Global Analysis of Resource Arbitration for MPSoC

Akash Kumar, Bart Mesman, Henk Corporaal, Jef van Meerbergen
Eindhoven University of Technology
5600MB Eindhoven, The Netherlands
Email: a.kumar@tue.nl

Ha Yajun
National University of Singapore
10 Kent Ridge Crescent, Singapore

Abstract

Modern day applications require use of multi-processor systems for reasons of scalability and power efficiency. As more and more applications are integrated on a single device, mapping and analyzing them on a multi-processor system becomes a multi-dimensional problem. Each possible set of applications that can be active simultaneously 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 is very demanding and often infeasible.

In this paper, we highlight the issue of composability, i.e. being able to analyze applications in isolation while still reason about their overall behavior. We observe that arbitration plays an important role in this analysis. We compare two simple, yet commonly used arbitration mechanisms, and highlight the properties that are important for such analysis. We conclude that none of these arbitration mechanism is ideal for such an analysis and propose some variations to make them more suited for the analysis.

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 traditional systems, with a single general-purpose processor supporting pre-emption, the analysis of schedulability of task deadlines is well known [1] and widely used. In high-performance 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 containing the *programming* effort of these systems.

It is therefore expected in the electronic design community that future electronic systems re-use platforms that integrate many IP-blocks and memories. These platforms will concurrently execute many applications and (sub-)tasks. The number of possible *use-cases* is enormous. For example, in a modern television platform 60 applications are running in parallel, corresponding to an order of 2^{60} 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 currently 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.

Since we have 2^N (an exponential number) use-cases for N number of applications, even a design time analysis is infeasible. In addition, there is a high run-time overhead of storing schedules of all the use-cases. We would ideally want to analyze each sub-application in isolation, thereby reducing the analysis time to a linear function in N, 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 with the total number of sub-tasks in the system. The sub-task would then have exclusive access to its share of resources. For example, if we have 100 MHz processors and a total of 10 sub-tasks in the system, each sub-task would get 10 MHz of processing resource. The same can be done for communication bandwidth and memory requirements. However, 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 task are taken into account, even though these peak requirements of sub-tasks may not overlap.

Another way to reduce the complexity would be to analyze the sub-tasks in isolation with as little information from other sub-tasks 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*.

One of the problems which arises in the above approach is contention of resources and arbitration plays an important role in resolving it. The overall system behavior depends on the arbitration mechanism to a large extent. In this paper, we also compare the suitability of two very simple, yet often used, arbitration mechanisms for such an analysis. We will state requirements for their application in future media

platforms, and analyze these properties as much as possible using SDF [2]. SDF (Synchronous Data Flow) graphs is a class of *models of computation* that allows analysis of system 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.

It should be emphasized that this is an investigative paper into the composability problem and provides direction into how research would be carried out to study it further and provide optimal solutions. We assume the following for the scope of this paper.

- Multiprocessor: 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. The interrupt delay can not easily be ignored, and interrupts make the systems much less predictable in terms of timing behavior. We do not exclude preemption for all functions and processors, but we have to consider non-preemption in more detail.
- Efficient arbitration: The arbitration mechanism should be efficient, since the time required for arbitration has to match the grain of actor executions.

The two arbitration mechanisms considered in this paper are *static order* and *Round Robin* (with skipping).

- Static order: Actors - as defined in SDF model - are repeatedly executed in a strict order specified by a pre-defined list. If an actor is not ready (its input data has not yet arrived) to execute, the processor will halt and wait.
- Round Robin: Actors are repeatedly executed in an order specified by a pre-defined list. If an actor is not ready to execute, the arbiter will skip the actor and proceed to the next actor in the list.

In this paper, we find that none of the above arbitration mechanisms can be applied directly to composability analysis, and we provide a direction for future work that needs to be carried out in the direction.

In the Section 2 we shall first give an overview on composability. Section 3 will talk about providing guarantees in performance and resource utilization. The properties and requirements for arbitration will be discussed in Section . In Section 5 we shall present the conclusions and a direction of our future work. A comparison between the two arbitration mechanisms mentioned above will be made throughout the paper in all these sections.

2. 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 same time, such as listening to mp3 music, typing an sms and downloading some 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, for example, are deadlock analysis, throughput analysis and computing the static orders if needed. Clearly, since there are more than one jobs mapped on a multi-processor system, there is bound to be contention for the resources. Due to this contention, the throughput analyzed for a job in isolation might not be achievable when put together in the whole system. This will be demonstrated with the aid of an example in Section 2.2. In section 2.3 we show that in the case of static order, the schedule complexity (and therefore the program storage requirements) grows more than linearly with the number of mapped applications. In section 2.4 we consider the computational requirements for computing the timing behavior, and in section 2.5 we consider the smooth transition when a new job enters the system. First, however, we shall provide a short introduction to the modeling used for analysis in this paper, namely SDF in Section 2.1.

2.1. SDF Modeling

Various data flow models have been proposed in literature to model real applications. In this paper, we shall focus on SDF - Synchronous Data Flow - model proposed by Lee and Messerschmitt [2]. As in a typical data flow graph, a directed edge represents the dependency between tasks. Tasks also has needs some input data (or control information) before they can start and usually also produce some output data; such information is referred to as*tokens*. The number of tokens consumed and produced by an actor is indicated on the edge, as shown in Figure 1. In an actual implementation edges indicate buffers in physical memory. The edges may also contain *initial tokens* which denote the data dependencies across various iterations of the algorithm. These are indicated by bullets on the edges.

Figure 1 shows an example of a simple SDF graph. There are four actors in this graph. An actor can only start execution (also called firing) when the required number of tokens are present on each of its incoming edge. As can be seen from the graph, 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 the produce 4 tokens on the edge to C. Another thing to note is that since there are two initial

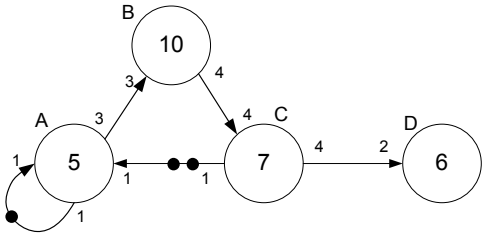


Figure 1. Example of a simple SDF graph

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. Thus, at any point of time, there can be two iterations of the graph active at any point of time.

SDF graphs allow you to analyze maximum achievable throughput of a system by computing the MCM (Maximum Cycle Mean) [3]. Further they can also allow us to obtain the optimal order (i.e. the schedule) which will result in that throughput (for a given buffer size). The only cycle in the graph shown in Figure 1 has an execution time of 22 cycles - A, B and C. Since there are two tokens in this cycle, the MCM is 22/2 = 11. 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.

2.2. Composability Problem

Figure 2 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 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, thereby needing a maximum overall utilization of 67%. However, due to the dependencies, the maximum achievable processor utilization is only 50%.

Figure 2 also shows a schedule obtained when the actors are scheduled using round robin with skipping. 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 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 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.

As can be seen from the above example, simply adding

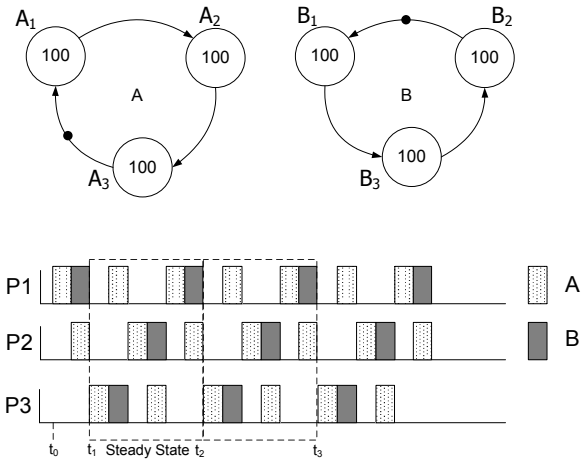


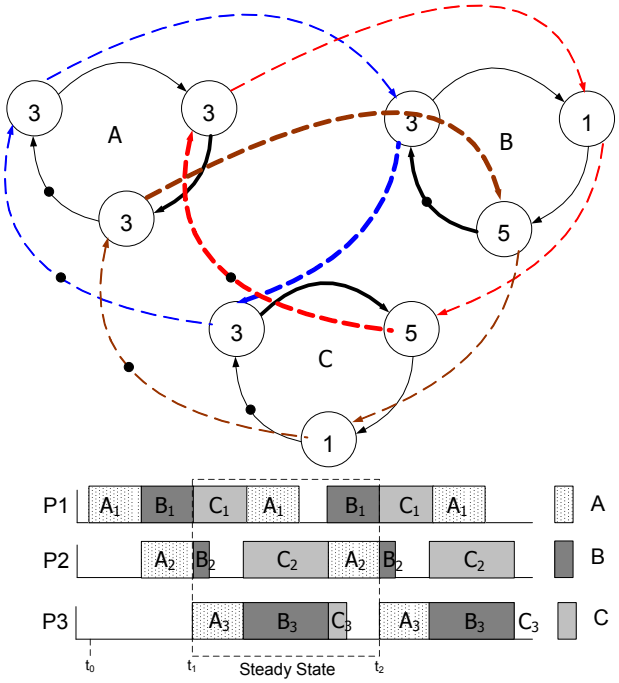
Figure 2. An example showing why composability needs to be examined. Individually each task takes 300 cycles to complete an iteration and requires only 33% of processor resources. However, when another job enters in the system, it is not possible to schedule both of them with their optimal schedule of 300 cycles, even though the total request for a node is only 67%.

up computational load of a processor is not realistic. Composability, therefore, is not a black or white issue, since arbitration can cause interference between job executions to different degrees.

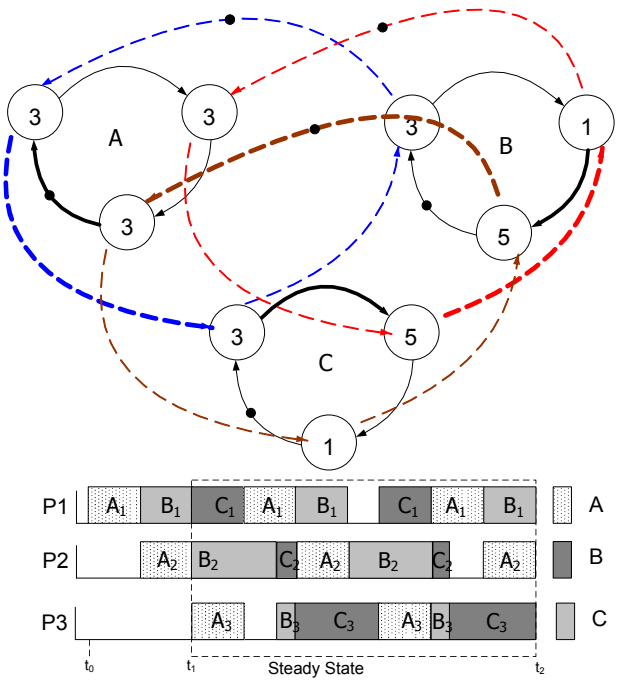
2.3. Overhead of Multiple Use-cases

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 use-cases that have to be considered rises exponentially. For a system with 10 possible applications in which up to 4 tasks are allowed to be active at the same time, there are approximately 400 possible combinations - and it grows exponentially as we increase the number of sub-tasks simultaneously active. 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. As such the scalability of using static order for scheduling multiple jobs is limited.

In round-robin scheduling on the other hand, the easiest approach would be to store each actor in the schedule. When a task is not active, the actors in it are simply skipped, without causing any trouble for the scheduling kernel. Thus, in some way, a super-set i.e. set of all the actors that can be ever active on a particular processing node, can be stored in a list and that is the one that is used for scheduling. It should be emphasized 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. The performance of this strategy as compared to an ordered transaction strategy is discussed in Section 3.



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



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

Figure 4. 3 individual task graphs give different MCM when scheduled differently. The respective cycles which gives this MCM are shown in bold.

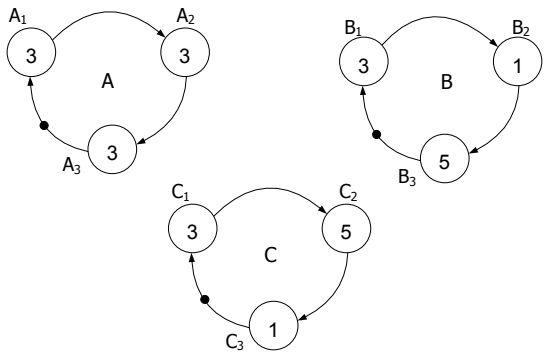


Figure 3. Example of a system with 3 different tasks.

2.4. Computing Static Orders

Three task graphs - A, B and C are shown in Figure 3. 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 4. 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 combine different graphs into one big HSDF for complete analysis. Figure 4(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 be only 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.

When MCM (Maximum Cycle Mean) analysis is done for this complete graph, we obtain a mean cycle count of 11 [3]. This also gives us the ideal order for each processing node. The bold arrows represent the edges used to compute MCM. The schedule for the graph 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 cycles. The period is denoted in the graph between the time instant t_1 and t_2.

Figure 4(b) shows just another of the many possibilities for ordering the actors of the complete HSDF. Interestingly, the MCM for this graph is 10, as indicated by the bold arrows. The corresponding schedule for the graph is also shown. In this case, the period is longer as indicated by difference in time instants t_1 and t_2 i.e. 20 cycles. However, since two iterations for each task are completed, the actual MCM is only 10 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 cycles before task A needs processor 1

Table 1. Table showing a deadlock condition

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

again. Also, since B and C each takes only 3 cycles, 6 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 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.

As can be seen, there are many possibilities for constructing the HSDF from individual graphs. In fact, if one tries to combine g graphs of say a actors each, there happen to be $((g-1)!)^a$ unique combinations, each with a different actor ordering. To give an example, if there are 5 graphs with 10 actors each we get 24^{10} or close to $6.34e13$ graphs. MCM computation of an HSDF already takes exponential time (in the number of actors) as has been analyzed in [5]. If each computation takes 1ms to compute, 2009 years are needed to evaluate all the 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 more actor copies. An exhaustive search through all the graphs is, therefore, not an option. Thus, a simpler algorithm for arbitration is needed with lesser design overhead.

2.5. Deadlock

Deadlock avoidance and detection is an important concern when tasks arrive 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 5.

Say task A and B are running in the system on processor node 2 and 3 respectively. Further assume that the static scheduling 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 enters the system, it gets processor 1 since that is idle. Lets see what happens to processor 2. Task A is running on it and it is then assigned to task B. Processor 3 is assigned to C after B is done. Thus, after each task is finished executing on its currently assigned processor, we obtain A waiting for processor 3 that is assigned to task C, task B waiting for processor 1 which is assigned to A, and task C waiting for processor 2, which is assigned to B. This can be expressed by Table 1.

Looking at Figure 5, 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.

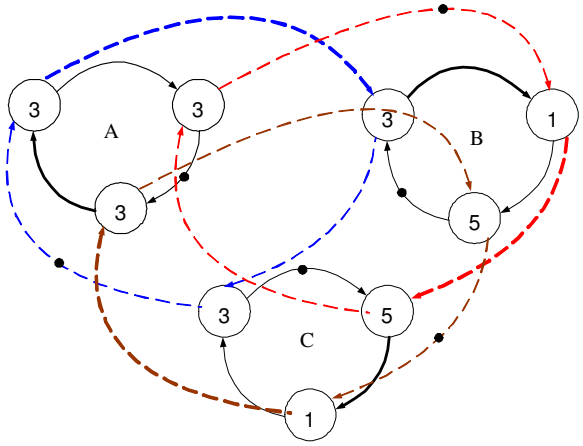


Figure 5. Deadlock situation when a new job, C arrives in the system. A cycle $A_1, B_1, B_2, C_2, C_3, A_3, A_1$ is created without any token in it.

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 [6]. This cycle is shown in Figure 5 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 having round robin strategy with skipping. When the system enters into a deadlock, the round-robin assignment 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 1. In addition, a task can be inserted at any point of time without worrying about deadlock. In this approach, there can never be a deadlock due to dependency on processing resources.

3 Performance Guarantees

Providing realistic performance guarantees is critical when it comes to real-time tasks with deadlines. Figure 6 shows a simple example with two task graphs which share computing resources $P1$ and $P2$. As shown in the figure, each actor takes one cycle of resources. Figure 7 shows how a static order would look like. Extra dashed arrows have been added in the graph to denote arbitration of each of the processors with relevant initial tokens. All the cycles in the graph have an MCM of 2. Thus, every two cycles one iteration of both A and B is done. The corresponding schedule is also shown in the figure. It can be observed that the processor utilization is 100% for both the processors, once the periodic schedule is obtained. This implies the schedule is optimal and can not be improved further.

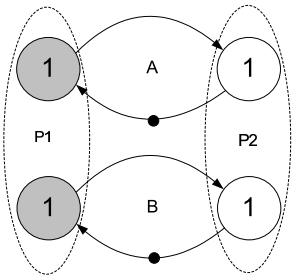


Figure 6. An example with two task graphs to be scheduled. The actors compete for resources P_1 and P_2 as shown in the figure.

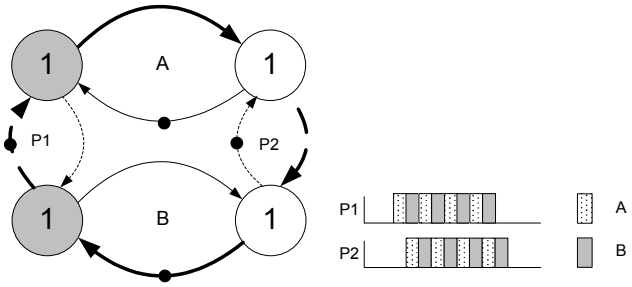


Figure 7. Computing MCM for a combined SDF. A cycle with two tokens is shown leading to an MCM of 2.

For Round Robin with skipping, the maximum waiting time for a particular actor, can be computed by considering the *critical instant* as defined by Liu and Layland [1]. The critical instant for a task is defined 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 Round Robin, it will be when an actor is ready just after being checked by the scheduler, and all the other actors in the list being ready. Thus, the total waiting time is equal to the sum of processing times of all the other actors on that particular node as given in Eqn. 1.

$$W(T_{ij}) = \sum_{k=1,k\neq i}^{m} ET(T_{kj}) \tag{1}$$

Here $ET(T_{ij})$ denotes the execution time of actor T_{ij}, i.e. actor of task T_i mapped on processor j. Thus, we obtain the maximum waiting time for both actors of task A a_{11} and a_{12} as 1. The same has been modeled in Figure 8. This method allows us to analyze the SDF graphs in isolation and get an upper-bound on the waiting times of a particular actor. One of the major drawbacks of this mechanism is that the bound is often pessimistic as can be seen from this example. Analyzing task A in isolation - but including the waiting times as shown in Figure 8 - on processors $P1$ and $P2$ results in an MCM of 4 cycles. This, as seen from the static schedule in Figure 7, is not a realistic case.

In some cases, however, this happens to be indeed the

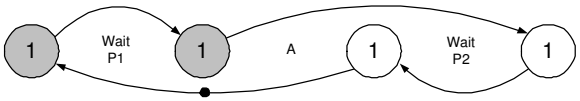


Figure 8. Modeling waiting times using SDF. Extra actors have been added for Task A to model the waiting times due to resource conflicts on both the processor nodes.

realistic bound. We now revisit the example shown in Figure 2 before. The worst-case waiting time for each actor is 100 cycles as well, as computed from Equation 1. The schedule (as mentioned earlier) is an extreme case in which A wins every arbitration, when both A and B are ready at the same time. (We can also assume that A finished a cycle too early, thereby winning arbitration every time.) Thus, while a cycle of A takes only 300 cycles (the minimum possible, since it does not have to wait), a cycle of B takes 600 cycles (the maximum possible including the waiting time).

An interesting thing that can be observed from this example that the starting state does not always have an effect on the steady-state. Even if we let B start first, the same periodic schedule is obtained (assuming of course that A wins the arbitration).

There can be other approaches to remove such possible bias. It is trivial to see that a Round Robin approach *without* skipping would actually in this case give equal rates of execution to A and B in the long run. A static-order which gives priority to A and B alternately for processors that have a conflict will also help solve the situation as shown in Figure 9. In this schedule, we find that one iteration of both A and B takes only 400 cycles.

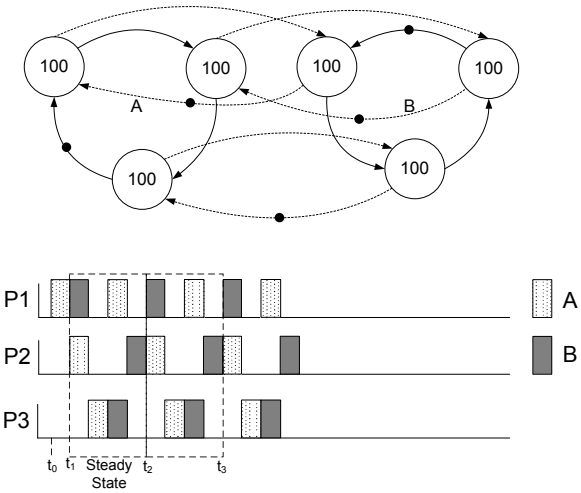


Figure 9. An example showing a static order strategy. The same is obtained for round robin without skipping.

Interestingly, in either of the cases (Figures 2 and 9) the *processor utilization* comes out to be the same. Processor utilization is defined as the time during which the processor node is doing some computation as compared to the total

time available. In Figure 2 each processor is active for exactly half the time in the periodic schedule. Therefore, the utilization for each processor node is 0.5. The same goes for Figure 9, in which each node is active for 200 cycles out of 400 cycles. Alternatively, we can also compute processor utilization using Equation 2.

$$U(P_j) = \sum_{k=1}^{m} \frac{ET(T_{kj})}{T_k} \quad (2)$$

Here T_k refers to the period of task k. In schedule shown in Figure 2, the period for task A and B is 300 and 600 cycles respectively. Utilization for each processor is therefore, 100/300 + 100/600 = 0.5. For schedule in Figure 9, the period for each task is 400 cycles. We therefore obtain the utilization as 100/400 + 100/400 = 0.5.

The problem of composability becomes most obvious in this example. We have two tasks each with an MCM of 300 cycles, each requiring only a-third of each processor resources, thereby giving a total of 67% of processor utilization. In spite of this, we are unable to find any schedule that can ensure that each task completes one iteration in only 300 cycles. Even if we allow pre-emption, achieving a processor utilization of more than 50% is not possible.

3.1. Estimating Resources

It is quite useful to be able to estimate resource requirement early in the design phase. Design managers often have to negotiate with the product divisions for the overall resources needed for the system. These estimates are mostly on a higher level, and the managers usually like to adopt a 'spread-sheet' approach for computing it. As we shall see, it is often not possible to use this view. We show how different approaches lead to differing estimates.

Consider again the example as shown in Figure 2. There could be many ways to consider how many resources we have available, and how much of it can be utilized. Table 2 shows how different estimating strategies can lead to different results. Some of the methods give a false indication of processing power, and are not achievable. For example, in the second column the execution time of only the actor is considered. This is a very naive approach and would be the easiest to estimate. It assumes that all the processing power that is available for each node is shared between the two processes equally. Each actor takes 100 cycles of processing power, and there are two actors on each node. Therefore, in a time period of 1 million cycles, we should be able to run each actor 5,000 times assuming uniform distribution of resource, thereby running each task 5,000 times as well. This, however, is not achievable in practice due to intra-task dependency. When, this intra-task dependency is taken into account, we see that the maximum number of iterations for each task can only be 3,333, since each iteration will take at least 300 cycles and only one iteration of each task can be active at any point in time (only one initial token is present).

Table 2. Accounting of resource utilization: Iterations for each task for 1,000,000 cycles

Task	Only actors	Indiv. graph	WC Analysis (both graphs)	Static	RR A pref	RR B pref
Task A	5,000	3,333	1,666	2,500	3,333	1,667
Task B	5,000	3,334	1,667	2,500	1,667	3,333
Total	10,000	6,667	3,333	5,000	5,000	5,000

The next case is to take the worst-case waiting time for a strategy like round robin into account and estimate the number of iterations possible. As mentioned in Equation 1, we get a worst case execution time for one iteration of the task as 600 cycles. The next column gives the number of executions as obtained by a static order schedule. The next two columns show the iterations for a round robin strategy which favors A and B respectively. Interestingly, the total number of iterations for a static and for a round-robin with skipping and of static order happen to be same. The distribution in the round robin (with skipping) case however, is more unpredictable so its hard to guarantee the number of individual iterations, while in the static order it is deterministic.

3.2. Suitability for Resource Management

With the increasing dynamism in modern applications, the need for a separate task to monitor and direct the usage of resources has arisen. 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.

Predictability of an arbitration mechanism is one of the most important criteria when it comes to suitability for such a task. Computing static orders is certainly more predictable than using a round robin strategy for scheduling tasks. Given a particular use case, the resource manager would like to know the exact usage of resources before and after accommodating the new job. A round robin mechanism can not provide such guarantees. However, as mentioned earlier this guarantee comes at a high design and run-time cost.

A round robin strategy allows for more dynamism in the overall system since specific use cases do not have to be considered. Further, deadlock issues do not have to considered in this strategy either. Any new job in the system can be added when it arrives, unlike in a static order when deadlock has to be specifically avoided by extra heuristics in the system.

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-

Table 3. Summary of all the performance parameters

Properties		Static Sched.	RR with skipping
Design time overhead	Computing order	- -	++
Run-time overhead	Memory reqmt.	-	++
	Scheduler	+	+
Predictability	Throughput	++	- -
	Resource Utilization	+	-
New job admission	Admission criteria	++	- -
	Deadlock	-	++
	Reconfigurability	-	+
Dynamism	Variable Exec. time	-	+
	New Use-case	- -	++

time scheduler needed for both schedulers is quite simple, since only a simple check is needed to see when the actor is active and fire. The memory requirement for static scheduling is however, higher than that for a round-robin mechanism, since all the use-case orders need to be stored. The static order certainly scores better than round-robin 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 2.5.

A round-robin 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 round-robin satisfies 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. Such enforcing mechanism suffers from a dilemma of wasting processing resources while waiting for an actor. For example, if an actor T_{ij} has used all of its allocated resources and T_{kj} is next in queue. T_{kj}, however, is not yet ready while T_{ij} needs more of resource P_j. Deciding whether to wait for T_{kj} or let T_{ij} execute is non-trivial. Ideally speaking we would like to be able to look into the future and know when the actor would be ready to use its available resource. Since we are dealing with multi-processor system, this implies looking into the state of other nodes, which involves huge overhead and is often infeasible.

Secondly, assigning priorities to the actors scheduled on a node can also be a possible extension to round-robin, to limit the waiting time for an actor. This could perhaps help us analyze the performance in round-robin better and thereby allow us to provide more realistic guarantees for resource requirement and utilization.

Another approach might be to use a budget-based method where an actor is given some credits which decrements depending on how much resources it uses. When the credits fall below a certain number or become negative, the actor is flagged off and not allowed to execute. The credits can be handed out by the resource manager and decremented by individual processing nodes. The initial credit limit - budget - and the decrement sets the grain of control. This, however, denotes a higher run-time overhead.

The above methods are primarily using round-robin as a base and introducing techniques to improve predictability. We could also start from the static-order approach and introduce measures to reduce the design-time overhead and the ability to handle dynamism more gracefully.

5. 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 has a lower run-time and design-time overhead, and also handles dynamism in the tasks more efficiently. When a new job arrives in the system, round-robin has little overhead for reconfiguration. It however, suffers, heavily from the lack of performance predictability in the design - one of the most important requirements for a resource manager in an MPSoC. We would like to use round-robin as the basic arbitration mechanism and build upon it in order to realize a resource manager. This implies firstly, a better performance estimate and secondly, a mechanism for the resource manager to impose specified utilization or time-budget. Further, we would like to evaluate the arbitration mechanism on real hardware implementation. An FPGA infrastructure has already been developed in order to make architectural explorations and do quick design iterations.

References

[1] C. L. Liu and James W. Layland; *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*, Journal of the ACM, Vol. 20, No. 1, 1973.

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

[3] S Sriram and S. S. Bhattacharya; *Embedded Multiprocessors: Scheduling and Synchronization*, Marcel Dekker Inc 2000.

[4] Rob Hoes; *Predictable Dynamic Behavior in NoC-based Multiprocessor Systems-on-Chip*, Masters Thesis, 2004. http://www.es.ele.tue.nl/epicurus/publications.php.

[5] A. Dasdan; *Experimental analysis of the fastest optimum cycle ratio and mean algorithm*, ACM Transactions on Design Automation of Electronic Systems, 2004.

[6] R. M. Karp et al; *Properties of a model for parallel computations, determinacy, termination, and queueing*, SIAM Journal of Applied Mathematics, 14(6):1390–1411, Nov. 1966.