Analyzing Composability of Applications on MPSoC Platforms

Akash Kumar, Bart Mesman, Bart Theelen, Henk Corporaal
Eindhoven University of Technology
5600MB Eindhoven, The Netherlands
Email: {a.kumar,b.mesman,b.d.theelen,h.corporaal}@tue.nl

Ha Yajun
National University of Singapore
10 Kent Ridge Crescent, Singapore
Email: elehy@nus.edu.sg

Abstract

Modern day applications require use of multi-processor systems for reasons of performance, scalability and power efficiency. As more and more applications are integrated in a single system, mapping and analyzing them on a multi-processor platform becomes a multi-dimensional problem. Each possible set of applications that can be concurrently active leads to a different use-case (also referred to as scenario) that the system has to be verified and tested for. Analyzing the feasibility and resource utilization of all possible use-cases becomes very demanding and often infeasible.

Therefore, in this paper, we highlight this issue of being able to analyze applications in isolation while still being able to reason about their overall behavior - also called as composability. We make a number of novel observations about how arbitration plays an important role in system behavior. We compare two commonly used arbitration mechanisms, and highlight the properties that are important for such analysis. We conclude that none of these arbitration mechanisms provide the necessary features for analysis. They either suffer from scalability problems, or provide unreasonable estimates about performance, leading to waste of resources and/or undesirable performance.

We further propose to use a resource manager (RM) to ensure applications meet their performance requirements. The basic functionalities of such a component are introduced. A high-level simulation model is developed to study the performance of RM, and a case study is performed for a system running an H.263 and a JPEG decoder. The case study illustrates at what granularity of control a resource manager can effectively regulate the progress of applications such that they meet their performance requirements.

***Index Terms*—Composability, Predictability, MPSoC, Arbitration, Non-preemptive, Heterogeneous, Resource Manager.**

1. Introduction

Current developments in set-top box products for media systems show a need for integrating a (potentially large) number of applications or functions in a single device. The consumer should not experience any significant artifacts or delays when functions are switched on or off, or when multiple functions are executed concurrently. This places high demands on the arbitration of available computational resources as well as memory accesses. For systems with a single general-purpose processor supporting pre-emption, the analysis of schedulability of task deadlines is well known [17] and widely used. In heterogeneous multi-processor embedded systems without pre-emption however, the theory of rate-monotonic analysis and the likes do not apply. In order to predict the timing behavior of applications running on current and future hardware platforms, an alternative method for analysis is a necessary requirement for limiting the design cost.

The analysis becomes a daunting task with the large number of possible *use-cases*. (A *use-case* is defined as a possible set of concurrently running applications.) Future multimedia platforms may easily run 20 applications in parallel, corresponding to an order of 2^{20} possible use-cases. It is clearly impossible to verify the correct operation of all these situations through testing and simulation. The product divisions in large companies already report 60% to 70% of their effort being spent in verifying potential use-cases and this number will only increase in the near future. This has motivated researchers to emphasize the ability to analyze and predict the behavior of applications and platforms without extensive simulations of every *use-case.*

We would ideally want to analyze each sub-application in isolation, thereby reducing the analysis time to a linear function, and still reason about the overall behavior of the system. One of the ways to achieve this, would be complete *virtualization.* This essentially means dividing the available resources by the total number of applications in the system. The application would then have exclusive access to its share of resources. For example, if we have 100

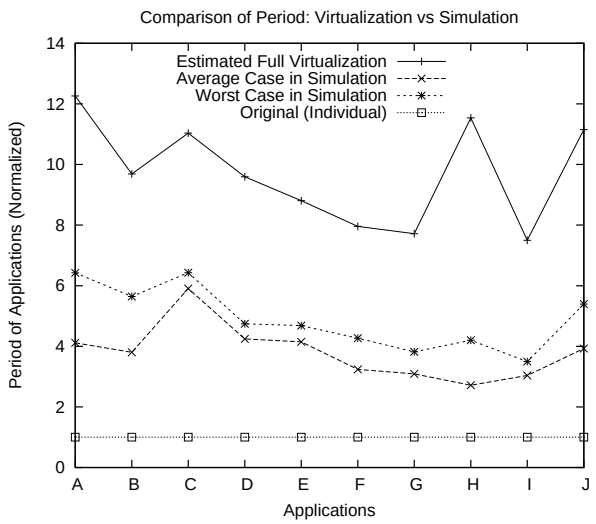


Figure 1. Application performance as obtained with full virtualization in comparison to simulation.

MHz processors and a total of 10 applications in the system, each application would get 10 MHz of processing resource. The same can be done for communication bandwidth and memory requirements. Even discounting the fact that we cannot achieve true *generalized processor sharing*, we have two kinds of problems. When fewer than 10 tasks are active, the tasks will not be able to exploit the extra available processing power, leading to wastage. Secondly, the system would be grossly over-dimensioned when the peak requirements of each application are taken into account, even though these peak requirements of applications may rarely occur and never overlap.

Figure 1 shows this disparity in more detail. The graph shows the period of ten streaming multimedia applications (inverse of throughput) when they are run concurrently. The period is the time taken for one iteration of the application. The period has been normalized to the original period that is achieved when each application is running in isolation. If full virtualization is used, the period of applications increases to about ten times on average. In practice, however, it increases only about five times. A system which is built with full-virtualization in mind, would therefore, utilize only 50% of the resources. Thus, throughput decreases with complete virtualization.

Another way to reduce the complexity would be to analyze the applications in isolation with as little information from other applications as possible and then define a *compose* function to compute total requirement of the system. This reduces the complexity of the analysis and still leads to higher utilization of resources. In this paper, we study *how to reduce exponential analysis complexity to linear (or at most polynomial) complexity, without paying the overhead of complete virtualization.* This problem is called as *composability problem.* In this paper, we study this problem in detail, and bring to notice some very interesting and novel observations. These are not found in any published literature, and are non-trivial to derive and important. While the concepts are fairly basic, the interaction of multiple applications on a heterogeneous multiprocessor platform makes the analysis complex.

Clearly, arbitration plays an important role in resolving contention of resources. The overall system behavior depends on the arbitration mechanism to a large extent. In this paper, we compare the suitability of two very simple, yet often used, arbitration mechanisms for such an analysis. We propose a novel multiprocessor architecture with an arbiter for each processor tile. We state requirements for arbiters in future media platforms, and analyze these properties (as much as possible) using Synchronous Data Flow (SDF) graphs [16]. SDF graphs are a class of *models of computation* that allows analysis of systems at design time. There are two requirements for using these models of computation, namely the development of good analysis tools that exploit these models, and the ability to capture real-world behavior.

Two kinds of arbitration mechanisms are considered in this paper - *static*- and *dynamic-ordering*.

- Static order: Actors - as defined in SDF model - are repeatedly executed in an order specified by a predefined list. If an actor is not ready to execute (i.e. its input data has not yet arrived), the processor will halt and wait.
- Dynamic order: Arbitration mechanisms which do not have a pre-defined order fall in this category. Examples of dynamic order arbitrations are round-robin with skipping and first-come-first-serve (FCFS). Order is either not specified at all at design-time (FCFS) or only a recommended order is specified which can be changed in favor of work-conserving arbitration (round-robin with skipping - RRWS). We shall limit to two dynamic arbitration mechanism, namely, FCFS and RRWS.

Further, we find that none of the above arbitration mechanisms can be applied directly to composability analysis. We therefore propose an alternative - the use of a *Resource Manager (RM)* for non-preemptive heterogeneous Multi-Processor Systems-on-Chip (MPSoCs). We state the basic functionalities expected from such a component. While most research only focuses on schedulability analysis, here we also discuss the required protocol needed to realize a working system. Further, we present a simulation model developed using the modeling language POOSL [22] to validate this protocol. POOSL is a very expressive modeling language with a small set of powerful primitives and completely formally defined semantics. It furthermore serves as a basis for performance analysis. This setup can be used to

study various trade-offs in design of an MPSoC. In this paper, we use the setup to study the trade-off between control overhead of monitoring and budget enforcement on the one hand, and performance of applications on the other.

We assume the following for the scope of this paper.

- Multi-processor: For reasons of scalability and energy consumption, a single high-performance processor is not suitable for satisfying the computational demands placed on future consumer devices.
- Non-preemptive: DSP processors and accelerator hardware typically have a lot of states. As a result, the interrupt delay is significant, whereas the typical execution time of an actor is much smaller than a task on a conventional general-purpose processor. For embedded systems in particular, non-preemptive scheduling algorithms are easier to implement than preemptive algorithms and have dramatically lower overhead at runtime [13].
- Efficient arbitration: The arbitration mechanism should be efficient, since the time required for arbitration has to match the grain of actor executions.
- Heterogeneous architecture: The heterogeneity of systems is increasing with the use of specialized digital hardware, application domain processors and other IP (intellectual property) blocks on a single chip using complex networks. This implies that mapping options to hardware are often limited.

Section 2 provides an overview of related work. In Section 3 we explain the concepts of composability and analyze the effects of arbitration on the performance of systems. The properties and requirements for arbitration are further discussed in Section 4. Section 5 discusses the system we propose with the use of a resource manager. Experimental results with this setup are presented in Section 6. In Section 7 we present the conclusions and directions for future work.

2. Related Work

For traditional systems, with a single general-purpose processor supporting pre-emption, the analysis of schedulability of task deadlines is well known [17] and widely used. Non-preemptive scheduling has received considerably less attention. It was shown by Jeffay et al. [13] and further strengthened by Cai and Kong [6] that the problem of determining whether a given periodic task system is feasible even upon a single non-preemptive processor is already intractable. Also, research on multiprocessor real-time scheduling has mainly focused on preemptive systems [8, 3].

Recently, more work has been done on non-preemptive scheduling for multiprocessors [2]. Alternative methods have been proposed for analyzing task performance and resource sharing. A formal approach to verification of MPSoC performance has been proposed in [20]. Use of real-time calculus for schedulability analysis was proposed in [23]. Besides providing a very pessimistic bound, the above analysis techniques are also very compute intensive and require a very high design time effort.

SDF models have also been used extensively for analyzing performance of systems. In [1], the authors propose to analyze performance of a *single application* modeled as an SDF graph (SDFG) mapped on a multi-processor system by decomposing it into a homogeneous SDFG (HSDFG) [21], and modeling dependencies of resources by adding extra edges on the vertices. Vertices model actors in the application. This can result in an exponential number of vertices [19], after which the cycle mean of each cycle is computed [7]. Throughput computation algorithms that have a polynomial complexity for HSDFGs, therefore have an exponential complexity for SDFGs. Algorithms have been proposed to reduce average case execution [10], but it still takes in practice $O(n^2)$ time where n is the number of vertices in the graph. Besides, even for one application, the number of ways extra edges can be added to model actor dependency is exponential. For multiple applications the number of computations is huge; and if we need to obtain throughput of the graph, an HSDFG can take too much time to analyze and provide results [10]. Further, only static order arbitration can be modeled using this technique while the best performance of SDFG applications is obtained in a self-timed mechanism [21]. Dynamic ordering allows for this self-timed behavior in applications.

For *multiple applications*, an approach that models resource contention by computing *worst-case-response-time* for TDMA scheduling (requires preemption) has been analyzed in [4]. This analysis also requires limited information from the other SDFGs, but gives a very conservative bound that may be too pessimistic. As the number of applications increases, the bound increases much more than the average case performance. Further, this approach assumes a preemptive system. A similar worst-case analysis approach for round-robin is presented in [12], which also works on non-preemptive systems, but suffers from the same problem of lack of scalability. This is best illustrated by means of an example shown in Figure 2. The example shows 3 application SDF [16] graphs - A, B, and C, with 3 actors each. Actors T_i are mapped on processing node P_i where T_i refers to A_i, B_i and C_i for $i = 1, 2, 3$. Each actor takes 100 time units to execute as shown.

Since 3 actors are mapped on the same node, an actor may need to wait when it is ready to be executed at a node. The maximum waiting time for a particular actor, can be computed by considering the *critical instant* as defined by Liu and Layland [17]. The critical instant for a task is de-

Properties	Liu *et* al [17] 1973	Jeffay *et* al [13] 1991	Baruah [2] 2006	Richter *et* al [20] 2003	Hoes [12] 2004	Our method
Multiprocessor	No	No	Yes	Yes	Yes	Yes
Heterogeneous	N. A.	N. A.	No	Yes	Yes	Yes
Non-preemptive	No	Yes	Yes	Yes	Yes	Yes
Non-Periodic support	No	Yes	No	Yes	Yes	Yes
Utilization	High	High	Low	Low	Low	High
Guarantee	Yes	Yes	Yes	Yes	Yes	No

Table 1. Summary of related work (Heterogeneous property is not applicable for uniprocessor schedulers)

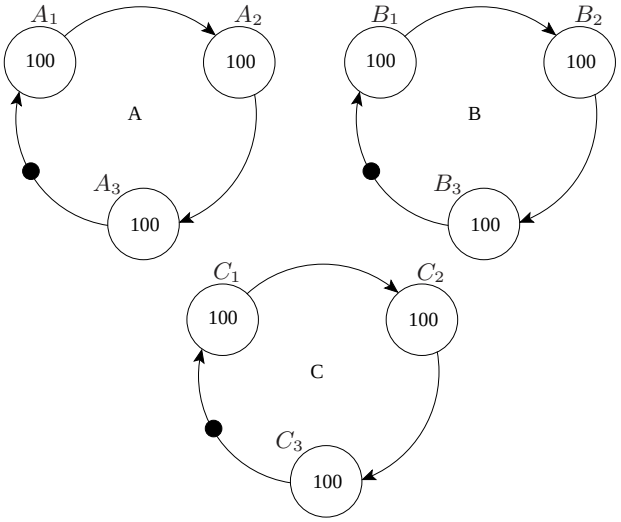


Figure 2. Example of a set of 3 application graphs.

fined as an instant at which a request for that task will have the largest response time. Since the response time is equal to sum of its waiting time and execution time, with executing time being assumed constant, it can be translated as the instant at which we have the largest waiting time. For dynamic scheduling mechanisms, it occurs when an actor becomess ready just after all the other actors, and therefore has to wait for all the other actors. Thus, the total waiting time is equal to the sum of processing times of all the other actors on that particular node and given by the following equation.

$$t_{wait}(T_{ij}) = \sum_{k=1,k\neq i}^{m} t_{exec}(T_{kj}) \quad (1)$$

Here $t_{exec}(T_{ij})$ denotes the execution time of actor T_{ij}, i.e. actor of task T_i mapped on processor j. This leads to a waiting time of 200 time units as shown in Figure 3. An extra node has been added for each 'real' node to depict the waiting time (WT_A_i). This suggests that each application will take 900 time units in the worst case to finish execution. This is the maximum period that can be obtained for applications in the system, and is therefore guaranteed. A resource manager approach can nicely interleave the actors and each application will only require 300 time units, thereby achieving three times the performance guaranteed by analysis in [12].

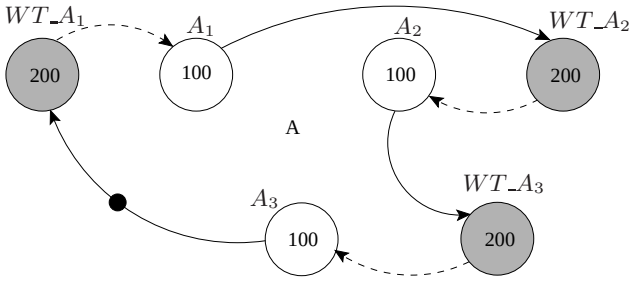


Figure 3. Modeling worst case waiting time for application A in Fig. 2.

Table 1 shows a comparison of some analysis techniques that have been presented in literature, and where our approach is different. As can be seen, all of the research done in multiprocessor domain provides low utilization guarantees. Our approach on the other hand aims at achieving high utilization by sacrificing hard guarantees.

3. Composability

A typical multi-processor system-on-chip application is usually composed of more than one smaller sub-applications. For example, a mobile phone supports various applications that can be active at the same time, such as listening to mp3 music, typing an SMS and downloading a java application in the background. Evaluating resource requirement for each of these cases can be quite a challenge even at design-time, let alone at run-time.

We define composability as the degree to which the mapping and analysis of applications on the platform can be performed in isolation. Some of the things we would like to analyze in isolation are for example, deadlock occurrence, application throughput and computing a static schedule if needed. Clearly, since there are more than one application mapped on a multi-processor system, there is bound to be contention for the resources. Due to this contention, the throughput analyzed for an application in isolation is not always achievable when the application runs together with other applications. This will be demonstrated with the aid

of an example in section 3.2. In section 3.3 we show that in case of static scheduling, the schedule complexity (and therefore the program storage requirements) grows more than linearly with the number of mapped applications. In Section 3.4 we consider the computational requirements for computing the timing behavior, and in section 3.5 we consider the smooth transition when a new application enters the system. All these observations are novel and have not been studied in this context of heterogenous MPSoC platforms. First, however, we shall provide a short introduction to the modeling used for analysis in this paper, namely SDF in Section 3.1.

3.1. SDF Modeling

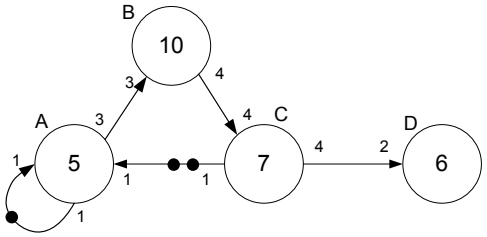


Figure 4. Example of an SDF Graph

Figure 4 shows an example of an SDF Graph. There are four actors in this graph. As in a typical data flow graph, a directed edge represents the dependency between tasks. Tasks also need some input data (or control information) before they can start and usually also produce some output data; such information is referred to as *tokens*. Actor execution is also called *firing*. So when sufficient input tokens are available on all incoming edges of an actor, it is ready to fire. Further, when an actor is ready to fire on a processing node P, we say it *arrives* at P.

An actor can only start execution when the required number of tokens are present on each of its incoming edge, and upon completion produces the number of tokens specified on the outgoing edge. In an actual implementation, edges indicate buffers in physical memory. The edges may also contain *initial tokens* which denote data dependencies across various iterations of the algorithm. These are indicated by bullets on the edges.

In the above example, only A can start execution from the initial state, since the required number of tokens are present on all of its incoming edges. Once A has finished execution it will produce 3 tokens on the edge to B. B can then proceed as it has enough tokens and upon completion produce 4 tokens on the edge to C. Another thing to note is that since there are two initial tokens on the edge from C to A, A can again fire as soon as it has finished the first execution, without waiting for C to execute.

SDF graphs allow analysis of maximum achievable throughput of a system by various algorithms [21]. Further, SDF analysis also allows us to identify if a particular graph or a schedule will result in a deadlock. HSDF - Homogenous SDF - is a special class of SDF in which the number of tokens consumed and produced is always equal to 1. For simplicity (and without loss of generality), we shall consider only HSDF graph, unless otherwise mentioned.

3.2. Composability Problem

Figure 5 shows an example of two task graphs A and B with three actors, each mapped on a 3-processor system. Actors A_1 and B_1 are mapped onto P_1, A_2 and B_2 are mapped onto P_2, and A_3 and B_3 are mapped onto P_3. Each actor as shown takes 100 clock cycles to execute and because of dependency within the task graph, only one iteration of each can be active. Thus, each task uses only 33% of each processor node, and hence each processor can be utilized at maximum for 67% because of executing a task of each application. However, due to the dependencies - both inter- and intra-task graphs, the maximum achievable processor utilization is only 50%.

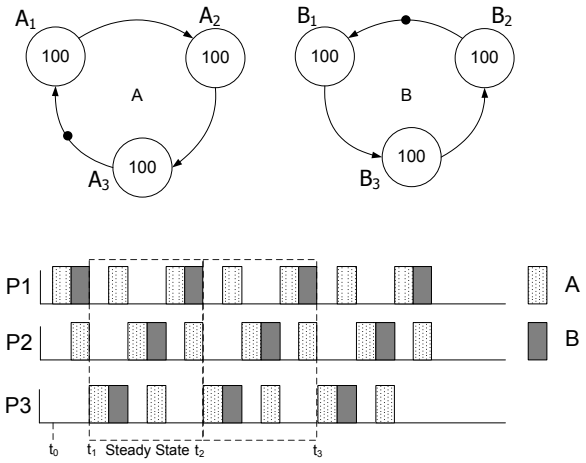


Figure 5. An example showing why composability needs to be examined. Individually each task takes 300 clock cycles to complete an iteration and requires only 33% of processor resources. However, when another job enters in the system, it is not possible to schedule both of them with their optimal schedule of 300 clock cycles, even though the total request for a node is only 67%.

Figure 5 also shows a schedule obtained when the actors are scheduled using dynamic scheduling. The first contention between tasks A and B occur at instant t_0, when both A_1 and B_1 are ready to execute on P_1. This arbitration goes to A_1, while B_1 waits. Another contention occurs at t_1 for processor P_3, and then for P_2 followed by P_1. The schedule shown in the figure assumes that A wins every arbitration.

The schedule soon settles into a steady state of 600 clock cycles, in which A completes two iterations, while B completes only one. If B wins every arbitration, the situation is reversed and B would execute twice as many times as A. Since each processor is idle for half the number of clock cycles, the utilization is only 50%. We tried many other schedules (including preemption), some of which will be shown later, and we could not achieve better performance. This is a very important observation, and demonstrates that even for very simple cases, it is not easy to predict the performance of multiple applications. As can be seen from the above example, simply adding up computational load of a processor is not realistic.

3.3. Overhead of Multiple Use-cases

A static order strategy requires one to compute the optimal schedule for each of the possible combinations. As the number of applications increases, the total number of potential use-cases rises exponentially. For a system with 10 applications in which up to 4 can be active at the same time, there are approximately 400 possible combinations - and it grows exponentially as we increase the number of concurrently active applications. If static scheduling is used, besides computing the schedule for all the use-cases offline (design-time), one also has to be aware that they need to be stored at run-time. The scalability of using static scheduling for multiple jobs is therefore limited.

Dynamic scheduling mechanisms are more scalable in this context. Clearly in FCFS, there is no such overhead as no schedule is stored beforehand. In RRWS the easiest approach would be to store all actors for a processor in the schedule. When an application is not active, its actors are simply skipped, without causing any trouble for the scheduling kernel. It should also be mentioned here that if an actor is required to be executed multiple number of times, one can simply add more copies of that actor in this list. This is one reason why RRWS might be preferred over FCFS. RRWS in this way can provide easy rate-control mechanism.

3.4. Computing Static Order Schedules

Three task graphs - A, B and C are shown in Figure 6. Each is an HSDF with three actors. Let us assume each actor is mapped onto one processing node. Let us also assume that actors T_{i1} are mapped onto P_1, T_{i2} are mapped onto P_2 and T_{i3} are mapped onto P_3; where T_i refers to tasks A, B and C. This contention for resources is shown by the dotted arrows in Figure 7. Clearly, by putting these dotted arrows, we have fixed the actor-order for each processor node. If an optimal ordering is to be computed for the entire system when all three tasks are active at the same time, we need to

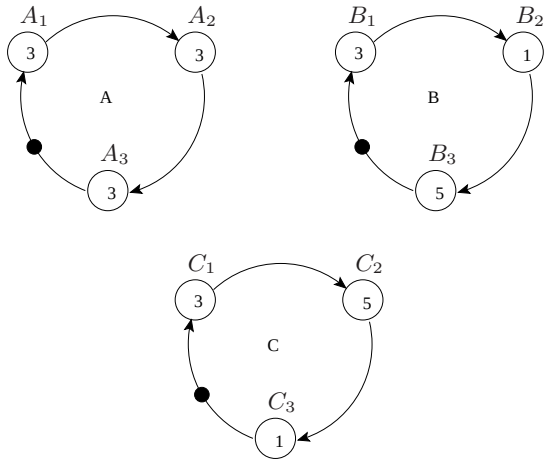


Figure 6. Example of a system with 3 applications.

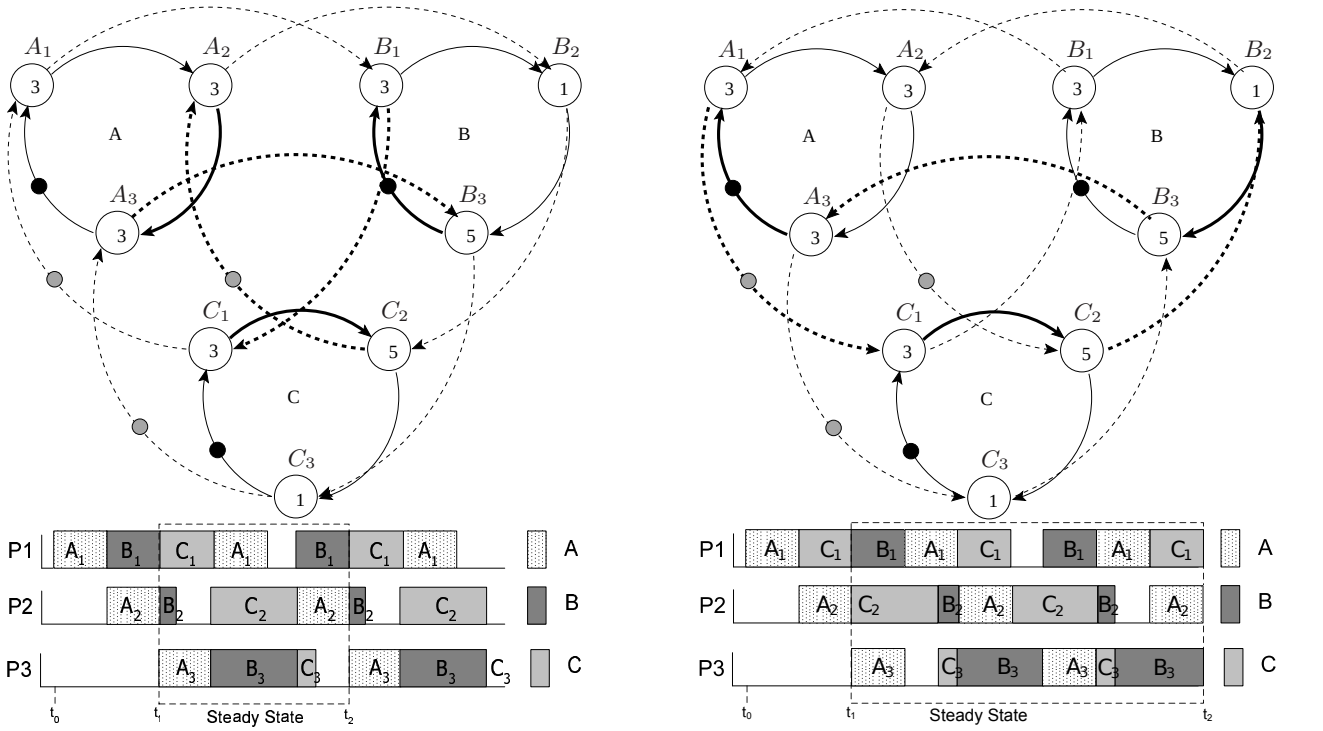
combine different graphs into one big HSDF for complete analysis. Figure 7(a) shows one such possibility when the dotted arrows are used to combine the three task graphs. Extra tokens have been inserted in these dotted edges to indicate initial state of arbiter.

An astute reader would have noticed that this would only be possible if each task is required to be run an equal number of times. If the rates of each task are not the same, we need to introduce multiple copies of actors to achieve the required ratio, thereby increasing analysis complexity.

When throughput analysis is done for this complete graph, we obtain a mean cycle count of 11 [21]. This also gives us the ideal order for each processing node. The bold arrows represent the edges that limit the throughput. The corresponding schedule is also shown. One actor of each of the tasks, namely T_{i1}, is ready to fire at instant t_0. We find that the graph soon settles into the periodic schedule of 11 clock cycles. The period is denoted in the schedule diagram of figure 7(a) between the time instant t_1 and t_2.

Figure 7(b) shows just another of the many possibilities for ordering the actors of the complete HSDF. Interestingly, the mean cycle count for this graph is 10, as indicated by the bold arrows. In this case, the schedule starts repeating after time t_1, and the steady state length is 20 clock cycles, as indicated by difference in time instants t_1 and t_2. However, since two iterations for each task are completed, the period is only 10 clock cycles.

From arbitration point of view, if task-graphs are analyzed in isolation, there seems to be no reason to prefer task B or C after A has finished executing on processor 1. There is at least a delay of 6 clock cycles before task A needs processor 1 again. Also, since B and C each takes only 3 clock cycles, 6 clock cycles are enough to finish their execution. Further both are ready to be fired, and will not cause any delay. Thus, the local information about a job and the tasks that need a processor resource does not easily dictate preference of one task over another. However, as we see in this



(a) Graph with clockwise schedule (static) gives MCM of 11 cycles.

(b) Graph with anti-clockwise schedule (static) gives MCM of 10 cycles.

Figure 7. Three individual task graphs give different MCM when scheduled differently. The respective cycles which gives this MCM are shown in bold.

example, executing task C is indeed better for the overall performance. Computing a static order relies on the global information and produces the optimal performance. This becomes a serious problem when considering MPSoC platforms, since constructing the overall HSDF graph and then computing its throughput is very compute intensive.

The number of possibilities for constructing the HSDF from individual graphs is very large. In fact, if one tries to combine g graphs of say a actors, scheduled in total on a processors, there happen to be $((g-1)!)^a$ unique combinations, each with a different actor ordering, for only single occurrence of each application. (Each processor has g actors to schedule, and therefore $(g-1)!$ unique orderings on a single processor. This leads to $((g-1)!)^a$ unique combinations since ordering on each processor is independent of ordering on another.) To get an idea of vastness of this number, if there are 5 graphs with 10 actors each we get 24^{10} or close to $6.34 \cdot 10^13$ possible combinations. If each computation takes 1ms to compute [10], 2009 years are needed to evaluate all possibilities. This is only considering the cases with equal rates for each, and only for HSDF graphs. A typical SDF graph with different execution rates would only make the problem even more infeasible, since the transformation to HSDF yields many actor copies. An exhaustive search through all the graphs is, therefore, not an option. Thus, a simpler arbitration mechanism is needed with lesser design overhead.

3.5. Deadlock

Deadlock avoidance and detection is an important concern when tasks are activated dynamically. When static order is being used, every new use-case requires a new schedule to be loaded into the kernel. A naive reconfiguration strategy can easily send the system into deadlock. This is demonstrated with an example in Figure 8.

Say actors A_2 and B_3 are running in the system on processor node 2 and 3 respectively. Further assume that static order for each processor currently is A, B when only these two are active, and with a third task C, it becomes A, B, C for each node. When C is activated, it gets processor 1 since that is available. Let us see what happens to processor 2: application A is running on it and it is then assigned to application B. Processor 3 is assigned to C after B is done. Thus, after each actor is finished executing on its currently assigned processor, we obtain A_3 waiting for processor 3 that is assigned to task C_3, B_1 waiting for processor 1 which is assigned to A_1, and C_2 waiting for processor 2, which is

Node	Assigned to	Task waiting	Reassigned in RRWS
P1	A	B	B
P2	B	C	C
P3	C	A	A

Table 2. Table showing a deadlock condition

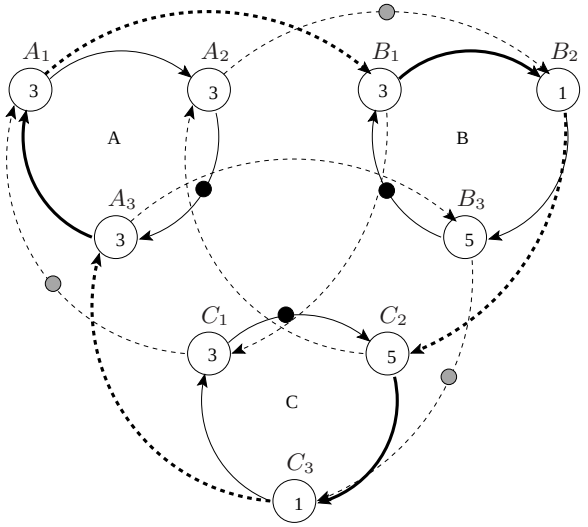


Figure 8. Deadlock situation when a new job, C arrives in the system. A cycle $A_1, B_1, B_2, C_2, C_3, A_3, A_1$ is created without any token in it.

assigned to B_2. This can be expressed by Table 2.

Looking at Figure 8, it is easy to understand why the system goes into a deadlock. The figure shows the state when each task is waiting for a resource and not able to execute. The tokens in the individual sub-graph show which actor is ready to fire, and the token on the dotted edge represents which resource is available to the task. In order for an actor to fire, the token should be present on all its incoming edges - in this case both on the incoming dotted edge and the solid edge. It can be further noted that a cycle is formed without any token in it. This is clearly a situation of deadlock [14] since the actors on this cycle will never be enabled. This cycle is shown in Figure 8 in bold edges. It is possible to take special measure to check and prevent the system from going into such deadlock. This, however, implies extra overhead at both design-time and run-time. The task may also have to wait before it can be admitted into the system.

The deadlock situation can be avoided quite easily by using dynamic scheduling. Clearly, for FCFS, it is not an issue since resources are never blocked for non-ready actors. For RRWS, when the system enters into a deadlock, the arbiter would simply skip to the actor that is indeed ready to execute. Thus, processors 1, 2 and 3 are reassigned to B, C and A as shown in Table 2. In addition, a task can be activated at any point of time without worrying about deadlock. In dynamic scheduling, there can never be a deadlock due to dependency on processing resources for atomic non-preemptive systems.

4. Arbitration Requirements

Table 3 shows a summary of various performance parameters that we have considered in this paper. The static scheduling clearly has a higher design-time overhead of computing the optimal order for each use-case. The run-time scheduler needed for both schedulers is quite simple, since only a simple check is needed to see when the actor is active and ready to fire. The memory requirement for static scheduling is however, higher than that for a dynamic mechanism. The static order certainly scores better than a dynamic one when it comes to predictability of throughput and resource utilization. Static-order approach is also better when it comes to admitting a new job in the system since the resource usage prior and after admitting the job are known at design time. Therefore, a decision whether to accept it or not is easier to make. However, extra measures are needed to reconfigure the system properly so that the system does not go into deadlock as mentioned earlier in Section 3.5.

A dynamic approach is able to handle dynamism better than static order since orders are computed based on the worst-case execution time. When the run-time varies significantly, a static order is not able to benefit from early termination of a process. The biggest disadvantage of static order, however, lies in the fact that any change in the design, e.g. adding a use-case to the system or a new processor node, can not be accommodated at run-time.

From this summary, we conclude that dynamic mechanisms satisfy most of our criteria. and is hence quite suitable for a resource manager in an MPSoC. However, a resource manager also needs a mechanism to enforce a time-budget on the available resources for each application separately, to ensure that they do not exceed the resource allocated for it. We modeled such a resource manager and implemented dynamic arbitration mechanisms. In the next section we explain the set up of the system

5. System Setup

An MPSoC consists of three kinds of resources - computation, communication and storage. We ignore in this paper the contention for communication and storage resources as computation resources already demonstrate the complexity of the scheduling problem; and management of other resources can be added in similar fashion. Figure 9 shows an overview of the model. It has a tiled architecture, where each tile can either be a processing tile, or a memory tile.

Property		Static Order	Round Robin with skipping
Design time overhead	Calculating Schedules	- -	++
Run-time overhead	Memory requirement	-	++
	Scheduling overhead	+	+
Predictability	Throughput	++	- -
	Resource Utilization	+	-
New job admission	Admission criteria	++	- -
	Deadlock-free guarantee	-	++
	Reconfiguration overhead	-	+
Dynamism	Variable Execution time	-	+
	Handling new use-case	- -	++

Table 3. Properties of Scheduling Strategies

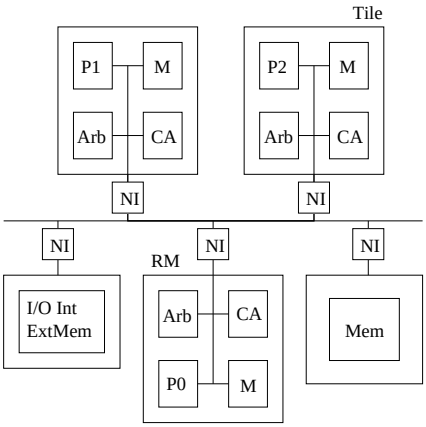


Figure 9. Proposed system architecture.

The I/O interface in the model allows for interaction with the input/output peripherals in the system, for example, keypad in case of a mobile phone. If some memory is stored off-chip, it can also be accessed via the same interface. The resource manager (RM) runs on one of the processing tiles. For example, in figure 9, RM is run on processor $P0$. The RM directs the usage of resources in the system. If the number of nodes is large, more than one tile can be allocated for better distributed control.

Each processing tile consists of a processor, a small local memory, a local arbiter and a communication assist (CA) that is similar to the one described in [5]. A network-interface (NI) is used to connect each tile to the network. Æthereal is assumed to be used for communication infrastructure [11]. Besides, other benefits, it provides flow-control at the level of connections. An architecture with both CA and Æthereal allows easy decoupling of computation and communication. When an actor of one application finishes its execution, the output data is placed in the local memory of the processor. The processor then proceeds with execution of other actor without spending time in communicating the data to the destination node. This communication is handled by the CA which transfers the data to the destination memory. When data from all the input edges has been received, the actor is ready to be executed and can be queued in the processor-arbiter.

5.1. Resource Manager

With increasing dynamism in modern applications, the need for a separate task to monitor and direct the usage of resources has arisen [24]. Such a *resource manager* is responsible for just that. It controls the access to resources - both critical and non-critical, and enforces their usage. Clearly, admission of a new job also falls in the scope of resource manager. When a new job arrives in the system and needs resources, the resource manager checks the current state of the system and decides whether it has enough resources to accommodate it. It also enforces a specified resource budget for a task to ensure it only uses what was requested. These two main functions, namely *admission control* and *resource budget enforcement*, are explained below.

5.1.1 Admission Control

One of the main uses of a resource manager is admission control for new applications. In a typical high-end multimedia system, applications are started at run-time, and determining if there are enough resources in the system to admit new applications is non-trivial. As mentioned earlier, the number of possible combinations with even a few applications can be very high. It is therefore, not feasible to analyze all possible scenarios of applications. Thus, it is difficult to predict the resource utilization when multiple applications are allowed to run in the system.

When the RM receives request for a new application to be started (through the I/O interface, or through another application already running in the system), it fetches description of the incoming application from the memory. The description contains the following

- Desired performance: specified as average throughput.
- Performance bound: typically the minimum throughput that the application may run at. This is explained in more detail in Section 5.2.
- Actor array: A list of actors along with their execution times, repetition vector [10] and node to which they are mapped to.

The above information can also be in form of a *pareto-curve* where each point in the curve corresponds to a desired quality of the application and specification as above. With this information, the RM checks if there are enough resources available for all the applications to achieve their desired performance using a *predictor* (or in the case of a pareto-curve, at which point the application can be run). The predictor module in this setup simply adds the processor utilization for each processing node and checks if it is less than 100%. However, more research needs to be done for a more intelligent predictor. An idea could be to reduce the 100% to $x\%$ where x is adaptive depending on the number of misses in the past, for example.

One possibility would be to compute a static order schedule of all the use-cases at design-time and store them, but it is not a good idea as mentioned earlier. Another possibility is to analyze the applications running at run-time and check how the incoming application will interact with it. This clearly also makes the predictor more complex. A good balance between run-time feasibility and design-time complexity needs to be obtained.

if the predictor admits the application, then the resource manager *boots* the application. This translates to loading the application code from memory (possibly external) into local memories of respective processors and enabling/adding the application in the arbiter. Once the application is started, it sends a signal to the RM at the end of every iteration to keep it updated.

5.1.2 Resource Budget Enforcement

This is one of the most important functions we expect a resource manager to do. When multiple applications are running in a system (often with dynamically changing execution times), it is quite a challenge to schedule all of them such that they meet their throughput constraints. Using a static scheduling approach is neither scalable, nor adaptive to dynamism present in a system [15]. A resource manager, on the other hand, can monitor the progress of applications running in the system and enforce the budgets requested at run-time.

Clearly, the monitoring and enforcement also has an overhead associated with it. Granularity of control is therefore a very important consideration when designing the system, and determines how often the RM inspects the system. We would like to have as little control as possible while

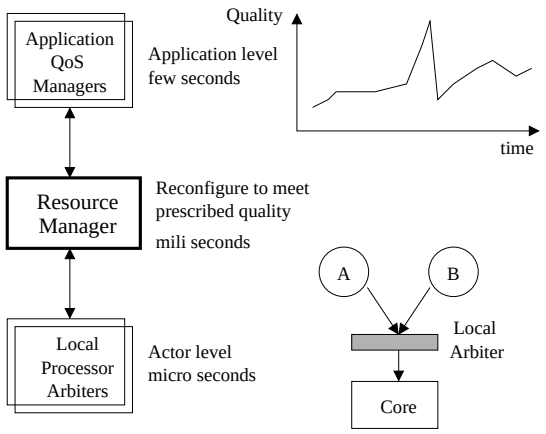


Figure 10. Resource Manager tries to achieve the specified quality without interfering at the actor level

achieving close to desired performance. This is explained further in Section 5.4.

Task migration

Task migration is not considered in this paper, but it could be useful in cases when a particular actor can be scheduled on multiple nodes. A technique to achieve low-cost task migration in heterogeneous MPSoC is proposed in [18].

Arbiter vs Resource Manager

Figure 10 shows how the resource manager differs from the application QoS manager and from the processor arbiter. A quality-of-service (QoS) manager for an application might dictate varying desired levels of application quality at different points in time. For example, if an application goes into background, its quality level might change. Such change is unlikely to happen more than once every few seconds. On the other end of control spectrum we have a processor arbiter, which may have to choose the actors that can be executed on a particular node after an actor has finished executing. This depends on the grain of actors, but is generally in the order of micro-seconds. The resource manager operates in between the two extremes and tries to achieve the quality specified by QoS manager by reconfiguring the arbiter once in a while.

Achieving desired rate for different applications running concurrently can also be achieved by rate-controlled-static-priority as mentioned in [25]. However, this would require a very complicated arbiter at every tile in the system, amounting to a high overhead in terms of chip area. Further, an application consists of multiple actors, which can often operate at different rates in different iterations. When the rate-control is achieved via local arbiters at cores, each arbiter has to be aware of the global state of the application. This

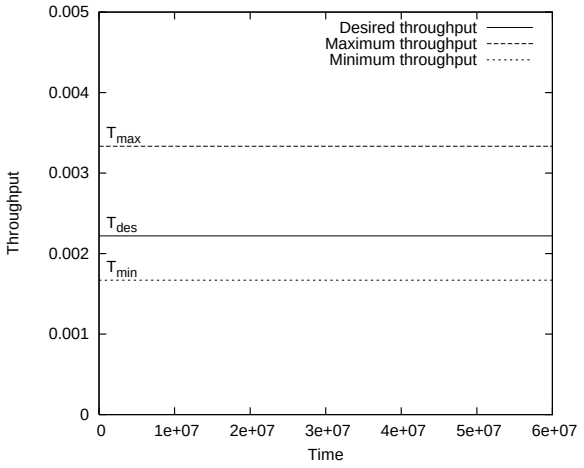


Figure 11. Boundary Specification for non-buffer critical applications.

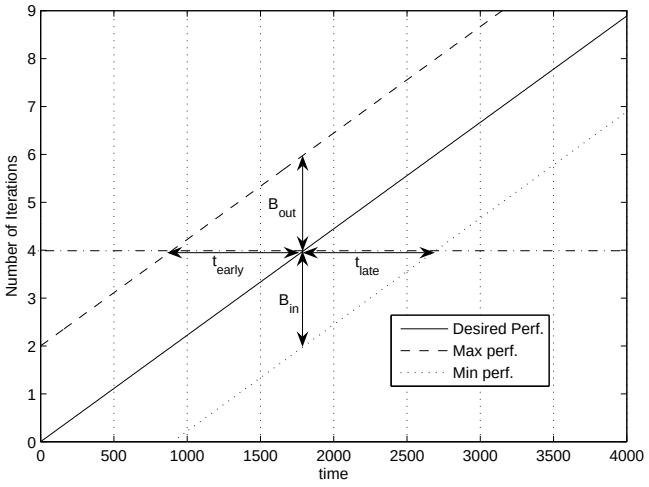


Figure 12. Boundary specification for buffer-critical applications or constrained by input/output rate.

is much harder to achieve in practice, and also expensive in terms of communication. Therefore, we propose using a simple arbiter and an extra component - resource manager, for this control.

5.2. Performance Bound Specification

Figure 11 shows an example of how performance bound may be specified. In the figure, the middle line T_{des} indicates the desired throughput, and T_{max} and T_{min} indicate the maximum and minimum throughput that an application may tolerate respectively. However, this only applies to applications that do not have an input or output throughput constraints. For applications with strict throughput constraints on input or output, we need a different kind of specification as shown in Figure 12.

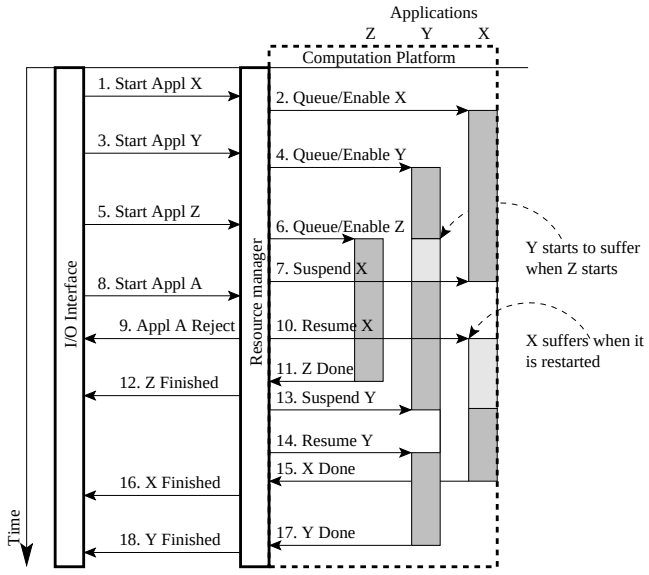


Figure 13. Interaction diagram between various modules in the system-setup.

The deviation from the desired throughput in the vertical direction signifies the extra buffering that is needed. The same is indicated in Figure 12 by B_{out} and B_{in} to indicate the maximum output and input buffer present for the application respectively. If an application is running two iterations ahead of the desired rate, the output of those two iterations needs to be buffered somewhere. The same applies to input buffer as well when an application is lagging behind. The deviation in the horizontal direction signifies how 'early' or 'late' the application completes its execution as compared to desired time. For applications where such jitter is more critical than the average throughput achieved, the performance bound should be as specified in Figure 12.

5.3. Resource Management Protocol

Figure 13 shows an example interaction diagram between various modules in the design. The user-interaction module sends a signal to the respective application manager when any application is to be started. The resource and application managers are responsible for actually running the applications and interacting with the computation platform. The platform module can consist of a number of processors and each processor has a scheduler which can for example be first-come-first-serve (FCFS), round-robin or round-robin-with-skipping [15]. Other scheduler types can be easily added in the model.

In Figure 13, the user-interface module sends a request to start application X (1). The resource manager checks if there are enough resources for it, and then admits it in the system (2). Applications Y and Z are also started respec-

tively soon after as indicated on the figure (3-6). However, when Z is admitted, Y starts to deteriorate in performance. The resource manager then sends a signal to suspend X (7) to the platform because it has slack and Y is then able to meet the desired performance. When X resumes (10), it is not able to meet its performance and Y is suspended (13) because now Y has slack. When the applications are finished, the result is transmitted back to the user-interface (12, 16 and 18). We also see an example of application A being rejected (9) since the system is possibly too busy, and A is of lower priority than applications X, Y and Z.

5.4. Suspending Applications

Each application sends a signal to the RM upon completion of each iteration. This is achieved by appending a small message at the end of the output actor of the application graph[1]. This allows the RM to monitor the progress of each application at little cost. After every sample period - defined as *sample points*, the RM checks if all applications are progressing at their desired level. If any application is found to be running below the desired throughput, the application which has the most slack (or the highest ratio of achieved to desired throughput) is suspended. The suspended application is re-activated if all applications are running above the desired throughput. Suspension and re-activation occur only at sample points.

Each arbiter maintains two lists - an actor ready queue, and an application enable list. Once the processor is available, the arbiter checks the actor at the head of the queue, and if its corresponding application is enabled, it is executed. Otherwise, the next available task (with enabled application) is executed. Suspension of an application is achieved by sending a temporary disable signal to the arbiters running the application. Say, for example, if application A_2 has actors mapped on 3 processors P_1 P_2 and P_4. The three processor-arbiters will be signalled to disable application A_2. Thus, even when actors of application A_2 are ready, they will not be executed, but actors behind them in the queue will be executed.

Suspension of an application is not to be confused with pre-emption. In our model, we do not allow actors to be preempted, but an application can be suspended after completing the execution of any actor. This is to limit the context that needs to be saved when an actor in the middle of its execution is to be stopped.

Communication Overhead

The overhead of monitoring and suspension is easy to compute. Consider ten applications running concurrently on a ten-processor system. Each application signals the RM every time it completes an iteration. Let us assume the period of each application is 100,000 clock cycles. Therefore, on average only one message is sent to the manager every 10,000 cycles. Let us consider the case with sampling period being 500,000 cycles. The RM messages at most every processing node every sampling interval. This is on average a message every 50,000 cycles, giving in total 6 messages every 50,000 cycles. If the length of each message is around 10 bytes, we get a bandwidth requirement of 60 bytes every 50,000 cycles. For a system operating at 100 MHz, this translates to about 120 Kb per second. In general, for N applications $A_0 \ldots A_{n-1}$ each with throughput T_{A_i} mapped on M processing nodes, with RM sampling at f_{samp} frequency, if each message is b bytes, the total communication bandwidth is given by following equation

$$BW_{reqd} = b \times \left(f_{samp}.M + \sum_{i=0}^{N-1} \frac{1}{T_{A_i}}\right) \qquad (2)$$

This is only the worst-case estimate. In practice, messages from the RM will not be sent to every processing node every sampling interval, but only when some application is suspended or resumed.

6. Performance Evaluation

We have developed a three-phase prototype tool-flow to automate the analysis of application examples. The first phase concerns specifying different applications (as SDF graphs), the processors of the MPSoC platform (including their scheduler type) and the mapping. For each application, desired throughput is specified together with the starting time for the application. After organizing the information in an XML specification for all three parts, a POOSL model [22] of the complete MPSoC system is generated automatically. The second phase relies on the POOSL simulator, which obtains performance estimations, like the application throughput and processor utilization. It also allows generation of trace files that are used in the final phase to generate schedule diagrams and graphs like those presented in this paper.

This section presents results of a case study regarding the mapping of H263 and JPEG decoder SDF models (described in [12] and [9] respectively) on a three-node MPSoC. Corresponding SDF graphs are shown in Figure 14. An FCFS scheduling policy was used in all the cases presented below. Table 4 shows the load on each processing node due to each application. The results were obtained after running the simulation for 100M cycles.

Figure 15 shows performance of the two applications when they are run in isolation on the platform and also when they are run concurrently. In this figure, the resource manager does not interfere at all, and the applications compete

[1] In the event of multiple output actors, any output actor may be chosen for this.

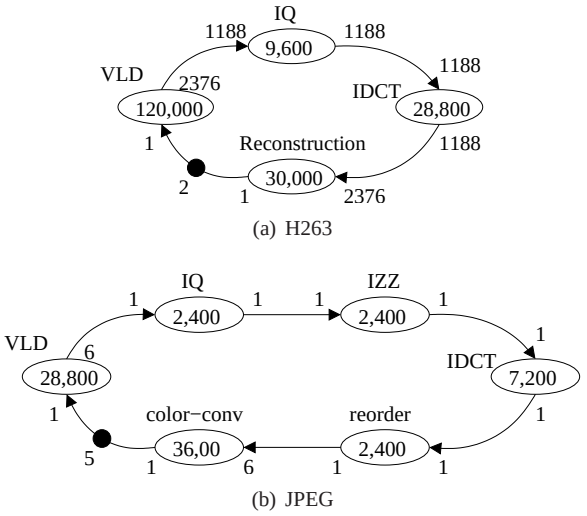


Figure 14. SDF Graphs of H263 and JPEG modeled from description in [12] and [9] respectively.

	H263	JPEG	Total
Proc 1	0.164	0.360	0.524
Proc 2	0.4	0.144	0.544
Proc 3	0.192	0.252	0.444
Total	0.756	0.756	1.512
Throughput Required	3.33e-6	5.00e-6	

Table 4. Load (in proportion to total available cycles) on processing nodes due to each application

with each other for resources. As can be seen, while the performance of H263 drops only marginally (depicted by the small arrow in the graph), a huge drop is observed in JPEG performance (big arrow in the graph). In fact, we see that even though the total load on each processing node is close to 50%, JPEG throughput is much lower than desired.

Figure 16 shows how a resource manager interferes and ensures that both are able to meet their minimum specified throughput. In this figure the resource manager checks every 5 million cycles whether applications are performing as desired. Every time it finds that either JPEG or H263 is performing below the desired throughput, it suspends the other application. Once the desired throughput is reached, the suspended application is re-activated. We observe that the RM effectively interleaves three infeasible schedules (JPEG Alone, H263 Alone, and H263/JPEG Together, in Fig. 15) that yields a feasible overall throughput for each application. (In *Alone*, only one application is active and therefore, those schedules are infeasible for the other application.)

Figure 17 shows applications' performance when the sample period of resource manager is reduced to 500,000 cycles. We observe that progress of applications is 'smoother' as compared to Figure 16. The 'transition phase'

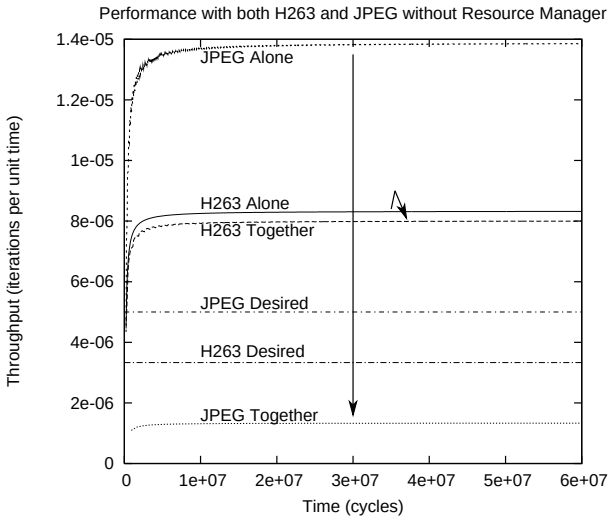


Figure 15. Progress of H263 and JPEG when they run on the same platform — in isolation and executing concurrently.

of the system is also shorter, and the applications soon settle into a 'long-term average throughput', and do not vary significantly from this average. This can be concluded from the almost horizontal curve of achieved throughput. It should be mentioned that this benefit comes at the cost of increasing monitoring from the resource manager, and extra overhead in reconfiguration (suspension and reactivation).

	Specified	No RM	RM sampling period		
			5,000k	2,500k	500k
JPEG	500	133	541	520	620
H263	333	800	554	574	504
Proc 1	0.524	1.00	0.83	0.85	0.90
Proc 2	0.544	0.56	0.71	0.71	0.72
Proc 3	0.444	0.46	0.55	0.55	0.56
Total	1.512	2.02	2.09	2.11	2.18

Table 5. Iteration count of applications and utilization of processors for different sampling periods for 100M cycles.

Table 5 shows the iteration count for each application specified, achieved without and with intervention from the RM. The first two columns clearly indicate that JPEG executes only about one-fourth of the required number of iterations, whereas H263 executes twice the required iteration count. The next three columns demonstrate the use of our RM to satisfy the required throughput for both the applications. The last row indicates that the utilization of resources increases with finer grain of control from the RM.

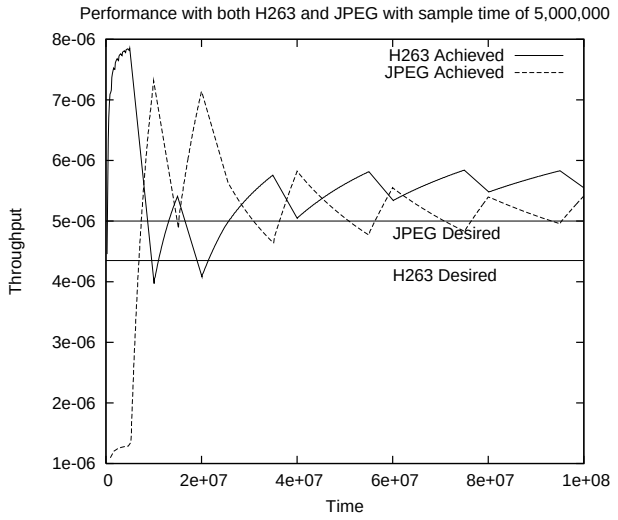


Figure 16. With a resource manager, the progress of applications is closer to desired performance.

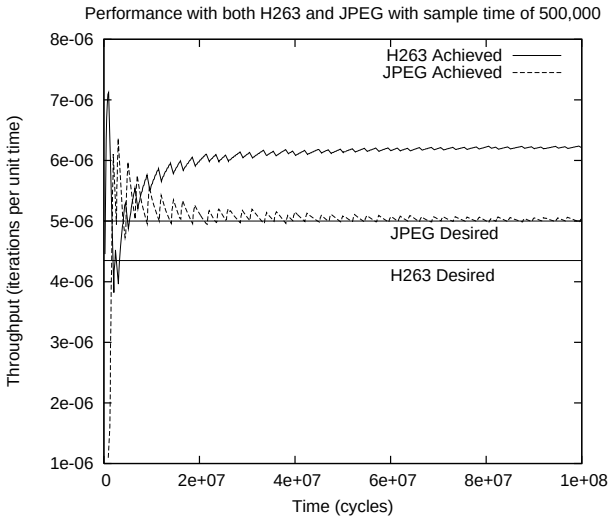


Figure 17. Increasing granularity of control makes the progress of applications smoother

7. Conclusions and Future Work

In this paper, we have introduced the composability problem and shown that combining resource usage is non-trivial. We also introduced properties and requirements for resource arbitration for a multi-processor based system-on-chip. Furthermore, using these requirements, we have compared two simple arbitration mechanisms. We observe that round-robin with skipping has lower design-time and run-time overhead, and also handles dynamism in the tasks more efficiently. When a new job arrives in the system, dynamic scheduling mechanisms have little overhead for reconfiguration. They however, suffer heavily from the lack of performance predictability in the design - one of the most important requirements for a resource manager in an MPSoC.

Further, we propose a resource manager (RM) for non-preemptive heterogeneous MPSoCs. Although the scheduling of these systems has been considered in the literature, the actual resource management in the context of concurrently executing applications is still unexplored area. Theoretically, design-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 [15]. Our resource manager shifts the burden of design-time analysis to run-time monitoring and intervention when necessary.

A high-level simulation model has been developed using POOSL methodology to realize RM. A case study with an H263 and a JPEG decoder demonstrates that RM intervention is essential to ensure that both applications are able to meet their throughput requirements. Further, a finer grain of control increases the utilization of processor resources, and leads to a more stable system.

Future research will focus on incorporating more intelligent prediction schemes for admission control. An FPGA implementation of an MPSoC is already realized, and we will use it to run our proposed RM on, to handle more realistic cases and to measure the overhead. Further, we would like to extend the model to include the communication and storage resources as well.

References

[1] N. Bambha, V. Kianzad, M. Khandelia, and S. S. Bhattacharyya. Intermediate representations for design automation of multiprocessor DSP systems. *Design Automation for Embedded Systems*, 7(4):307–323, 2002.

[2] S. Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors. *Real-Time Systems*, 32(1):9–20, 2006.

[3] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. An on line algorithm for real-time tasks allocation. *Algorithmica*, 15:600–625, 1996.

[4] M. Bekooij, R. Hoes, O. Moreira, P. Poplavko, M. Pastrnak, B. Mesman, J. D. Mol, S. Stuijk, V. Gheorghita, and J. van Meerbergen. Dataflow analysis for real-time embedded multiprocessor system design. In *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, pages 81–108. Springer, 2005.

[5] M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastrnak, and J. van Meerbergen. Predictable embedded multi-

processor system design. *Proceeding of the SCOPES workshop, September*, 2004.

[6] Y. Cai and M. C. Kong. Nonpreemptive scheduling of periodic tasks in uni- and multiprocessor systems. *Algorithmica*, 15(6):572–599, 1996.

[7] A. Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Trans. Des. Autom. Electron. Syst.*, 9(4):385–418, 2004.

[8] S. Davari and S. K. Dhall. An on line algorithm for real-time tasks allocation. *IEEE Real-time Systems Symposium*, pages 194–200, 1986.

[9] E. de Kock. Multiprocessor mapping of process networks: a JPEG decoding case study. In *Proceedings of 15th Intl. Symp. on System Synthesis, 2002.*, pages 68–73. IEEE Computer Society, 2002.

[10] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, B. Theelen, M. Mousavi, A. Moonen, and M. Bekooij. Throughput Analysis of Synchronous Data Flow Graphs. In *Sixth International Conference on Application of Concurrency to System Design, 2006. ACSD 2006.*, pages 25–36, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[11] K. Goossens, J. Dielissen, and A. Radulescu. Æthereal network on chip: concepts, architectures, and implementations. *IEEE Design and Test of Computers*, 22(5), 2005.

[12] R. Hoes. Predictable Dynamic Behavior in NoC-based MPSoC. Available from: www.es.ele.tue.nl/epicurus/, 2004.

[13] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of 12th IEEE Real-Time Systems Symposium*, pages 129–139, 1991.

[14] Karp, Richard M. and Miller, Raymond E. Properties of a model for parallel computations: Determinancy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, nov 1966.

[15] A. Kumar, B. Mesman, H. Corporaal, J. van Meerbergen, and Y. Ha. Global analysis of resource arbitration for mpsoc. In *Ninth Euromicro Conference on Digital System Design*. Euromicro, 2006.

[16] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous dataflow programs for digital signal processing. *IEEE Transactions on Computers*, Feb 1987.

[17] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[18] V. Nollet, P. Avasare, J.-Y. Mignolet, and D. Verkest. Low cost task migration initiation in a heterogeneous mp-soc. In *Proceedings of DATE '05*, pages 252–253. IEEE Computer Society, 2005.

[19] J. Pino and E. Lee. Hierarchical static scheduling of dataflow graphs onto multipleprocessors. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 4, pages 2643–2646, Detroit, MI, USA, 1995. IEEE.

[20] K. Richter, M. Jersak, and R. Ernst. A formal approach to MPSoC performance verification. *Computer*, 36(4):60–67, 2003.

[21] S. Siram and S. Bhattacharyya. *Embedded Multiprocessors; Scheduling and Synchronization*. Marcel Dekker, 2000.

[22] B. Theelen, O. Florescu, M. Geilen, J. Huang, P. van der Putten, and J. Voeten. Software/Hardware Engineering with the Parallel Object-Oriented Specification Langauge. In *Proceedings of the Fifth ACM-IEEE International Conference on Formal Methods and Models for Codesign*, pages 139–148, 2007.

[23] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings of ISCAS 2000 Geneva.*, volume 4, pages 101–104, Geneva, Switzerland, 2000. IEEE.

[24] W. Wolf. The future of multiprocessor systems-on-chips. In *Proceedings of the 41st DAC '04*, pages 681–685, 2004.

[25] H. Zhang and D. Ferrari. Rate-controlled static-priority queueing. In *INFOCOM '93. Proceedings.Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future. IEEE*, pages 227–236, San Francisco, CA, USA, 1993.