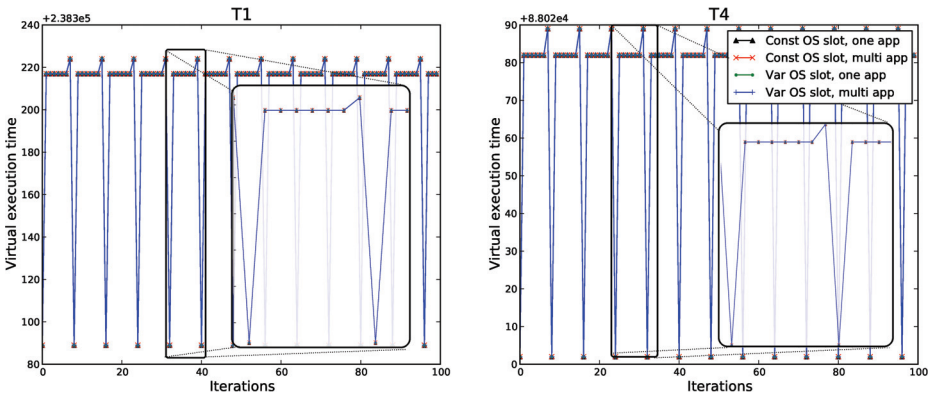

Fig. 5. Virtual execution time - local tasks

for constant iteration execution time (i.e. iterations are not data dependent). We employ *different task schedulers* inside an application, i.e. TDM for A1, and Round Robin for A2 and A3. We focus on A1, that has both possible type of tasks: (1) tasks that execute entirely inside a tile (T1 and T4) - 'local tasks', and (2) tasks that communicate data over the NoC (T2 and T3) - 'non-local tasks'. Non-local tasks are subject to variations from one iterations to the other due to arbiters alignment.

We define: *virtual execution time* as the number of cycles a task executes on a processor, from an iteration's start to end (the perceived execution time in the application virtual time line), and *physical execution time* as the processor cycles elapsed between an iteration's starting and ending times (the perceived execution time in the physical time line, i.e. the response time in RT terminology).

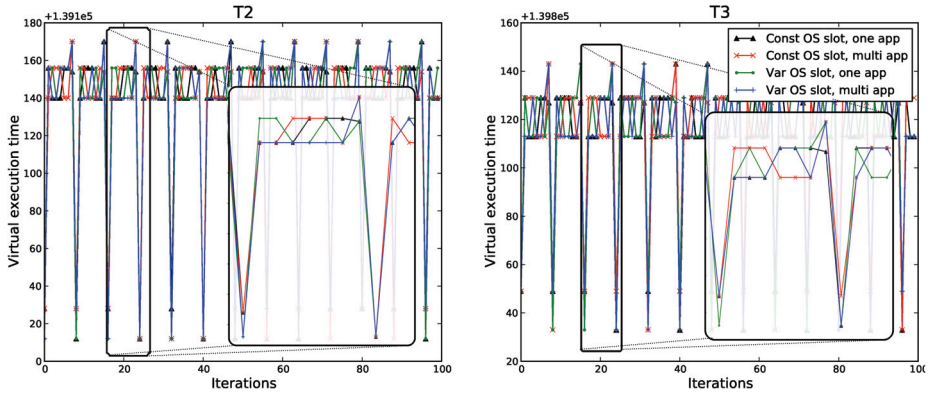

Fig. 6. Virtual execution time - non-local tasks

In all experiments the applications have fixed budgets. We execute four runs:

- A1 executing alone, with a fraction of the resources, fixed/constant OS slots ('Const OS slot, one app').
- A1, A2, A3 together, constant OS slots ('Const OS slot, multi app').
- A1 running alone, with a fraction of the resources, variable OS slots ('Var OS slot, one app').
- A1, A2, A3 running together, variable OS slots ('Var OS slot, multi app').

We investigate: (1) *composable capacity*, i.e. if the virtual execution time (the execution time 'perceived' by the application in its virtual time-line) is the same

in different runs, the capacity is composable, and (2) *composable timing*, i.e. if the physical execution time (the response time in RT terminology) is constant the timing is composable.

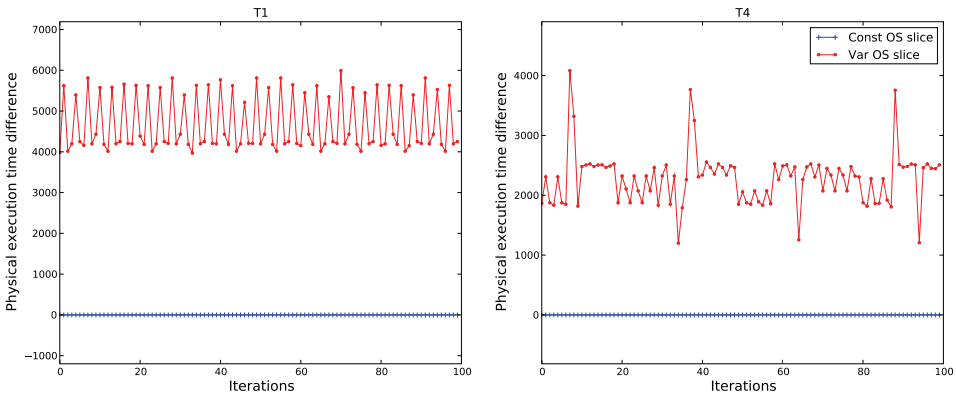

Fig. 7. Physical execution time difference - local tasks

As visible in Figure 5, local tasks exhibit composable capacity: they have exactly the same virtual execution time in all four cases. Note some small variation between iterations are visible; due to FIFO access implementation: reading/writing to the first and last FIFO position take different amount of cycles than accessing a position in the middle of the FIFO. However, these variations are identical for all runs. For non-local tasks (Figure 6), when comparing the four cases, we observe differences in the virtual execution time within 40 cycles. These variations occur due to different NoC and processor schedulers alignment, as argued in Section 4. Hence we can conclude that composable capacity is achieved for both types of tasks.

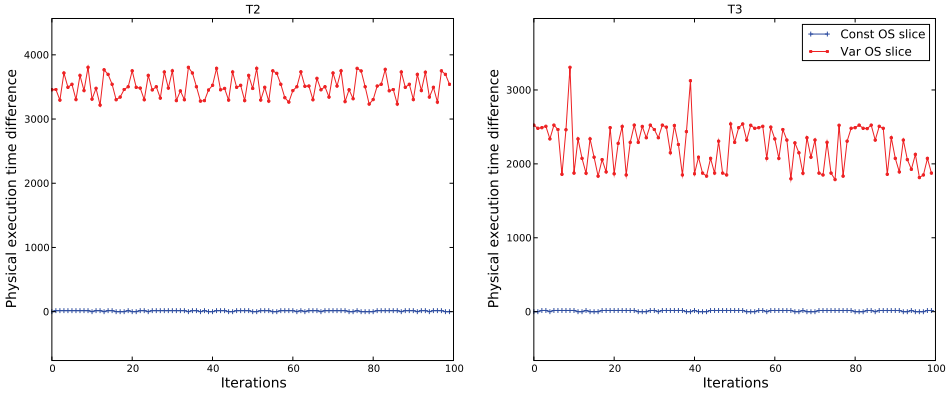

Fig. 8. Physical execution time difference - non-local tasks

When looking at the differences in physical execution time between 'single app' and 'multi app' (Figure 7 and 8), we notice that with constant OS slot ('Const OS slot'), they are very small (*0* for local tasks). For non-local tasks, the variations in virtual execution time propagate to the physical execution time (the difference between 'Const OS slot, single app' and 'Const OS slot, multi app'

are within 40 cycles). These indicate that enforcing constant OS slots induces timing composability. Not the same can be claimed by for variable OS slots. There, adding more applications result in different physical execution time, thus no timing composability. In absolute values, as expected, the physical execution time in the 'Const OS slot' cases is larger than the one in 'Var OS slot' cases, as the OS always takes its worst execution time. However, the OS overhead relative to the task slot size is 8%, and it can be decreased increasing by the task slot.

6 Conclusions

We presented a processor virtualization for embedded systems executing a mix of real-time and non real-time applications. This virtualization technique ensures composable timing and capacity, and allows applications to have their own task scheduler. Here we focused on the processors, and we assumed that all other shared platform resources are virtualized. Composable capacity is achieved by assigning fixed cycle budgets to applications. For composable timing we schedule these budgets equidistantly (preemptable budget scheduler, i.e. TDM using timer interrupts), using a combined hardware software mechanism able to delay an action (the application swap-in moment) until a fixed, future point in time. The OS uses this mechanism to remove the variation in timer interrupt latency and OS behaviour. To ensure a short and bounded interrupt latency, we do not allow a task blocking (the input data and output space availability is checked by the OS before starting a task) and we use DMA accesses instead of long latency load/store operations to remote memories (many processors cannot be interrupted during load/store). The proposed mechanisms are implemented on a multi-processor in FPGA and experiments indicate that applications number and timing of processor cycles are independent.

References

1. VMware Vsphere 4: The CPU scheduler in VMware esx 4. http://pubs.vmware.com.
2. http://www.ibm.com/developerworks/aix/library/au-aix5_cpu/index.html.
3. http://www.virtuallogix.com.
4. H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer, 1997.
5. G. Gössler et al., "Composition for component-based modelling," *LNCS*, vol. 2852/2003, 2004.
6. A. Hansson et al., "CoMPSoC: A template for composable and predictable multi-processor system on chips," *TODAES*, vol. 14, no. 1, 2009.
7. A. Hansson et al., "Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis," *IET CDT*, 2009.
8. C. Paukovits et al., "Concepts of switching in the time-triggered NoC", in *RTCSA*, 2008.
9. B. Akesson et al. "Composable resource sharing based on latency-rate servers," in *DSD*, 2009.
10. M. Bekooij, A. Moonen, and J. van Meerbergen. "Predictable and composable multiprocessor system design: A constructive approach," in Bits&Chips Symposium on ESS, 2007.
11. G. Heiser The role of virtualization in embed. systems. In *IIES*, 2008.
12. A. Nieuwland et al. C-HEAP: A heterogenous multi-proc. architecture template and scalable and flexible protocol for the design of embed. signal proc. systems. *Design Automation for Embedded Syst.*, 7(3), 2002.
13. S. Ruocco. RT programming and L4 microkernels. In *Proc. of the Workshop on OS Platforms for Embedded RT App.*. Citeseer, 2006.
14. M. Steine et al. A Priority-Based Budget Scheduler with Conservative Dataflow Model. *DSD'09*.
15. M.H. Wiggers et al. Monotonicity and RT scheduling. In *Proc. of the 7th ACM intl conf. on Embedded software*, pages 177–186. ACM, 2009.
16. O. Moreira *et al.*, "Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor," in *EMSOFT*, 2007.
17. M.H Wiggers et al. Efficient computation of buffer capacities for cyclo-static RT systems with back-pressure. *Proc. RTAS*, 2007.
18. A. Molnos et al., "Conservative dynamic energy management for real-time dataflow applications mapped on multiple processors," in *DSD*, 2009.