Distributed Resource Management for Concurrent Execution of Multimedia Applications on MPSoC Platforms

Ahsan Shabbir[1] and Akash Kumar[1,2] and Bart Mesman[1] and Henk Corporaal[1]

a.shabbir@tue.nl,akash@nus.edu.sg,b.mesman@tue.nl,h.corporaal@tue.nl
[1] Eindhoven University of Technology Eindhoven The Netherlands, [2] National University of Singapore, Singapore

***Abstract*—The last decade a trend can be observed towards multi-processor Systems-on-Chip (MPSoC) platforms for satisfying the high computational requirements of modern multimedia applications. The research community has mainly focused on communication issues (e.g. bus vs. networks-on-chip). Real-time operating systems for MPSoCs however, have gotten very little attention. Existing techniques like rate-monotonic scheduling from the real-time community are rarely applicable, because contemporary high-performance media processors (like Cell, graphics processors) do not support preemption. Furthermore, rate-monotonic scheduling cannot deal with multiple (heterogeneous) processors, data dependencies, and dynamically varying execution times that characterize modern media applications. This paper proposes new techniques to manage the computational resources of MPSoCs at run-time. We compare a *centralized* resource manager (RM) to two versions (Credit based and Rate based) of a novel, more *distributed* RM. The distributed RMs are developed to cope with a larger number of processors as well as concurrently executing applications. Experiments show that our distributed resource managers are more scalable, deal better with application and user dynamics, and require less buffering, while effectively enforcing throughput constraints.**

I. Introduction

Modern multimedia systems for the consumer market are increasingly based on multi-processors due to stringent performance, power and cost constraints. These MPSoCs typically use (massive scale) instruction-, data- and task-level parallelism to compensate for a lower clock frequency in order to consume less energy while satisfying high compute requirements. Intel projects the availability of 100 billion transistors on a $300mm^2$ die by 2015 [1] which allows to integrate thousands of processors or equivalent gates on a single chip. These MPSoCs will execute multiple applications concurrently that exhibit dynamic behaviour. For example, applications can be started or stopped by the user at run-time. The resource manager will also make trade-offs between the quality levels and compute requirements of the various applications, yielding even more dynamic application behaviour.

Theoretically, compile-time analysis of all possible use-cases (a use-case is a combination of applications active at the same time) can provide performance guarantees, but the potentially large number of use-cases in a real system makes such a static analysis impractical [2], while not dealing with other types of dynamic behavior, like data-dependent execution times (e.g. object-based vision processing). We therefore need to shift the burden from compile time analysis to run-time monitoring and intervention when necessary. This makes run-time resource management an essential part of MPSoCs.

A major requirement for run-time management approaches for these MPSoCs is that they no longer can assume preemption of the hardware platform; The (massively) parallel processing of multimedia data yields such a large amount of state in the processors, that it is no longer cost-effective to implement a context-switch. Even if a context-switch would be implemented, the time required to perform the context-switch can not be ignored, which excludes commonly used real-time scheduling approaches like Rate-Monotonic Scheduling, and other fixed-priority schedulers. The latest high-performance media MPSoCs for the consumer market, like the Cell Broad Band Engine (BBE) [3] and graphics processors, do not support preemption, thereby acknowledging the necessity of our non-preemptive assumption.

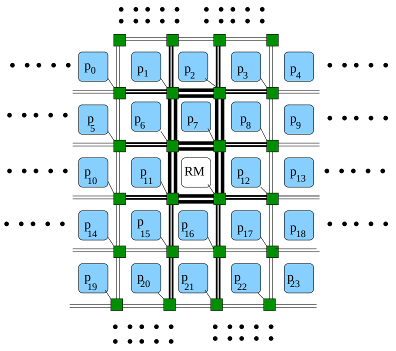


Fig. 1. Management bottleneck for centralized RM

A resource manager has been presented in literature [2] that performs management in a *central* way. It monitors the progress of applications, and enables and disables the applications at the smallest possible grain, that of an actor (explained in more detail in Section III). In this way, the central resource manager helps applications satisfy their throughput constraints. However, the central RM is not scalable with the number of applications and processors.

The centralized RM creates a hot spot in terms of management as shown in Figure 1. All the information regarding application progress and QoS is fed to the centralized manager. The centralized manager has to process the information in short time and take corrective measures. The compute requirement from the applications overwhelms the centralized manager.

In this paper, we propose two versions of resource managers which are scalable with number of applications and processors. Our resource managers ensure admission control- an application is only allowed to start if the platform can allocate the resources demanded by the application. Our distributed RMs are based on budget based schedulers and differ in their budget enforcement protocols. The first type of RM- *Credit Based* can be used for applications which have strict constraints on their performance i.e. their performance can not be more than a fixed level even if resources are available to have better performance. Our second type of RM- *Rate based* is suitable for applications which may have more performance than a minimum level if the compute resources are available. For example, streaming encoders are good target for these type of RMs so that if there are resources available in the MPSoC platform, they can encode at a higher rate and finish the job quickly.

In this paper, the distributed resource managers have been evaluated on the basis of their ability to satisfy the throughput constraints of multiple applications. We have also performed experiments by adding applications at run-time and studying their behaviour. The evaluation metrics used in this paper include deadline misses, buffer requirement, and maximum jitter. A deadline miss occurs when an application fails to meet its deadline. The difference in successive finish times of an application iteration should be fixed. Due to interference with other applications, the difference may not be constant. This variation in successive finish times is defined as Jitter. Other aspects like processor utilization, and run-time variation in application throughput constraint have also been studied in this paper.

Overview: The following section describes related work. In section III, our application and architecture models are presented. Section IV presents an example showing the scalability issues with centralized RM. In Section V, we present our credit based RM and rate based RM. The evaluation and comparison of these RMs is presented in Section VI and Section VII concludes this paper.

II. Related Work

Research on multi-processor real-time scheduling has mainly focused on pre-emptive systems [4], [5]. Non-preemptive scheduling has received considerably less attention. It was shown by Jeffay et al. [6] and further strengthened by Cai and Kong that the problem of determining whether a given periodic task system is non-preemptively feasible even upon a single processor is already intractable [7]. Recently, more work has been done on non-preemptive scheduling for multiprocessors. S. Baruah [8] presented sufficient conditions for a periodic task system to be feasible on multi-processor platform. In short, non-preemptive scheduling of periodic and non-periodic tasks on multiprocessor systems is NP-hard. There are many heuristic based solutions to this problem.

There are a number of budget schedulers [9], [10], [11] in the literature. These schedulers assign a fixed time for each task in a replenishment interval. Steine [10] has added priorities on top of the budgets (PBS). The priorities of all tasks have to be defined at design time and one task can have its priority defined during run-time. Their technique can be used at design time whereas our technique is for run-time resource management. The work by [11] is for single processor scheduling.

TDMA is also considered as budget based scheduler. Computing worst-case waiting time taking resource contention into account for round-robin [12] and TDMA [13] (requires preemption) scheduling has also been analyzed. However potential disadvantage of these approaches is that the analysis can be very pessimistic. In [14], internal and external contention in communication streams is considered, but their region forming approach is targeted at homogeneous meshed platforms, and is not suitable for heterogeneous or irregular architectures. In [9], an architecture driven approach is used to map tasks first on virtual tiles, which are in turn clustered on elements connected to the same router. They use TDMA schedulers at the processors for budget enforcement. Task switching in TDMA is preemptive while our RMs use non-preemptive scheduling. The distributed approach of [15] uses a static mapping algorithm inside its clusters. This approach requires hardware support for cluster management, while it poses more constraints on the size and structure of applications.

In [2], a resource manager has been proposed which shifts the burden from compile time analysis to run-time monitoring and intervention. They advocate the fact that compile-time analysis of all possible use-cases can provide performance guarantees, but the potentially large number of use-cases in a real system makes such analysis infeasible [2]. However their resource manager is centralized and not scalable. We have shown this with the help of experiments in the next section. We study the factors affecting the scalability of resource manager and propose two distributed resource managers aimed to be be more scalable with increasing number of applications and processing elements.

StarSs [16] is a programming model and run-time manager from Barcelona Super Computing Center. It is an extension of OpenMP [17]. It consists of a "C-C" compiler which converts a sequential C-code into a C-code that can execute on PPE and SPEs of Cell processor. The run-time manager executes on the PPE and distributes the jobs between the SPEs whenever it finds a free SPE. The run-time manager is centralized and communicates with the SPEs using the mailboxes. Nanos++ [18] is another centralized resource manager that consists of a CPU manager and a scheduler. The CPU manager acts as an admission controller and all applications send requests to the CPU manager. The CPU manager selects a scheduling policy and computes the required number of processors for each application and assigns them. The CPU manager monitors the performance of applications through file systems of the processors. Both StarSs and Nanos++ monitor the performance of applications centrally, in contrast to the resource managers presented in this paper; they monitor the performance of the tasks of applications locally to increase the scalability of distributed RMs.

III. Application and architecture modeling

In this paper, we assume applications to be specified as Synchronous Data Flow SDFGs [19], where vertices indicate separate tasks (also called actors) of an application, and edges denote dependencies between them. SDF is widely used; it is very suitable to express concurrency in applications, and is therefore useful to analyze multi-processor systems.

The architecture model consists of processors connected with each other through point-to-point/NoC interconnects. Our resource managers consist of an admission controller. The role of the admission controller is to evaluate the timing constraints of the new applications against available resources. If the available resources of the platform are less than the timing requirement of the new application, the admission controller rejects the request and the application can request service at a lower quality level. A similar admission controller has been presented in [2]. To find the budgets, following information is required by the admission controller.

- SDF model of each application.

- Worst-case execution time estimates for each task (in clock cycles).
- Desired performance of each application e.g. frames/sec etc.
- Mapping of tasks onto the platform is provided. Task migration is not supported.
- Buffer sizes needed for edges in the graphs.
- Performance prediction of each application in isolation with the given mapping. This can be achieved using SDF^3 [20].

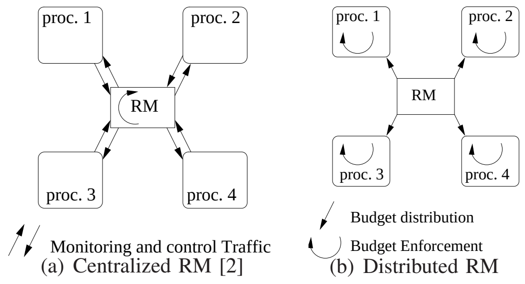


Fig. 2. Resource Management on MPSoC: Centralized vs Distributed

Figure 2 shows the functional diagram of the resource manager presented by [2] and our distributed RM. The RM presented by [2] monitors the throughput of each application and compares it with its desired throughput. The centralized RM enables an application performing less than its desired throughput. Similarly the application having more than its desired throughput is suspended. The monitoring and control overhead limits the scalability of the centralized RM as it has to monitor and control all applications and perform QoS negotiations as well. To solve this problem, we present two versions of distributed RMs. These RMs seek to minimize the involvement of the central manager in the process and invest more intelligence in the local processor arbiters as shown in Figure 2(b). The budgets for each application are calculated centrally but they are enforced on all the processors locally. The RM does not monitor each application so the scalability problem of [2] due to monitoring period is eliminated. In the next section, we present an example to further elaborate this problem.

IV. Motivating Example

The central resource manager monitors the performance of each application. The monitoring period of central RM should be less than or equal to smallest period amongst all applications being monitored otherwise it will not be able to monitor the variations in that application. To study the scalability of central RM with increase in number of applications we use the model of central resource manager as described in [2]. The computation platform has 10 processors. The central resource manager performs the following operations for monitoring the performance of the applications:

- Receive the iteration completion message from each application.
- Increment the total execution count of the application.
- Find the current period of the application.
- Compare the current period with the desired period.
- Send enable/suspend signal to the application according to the result of the comparison.

We implemented the above functionality on a Microblaze processor [21] and measured the clock cycles required for one application. It took 240 clock cycles per application to perform above mentioned functions. We modeled our RMs with timing information. The length of a monitoring period determines the number of applications which can be controlled by the resource manager as the central processor must complete the above mentioned functionality within the monitoring period. Assume we have a monitoring period of 7,000 clock cycles then we can monitor 7000/240 = 29 applications only. If we increase the number of applications to 30, the applications cannot meet their constraints as shown in Figure 3. In this experiment, we increased the number of applications and kept the monitoring period fixed at 7,000 clock cycles. The total number of deadline misses of these applications remain relatively low until 25 applications. For 30 applications, the total number of deadline misses increase tremendously as shown in Figure 3. The reason for this increase is the fact that we cannot monitor all 30 applications in the period of 7,000 cycles so every cycle some applications are not monitored, resulting in deadline misses.

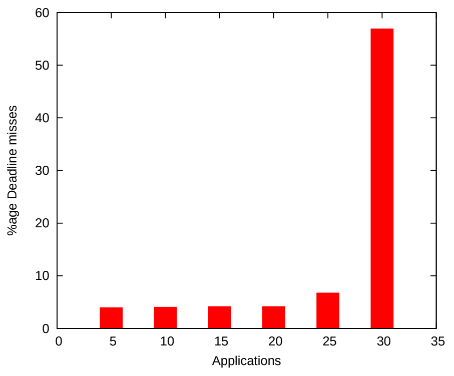


Fig. 3. Increase in deadline misses with increase in number of applications

The other problem with centralized RM is jitter in application execution. If the monitoring period is large then an application will remain enable/disable during the whole monitoring period resulting in periods where the performance of application is more than desired performance and periods where it is less than the desired performace. To handle this jittery behaviour, large buffers are required to store the outputs of the applications so that the average behaviour is acceptable. These large buffers increase the cost of the system and are highly undesirable. We illustrate this effect in section VI-E.

Note that the scalability problem is made even worse when Quality-of-Service (QoS) negotiations are performed in order to deal with scarcity of computational resources. These negotiations burden the resource manager for each application (to set the quality level) as well each processor (monitoring and quality settings). We therefore expect that future MPSoC systems that enable QoS will profit even more from our distributed resource manager, which are optimized for scalability.

We propose two versions of the distributed resource managers that are motivated to address the scalability issues. The first type of resource manager is called *Credit based resource manager*. Here the central admission controller distributes credits among the processors and they enforce these credits. This type of manager is useful where the throughput of applications has to be kept at a certain level. The second distributed RM is called *Rate based resource manager*. This RM uses the same admission controller as used by the credit based manager and differs in the budget enforcement mechanism. This type of manager is useful for those applications which can have more than a certain level of performance.

V. Proposed Resource Managers

In this section, we first describe the admission control and credit calculation mechanism. The same mechanism is used in both type of RMs. In both versions of distributed resource managers, there is a central admission controller connected with the processing cores through a NoC. The central admission controller is an interface to the user and calculates the credits and these credits are distributed to the processors. The arbiters at the processors locally enforce these credits such that the throughput constraints of the applications are satisfied.

Algorithm 1 shows the method of calculating the credits. Each processor has a large replenishment interval of time with in which all tasks have to execute. The central controller finds the processing load imposed by each task on each processor. This load is calculated by multiplying the repetition vector of each actor with its execution time and the ratio of desired to predicted throughput. The size of the replenishment interval should be greater than the total processing load. This process is repeated for all the processors and an application is only admitted if all processors satisfy this condition. The credits

Algorithm 1 Admission Control and Credit calculation

```
1: for all processors do
2:   load(processor)=0;
3:   for all applications do
4:     processing_load=0;
5:     for all actors $\in$ application do
6:       if (mapping(application,actor) == processor) then
7:         processing_load(actor) = $\gamma$(actor) $\times$ $\alpha$(actor) $\times$ desired_thr.
8:         load(processor)+=processing_load(actor)
9:         if (load(processor)/size_of_replenishment_interval(processor)>1) then
10:          remove_load(application,processor){admission not possible, remove all actors of the application from all processors}
11:        end if
12:      end if
13:    end for
14:  end for
15: end for
16: for all applications do
17:   for all actors(application) do
18:     credits(actor)=$\gamma$(actor) $\times$ desired_thr.
19:     send(credits,actor,processor)
20:   end for
21: end for
```

are calculated as shown in line number 18 in Algorithm 1. Here γ(actor) is the repetition vector entry of the actor and $desired\_thr$ is the desired throughput constraint of the application. We explain the credit calculation mechanism with the help of an example. Assume that we want to find the credits for the inverse quantization actor (IQ) from JPEG decoder. The JPEG decoder graph used for this example decodes one macro-block in one iteration. The IQ actor has to be called 6 times (4 times for luminance data and 2 times for chrominance). If the desired throughput constraint for JPEG decoder is set at 1 QCIF picture/sec then it is equivalent to 99 macro-blocks of JPEG encoded data and the number of credits for IQ are 6×99 = 594. Assuming a replenishment interval of one second, the processing load imposed by IQ is $6\times2400\times99$ = 1,425,600 clock cycles. Here, we assume that 2400 clock cycles are required by the IQ actor to perform inverse quantization function on one macro-block.

A. Credit Based Resource Manager

The central admission controller sends the credits to the processors according to the mappings of actors onto processors. To enforce these credits, each processor has a kernel which loads these credits into counters. Each actor is repeated the number of times as specified in its counter in one replenishment interval. After completion of the interval, the counters are reloaded with their values as received by the central controller and this process continues. During the execution, if an actor is not ready then it is skipped and the processor is assigned to other actor so that the processor time can be used more efficiently.

Note that in contrast to Time Division Multiple Access (TDMA), the replenishment interval of credit based RM is not necessarily always equal to maximum replenishment interval. It might be possible that some applications are stopped by the user so the length of that replenishment interval will be smaller than the maximum replenishment interval.

Algorithm 2 Credit based RM

```
1: Relod_Credits()
2: while (size_of_replenishment_interval(processor)>0) do
3:   if ($actor == ready \wedge (credits(actor) > 0)$) then
4:     execute(actor)
5:     credits(actor)=credits(actor)-1
6:   end if
7:   actor=next_actor_in_list
8: end while
```

B. Rate Based Resource Manager

The dependence of the credit based RM on the size of replenishment interval can be removed by the Rate based RM. In the rate based RM each processor has a local arbiter. The admission controller of the rate based RM calculates the credits in the same way as the credit based RM. The admission controller sends these credits to each distributed arbiter located at each processor. This arbiter receives these credits and executes these actors in such a way that the actor having the least achieved-to-desired execution ratio, is given the highest priority. Note that the rate based RM allows the applications to execute continuously so long as the ratio is maintained. This means that the applications can have more throughput than the desired throughput. To implement such an arbiter, a data structure for each actor is defined which contains the information required during the operation of the arbiter. This data structure contains registers to store the desired rates Rd_{a_i} (credits), achieved rates Ra_{a_i} and ratio of achieved-to-desired rates Rr_{a_i} for each actor a_i. The ratios of the achieved-to-desired rates are stored in non-decreasing order. Each time an actor executes, its achieved execution rate is incremented by one. Each time a processor needs to execute an actor it

Algorithm 3 Rate based distributed Manager

```
1: for all actors $a_i$ do
2:   Receive_rates(){Receive rates from admission controller}
3:   init($Rd_{a_i}$, $Ra_{a_i}$, $Rr_{a_i}$)
4: end for
5: loop
6:   for all actors $a_i$ do
7:     if ($a_i$ == ready && $Rr_{a_i}$ == min_rate) then
8:       Execute the actor
9:       $Ra_{a_i}$++
10:      $Rr_{a_i}$ = $Ra_{a_i}$ / $Rd_{a_i}$
11:      min_rate = find_min_rate()
12:    end if
13:  end for
14: end loop
```

finds the actor having least ratio and executes that actor if it is *ready* as shown in line 11 of Algorithm 3. In SDF semantics, an actor is ready when its input data is available and there is space in its output buffer.

We explain the mechanism of rate based RM by an example. Assume two applications a JPEG decoder and an H.263 decoder are to be executed on the platform. We assume that two actors, inverse quantization actor (IQ) from JPEG, and IDCT actor from H.263 decoder, are mapped onto a single processor. The constraint for JPEG decoder is 1 QCIF frame/sec as described in the previous example. The H.263 decoder has a performance constraint of 20 frames/sec then the credit for IDCT actor is 40. Assume that both actors have been executed twice then the achieved-to-desired ratios of IQ and IDCT are 2/594=0.003 and 2/40=0.05 respectively. The achieved-to-desired ratio of IQ is lower than that of IDCT so the IQ actor has the higher priority to use the processor as compared to IDCT. The platform will execute IQ actor and will update the achieved execution count of IQ (line 9-10 of Algorithm 3) and will calculate the priorities again.

Note that both RMs are very simple so hw/sw implementation is not expensive in terms of area/run-time overhead. This makes them suitable for run-time resource management of embedded systems.

VI. Comparison between the Resource Managers

In this section, we compare the centralized and distributed resource managers. The basis of comparison are different kinds of scenarios which are possible during run-time. These scenarios include:

- run-time admission of a new application
- run-time stoppage of an application
- variation in execution time of the actors
- run-time variation in application throughput constraint

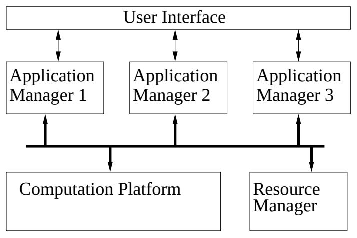


Fig. 4. Overview of the system setup

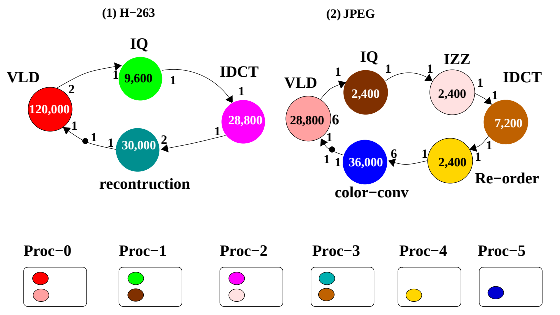


Fig. 5. Application graphs and their mapping onto Computation platform

Figure 4 shows the overview of the experimental setup. It consists of a user interface, local application managers (one for each application), a resource manager, and the computation platform. The user-interface simulates input from and output to the user; for example, in case of mobile phone, input can come from a keypad, while the output can be in the form of screen display or sound. The simulation models of our distributed resource managers have been developed using the modeling language POOSL [22]. POOSL is a very expressive modeling language with a set of powerful primitives and completely formally defined semantics.

We have used two application models: JPEG decoder and H.263 decoder, as shown in Figure 5. The JPEG decoder consists of 6 actors namely, Variable Length Decoding (VLD), Inverse Quantization (IQ), Inverse Zigzag (IZZ), Inverse Discrete Cosine Transform (IDCT), Reorder (Re-order) and Color Conversion (CC). The H.263 decoder has four actors namely Motion Compensation (MC), IDCT, IQ and VLD. The computation platform consists of 6 processors and the figure also shows the mapping of actors onto processors. In the experiments, the monitoring period for centralized RM is 40 million clock cycles. The processors in credit based RM and rate based RM are executing at 40 MHZ. The replenishment interval for credit based RM is 1 second so that the RMs are matched. We also compare the buffer requirement and processor utilization of these RMs.

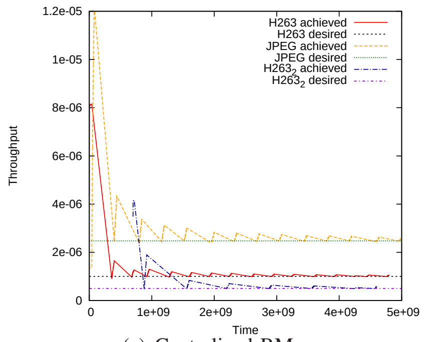
A. Admission of New Application

In this experiment, a new application enters the MPSoC platform and requests for resources at (simulation) time 700 million clock cycles. The applications already executing on the platform are a JPEG decoder at 1 QCIF/sec and a H.263 decoder at 40 frames/sec. Another instance of H.263 decoder enters the platform and requires to run at a throughput constraint of 20 frames/sec. Figure 6(a) shows the behaviour of centralized RM against this dynamic situation. The applications already executing do not show any impact on their performance. The high jitter in the application execution is due to the fact that the applications remain enable/disable across the monitoring period. The throughput of applications gets more than the desired throughput in monitoring period where they remain enabled. Similarly the application throughput decreases in the monitoring period where they remain disabled.

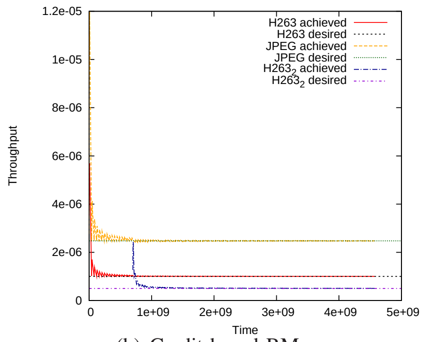
Our admission controller evaluates the resources needed for the application according to Algorithm 1 and allows it to execute. Figure 6(b) shows robustness of our credit based RM. The applications already executing on the platform do not have any adverse effect on their performance due to virtualization of the resources. Moreover, the jitter in application execution is very small as compared to the centralized RM. Figure 6(c) shows the response of the rate based RM. The performance of JPEG and H.263 decoders decrease immediately as soon as the second instance of H.263 decoder is accepted by the RM. This is because of the fact that when the second instance of H.263 decoder enters, it has the lowest achieved to desired ratio so it gets preference and quickly gains the required performance level. As the time goes by other applications also get the compute resources and the system very quickly converges towards the new steady state.

B. Application stopped by the user

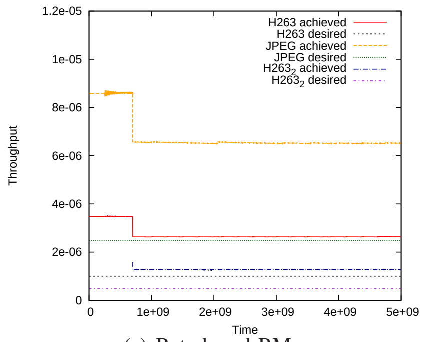
This experiment starts with three applications executing on the platform. The JPEG decoder is executing at 1 QCIF/sec and two H.263 decoders are executing at 40 and 20 frames/sec respectively. The JPEG decoder is stopped by the user at simulation time of 700 million clock cycles.



(a) Centralized RM

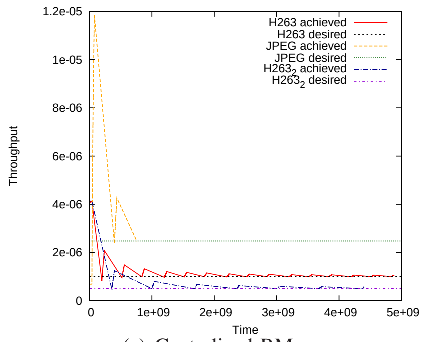


(b) Credit based RM

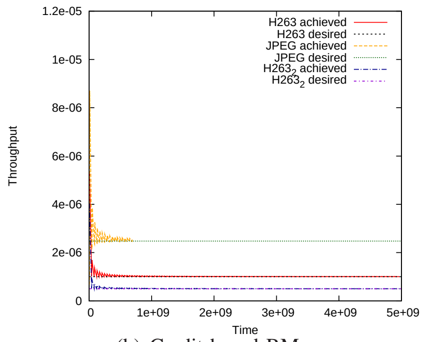


(c) Rate based RM

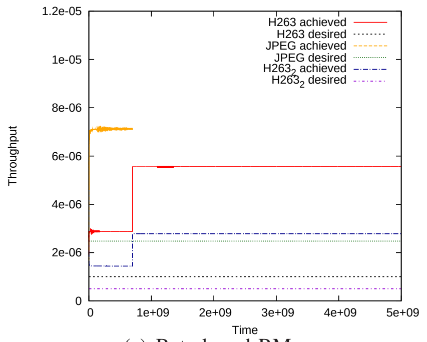
Fig. 6. Example of run-time application admission at simulation time of 700 million clock cycles



(a) Centralized RM



(b) Credit based RM



(c) Rate based RM

Fig. 7. JPEG application stopped by the user at 700 million clock cycles

The behaviour of centralized RM under this dynamic condition is shown in Figure 7(a). The centralized RM can handle this situation and there is no effect on other applications because of virtualization of resources. Figure 7(b) shows the behaviour of our credit based RM. There is no effect on other applications as our credit based distributed resource manager provides virtualized platform for each application. The applications execute concurrently but do not interfere with others for the resources.

Figure 7(c) depicts the response of our rate based RM in the event of application stoppage. After the JPEG decoder has stopped, the second instance of H.263 gets the processor more often as its ratio is the lowest. The system goes into steady state and both H.263 decoders share the resources freed by the JPEG decoder. The performance of one of H.263 decoder is twice of the second instance of H.263 decoder and both are above their specified throughput constraints.

C. *Variation in Actor Execution Times*

In this experiment, the execution time of the actors is varied randomly between 1 clock cycle to the actual execution time of the actor. We use uniform random number generation for this experiment. Figure 8(a) shows that the variation in the execution time has no effect on the throughput for centralized RM. Similarly our credit based RM has no effect on the throughput of the applications as shown in Figure 8(b). The reason for this result is the fact that the credits in our distributed resource managers are calculated as the number of iterations each actor has to execute for satisfying the application throughput constraints. This property makes it independent of the variations in the actor execution times.

However, the rate based distributed RM is slightly effected by the variation in actor execution times. The variation in execution time affects the achieved to expected ratios and consequently the throughput of applications observe some variation but the magnitude of this variation is pretty small as shown in Figure 8(c).

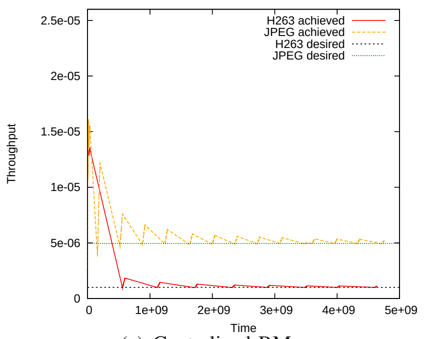
D. *Variation in Application Throughput Constraint*

In this experiment, the H.263 decoder is required to decode 40 frames/sec and the constraint for the JPEG encoder is 2 QCIF frames/sec. The distributed credit based resource manager finds the credits based on this information. The arbiters in the processors enforce these credits as shown in Figure 9(b). Now at 700 million clock cycles, the distributed resource manager receives a request from the user to decrease the frame rate of H.263 decoder from 40 frames/second to 20 frames/second. This implies that the distributed resource manager has to re-calculate the credits based on the new application constraints and it re-sends them to the processors. Figure 9(b) shows that the new credits are enforced by the processors and the new throughput constraint is met successfully. Further, there is no effect on the throughput of the other application.

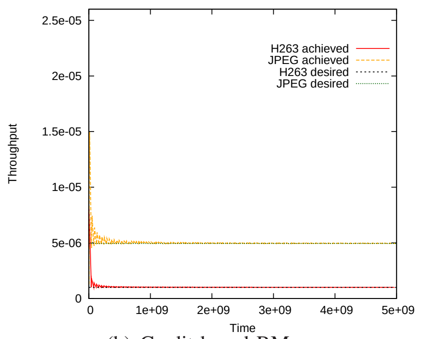
For the same experiment with the centralized RM, there is performance degradation for the JPEG decoder. The reason for this degradation is the monitoring period of centralized decoder as shown in Figure 9(a). It is clear that the response of centralized RM is slow as compared to that of credit based RM. Figure 9(c) shows the same experiment with rate based RM. The resources freed by the second instance of H.263 are consumed by the H.263 decoder executing at higher rate and the system achieves steady state quite quickly.

E. *Buffer Requirement*

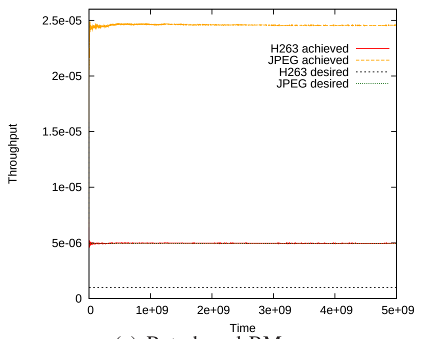
The throughput constraints for this experiment have been assumed to be 2 QCIF frames/sec for JPEG decoder and



(a) Centralized RM

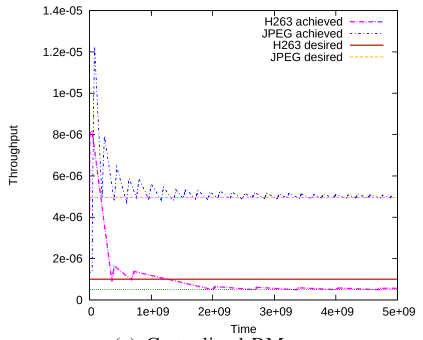


(b) Credit based RM

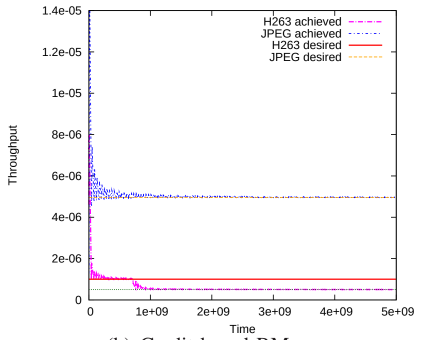


(c) Rate based RM

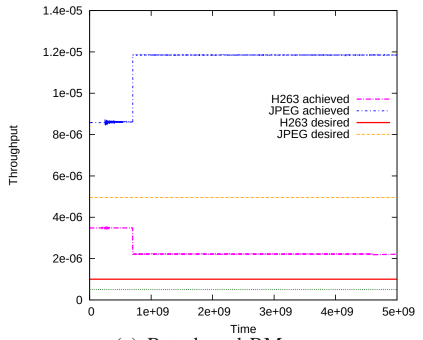
Fig. 8. Variation in actor execution times



(a) Centralized RM

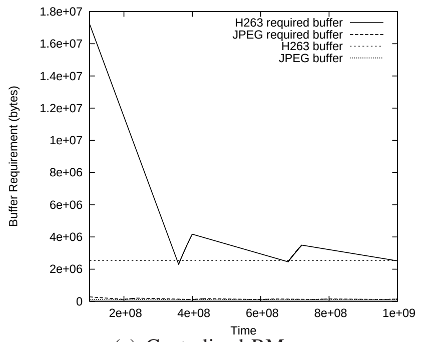


(b) Credit based RM

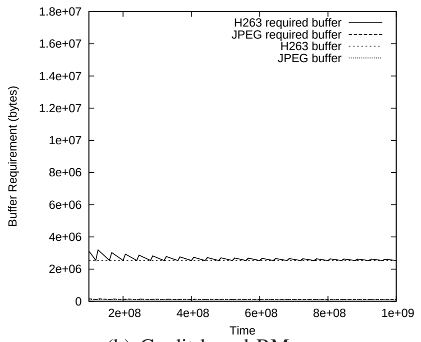


(c) Rate based RM

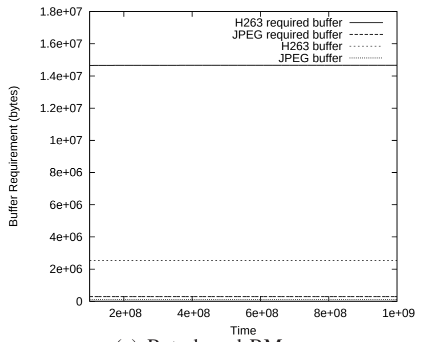
Fig. 9. Run-time change in application constraints of H.263 decoder



(a) Centralized RM



(b) Credit based RM



(c) Rate based RM

Fig. 10. Comparison of buffer requirement

40 frames/sec for H.263 decoder. This translates to 4.95 $\times 10^{-06}$ iterations/clock cycle for JPEG and 1.0 $\times 10^{-06}$ iterations/clock cycle for H.263 decoder. In this experiment, we study the jitter in application execution introduced by the RMs. The jitter results in large buffers at the output of the applications to store the frames/macro-blocks decoded by the applications. The desired buffer space shown in the Figure 10 is ideal buffer space required assuming no jitter in application execution.

Figure 10(a) shows that the control is not as smooth and the platform needs higher buffer space because when an application achieves more throughput than the desired throughput then the frames decoded are to be stored in a buffer memory. The jitter in the application execution is introduced due to monitoring period. Figure 10(a) shows the buffer requirement of both applications. The JPEG decoder has to decode 198 macro-blocks in one second e.g. 2 QCIF frames/sec. H.263 decoder has to decode 99$\times$40 macro-blocks/sec e.g. 40 frames/second. This is equal to almost 3 Mbytes of buffer space. The extra buffer space needed for centralized RM is quite large as compared to ideal buffer space requirement.

Figure 10(b) shows the buffer requirement for credit based RM. The actors from the applications are executed according to their credits and this process repeats each second. It is evident that our credit based RM requires very small amount of extra buffer space as compared to the ideal buffer space. The

TABLE I
COMPARISON OF JITTER (IN CLOCK CYCLES) BETWEEN THREE RMS

App.	Centralized based		Credit based		Rate based	
	Avg. jitter	max. jitter	Avg. jitter	max. jitter	Avg. jitter	max. jitter
JPEG	256,528	1.59e^8	43,200	123,600	355	93,600
H.263	281,766	2.79e^8	172,800	1,983,003	730	172,800

buffer requirement for rate based RM is more than credit based RM and centralized RM as shown in Figure 10(c). The high

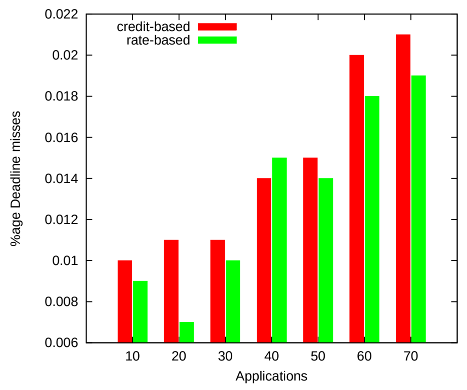


Fig. 11. Scalability of RMs with number of applications and processors

buffer requirement is due to the fact that in the rate based RM, there is no replenishment interval so the applications execute continuously. This leads to the case where they can have more than a minimum desired throughput. In the credit based RM, when credits are exhausted, the applications are stopped and re-enabled during the next replenishment interval. The rate based RM tries to execute the applications according to their achieved-to-desired ratios. The rate based RM can be suitable for applications which can tolerate more than specified levels of throughput. For example, encoders can benefit from such RMs so that they can encode at a faster rate whenever there is enough resource in the platform.

Table I compares the jitter (clock cycles) in application execution for three type of RMs. The maximum jitter for centralized RM is the highest. This is due to the fact that the applications in centralized RM can be executing or disabled for longer periods of time. Hence the time difference between two successive executions of applications can be very large. In Credit based RM, the maximum jitter is smaller than the centralized RM. This results in low buffer requirement. The maximum jitter for Rate based is the lowest. The average jitter of distributed RMs is also lower than the centralized RM. Low jitter is also an evidence of low buffer requirement.

F. Processor Utilization of RMs

The processor utilization of three RMs is shown in Table II. The applications and their constraints are the same as used in the experiment of subsection VI-E. The processor utilization is the ratio of time spent on the applications to the total processor time. Table II shows that the average processor utilization of the Rate based RM is highest of all RMs. This is due to the fact that the applications execute continuously and try to use the compute resources to the maximum.

TABLE II
PROCESSOR UTILIZATION OF RMS.

Centralized	Credit-Based	Rate based
0.1672	0.1625	0.8074

G. Scalability of RMs

In this experiment, the scalability of our RMs is evaluated with increasing number of processors and applications. Two platforms models consisting of 10 and 20 processors are built. The platform with 10 processors is used to simulate up to 30 applications and from 40-70 applications, the platform with 20 processors is used. Figure 11 shows that both RMs are scalable with number of applications and processors. The figure also shows that the deadline miss rate of Rate based RM is lower than Credit based RM.

VII. CONCLUSIONS

We have presented two versions of a distributed resource manager (RM) for multi-processor Systems-on-Chip, and compared them to a centralized resource manager. Experiments show that the credit based RM is very effective for enforcing throughput constraints, and the rate based RM is effective for obtaining a high resource utilization in the context of applications that can profit from the availability of more resources. Both distributed RMs can cope with a larger number of processors as well as large number of concurrently executing applications compared to a centralized RM. Furthermore, our experiments demonstrate that they deal better with application and user dynamics, and require less buffering. We conclude that our approach is effective for controlling the computational resources in a multi-processor platform, can deal with data dependencies and dynamically varying execution times that characterize modern media applications, without requiring support for preemption. We can therefore fill the gap left by existing techniques like rate-monotonic scheduling that cannot satisfy the abovementioned requirements.

REFERENCES

[1] S. Borkar, "Thousand core chips: a technology perspective," in *DAC '07*. New York, NY, USA: ACM, 2007, pp. 746–749.
[2] A. Kumar, *et al.*, "Analyzing composability of applications on mpsoc platforms," *J. Syst. Archit.*, vol. 54, no. 3-4, pp. 369–383, 2008.
[3] C. H. Crawford, *et al.*, "Accelerating computing with the cell broadband engine processor," in *5th CCF:*. New York, NY, USA: ACM, 2008, pp. 3–12.
[4] S. Davari and S. K. Dhall, "An on line algorithm for real-time tasks allocation," in *Real Time Systems Symposium*, 1986.
[5] S. K. Baruah, *et al.*, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.
[6] K. Jeffay and D. F. Stanat, "On non-preemptive scheduling of periodic and sporadic tasks," in *12th RTSS*, 1991, pp. 129–139.
[7] Y. Cai and M. C. Kong, "Nonpreemptive scheduling of periodic tasks in uni- and multiprocessor systems," *Algorithmica*, vol. 15, no. 6, pp. 572–599, 1996.
[8] S. K. Baruah, "The non-preemptive scheduling of periodic tasks upon multiprocessors," *Real-Time Syst.*, vol. 32, no. 1-2, pp. 9–20, 2006.
[9] O. Moreira, *et al.*, "Online resource management in a multiprocessor with a network-on-chip," in *SAC '07:*. New York, NY, USA: ACM, 2007, pp. 1557–1564.
[10] M. Steine, *et al.*, "A priority-based budget scheduler with conservative dataflow model," in *12th DSD*, 2009, pp. 37–44.
[11] C. W. Mercer, *et al.*, "Processor capacity reserves: Operating system support for multimedia applications," in *International Conference on Multimedia Computing and Systems*, 1994, pp. 90–99.
[12] R. Hoes, "Predictable dynamic behaviour in noc-based multiprocessor system-on-chip," Master's thesis, Eindhoven University of Technology, Eindhoven (The Netherlands), 2005.
[13] S. Stuijk, *et al.*, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," in *DAC '07:*. New York, NY, USA: ACM, 2007, pp. 777–782.
[14] C.-L. Chou and R. Marculescu, "User-aware dynamic task allocation in networks-on-chip," in *DATE '08:*. New York, NY, USA: ACM, 2008, pp. 1232–1237.
[15] M. A. Al Faruque, *et al.*, "Adam: run-time agent-based distributed application mapping for on-chip communication," in *DAC '08*. New York, NY, USA: ACM, 2008, pp. 760–765.
[16] P. Bellens, *et al.*, "Cellss: a programming model for the cell be architecture," in *ACM/IEEE Conference on Supercomputing*. ACM, 2006, p. 86.
[17] R. Chandra, *et al.*, *Parallel programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.
[18] E. Ayguade, *et al.*, "A proposal to extend the openmp tasking model for heterogeneous architectures," in *IWOMP*. Springer-Verlag, 2009, pp. 154–167.
[19] E. Lee and D. G. Messerschmitt, "Static scheduling of synchronous dataflow programs for digital signal processing," in *Proc. of IEEE Transactions on Computers*, vol. 36. IEEE, jan 1987, pp. 24–35.
[20] S. Stuijk, *et al.*, "SDF^3: SDF for free," in *ACSD*, 2006, pp. 276–278.
[21] Xilinx, *Microblaze Processor Reference Guide*, 2004.
[22] P. H. A. van der Putten and J. P. M. Voeten, "Specification of reactive hardware/software systems," Ph.D. dissertation, Eindhoven University of Technology, 1997.