technische universiteit eindhoven

A Probabilistic Approach to Model Resource Contention for Performance Estimation of Multi-featured Media Devices

Akash Kumar, Bart Mesman, Henk Corporaal, Bart Theelen, Yajun Ha

ES Reports
ISSN 1574-9517

ESR-2007-02
25 March 2007

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems

http://www.es.ele.tue.nl/esreports
esreports@es.ele.tue.nl

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems
PO Box 513
NL-5600 MB Eindhoven
The Netherlands

A Probabilistic Approach to Model Resource Contention for Performance Estimation of Multi-featured Media Devices

Akash Kumar[1,2], Bart Mesman[1], Henk Corporaal[1], Bart Theelen[1] and Yajun Ha[2]
[1]Eindhoven University of Technology, Eindhoven, The Netherlands
[2]ECE Department, National University of Singapore, Singapore
Email: a.kumar@tue.nl

ABSTRACT

The number of features that are supported in modern multimedia devices is increasing faster than ever. Estimating the performance of such applications when they are running on shared resources is becoming increasingly complex. Simulation of all possible use-cases is very time-consuming and often undesirable. In this paper, a new technique is proposed based on probabilistically estimating the performance of concurrently executing applications that share resources. Two different methods of employing this approach are presented and compared with state-of-the-art technique, and with achieved performance found through extensive simulations. The results are within 15% of simulation result (considered as reference case) and up to ten times better than a worst-case estimation approach. The approach scales very well with increasing number of applications, and can also be applied at run-time for admission control.

Categories and Subject Descriptors: C.4 [Performance of Systems]: Modeling techniques

General Terms: Algorithms, Performance, Theory

Keywords: Performance Estimation, Probability, Resource sharing

1. INTRODUCTION

Modern multi-media systems show a need of integrating a (potentially large) number of applications on a single device. Further, these systems are becoming increasingly heterogeneous with the use of dedicated IP blocks and application domain specific processors. To achieve high performance in such systems, the limited computational resources must be shared. The concurrent execution of dynamic applications on shared resources is a potential source of interference. Modeling and analyzing this interference is a key to building cost-effective systems which can deliver the desired performance of the applications.

This 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.

Synchronous Data Flow Graphs (SDFGs, see [9]) are often used for modeling modern DSP applications [13] and for designing concurrent multimedia applications implemented on multi-processor system-on-chip. Both pipelined streaming and cyclic dependencies between tasks can be easily modeled in SDFGs. Tasks are modeled by the vertices of an SDFG, which are called actors. SDFGs allow one to analyze a system in terms of throughput and other performance properties, e.g. latency, buffer requirements [15, 19]. However, when the number of applications increases, the analysis becomes overly pessimistic or computationally infeasible [7].

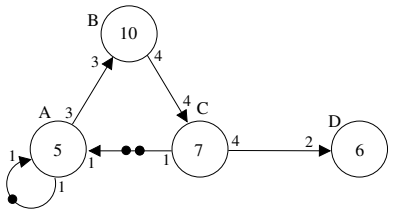

Figure 1: Example of an SDF Graph

Figure 1 shows an example of an SDF Graph. There are four actors in this graph. An actor can only start execution when the required number of tokens are present on each of its incoming edges, and upon completion, the actor produces the number of tokens specified on the outgoing edge.

Our Contribution. In this paper, we propose an estimation technique to compute application throughput taking resource contention into account. The main strength of this approach is the limited information that is needed from the other applications, thereby making it scalable as the number of applications increases. Our technique estimates delay using probability of a resource being blocked by actors. Two methods have been outlined for computing the effects of multiple actors mapped on a single processing node. The formulae for computing overall delay for both methods are presented. Their computational complexity is analyzed, and they are further optimized to allow for efficient implementation. Simulations with ten random application graphs that mimic DSP or multimedia applications were carried out. For over a thousand use-cases, throughput obtained using our estimation approach was compared with the throughput obtained through simulation. The results of

This report is an extended version of the paper "A. Kumar, B. Mesman, H. Corporaal, B.D.Theelen and Y. Ha. A Probabilistic Approach to Model Resource Contention for Performance Estimation of Multi-featured Media Devices. In 44th Design Automation Conference, DAC 2007, Proc ACM 2007". The report provides proofs omitted from the original paper.

this comparison are also presented in the paper. Further, our analysis takes about three minutes for each approach as compared to about 23 hours of simulation, and yet provides accurate estimates.

The rest of the paper is organized as follows. Section 2 discusses the related work in the area. Section 3 describes the probabilistic approach, and Section 4 discusses how the complexity of the approach can be reduced. Section 5 discusses the results of the proposed technique, and finally, Section 6 presents the conclusions.

2. RELATED WORK

In [2], the authors propose to analyze performance of a *single application* modeled as an SDFG mapped on a multiprocessor system by decomposing it into an HSDFG [13], and modeling dependencies of resources by adding extra edges on the nodes. This can result in an exponential number of vertices [11], after which the throughput is calculated based on analysing each cycle in the HSDFG [4]. Algorithms that have a polynomial complexity for HSDFGs, therefore have an exponential complexity for SDFGs. Algorithms have been proposed to reduce average case execution [5], 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 dependency is exponential [7]. 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 [5]. Further, only static order arbitration can be modeled using this technique while the best performance of SDFG applications is obtained when actors are allowed to execute with least contention on their own [13]. Our approach allows for that behavior since no ordering is imposed.

For *multiple applications*, an approach that models resource contention by computing *worst-case-response-time* for TDMA scheduling (requires preemption) has been analyzed in [3]. This analysis also requires limited information from the other SDFGs, but gives a very conservative bound. The analysis can be very pessimistic. As the number of applications increases, the bound increases much more than the average case performance. Further, this approach requires preemption for analysis. A similar worst-case analysis approach for round-robin is presented in [6], which also works on non-preemptive systems, but suffers from the same problem of lack of scalability. Real-time calculus has also been used to provide worst-case bounds for multiple applications [12, 18, 8]. Besides providing a very pessimistic bound, the analysis is also very intensive and requires a very high design time effort. Our approach on the other hand is very simple. However, we should note that above approaches give a worst-case bound that is targeted at hard-real-time (RT) systems, while our estimation approach is aimed at designing soft-RT systems.

A common way to use probabilities for modeling dynamism in application is using stochastic task execution times [1, 10]. In our case, however, we use probabilities to model the resource contention and provide estimates for the throughput of applications. This approach is orthogonal to the approach of using stochastic task execution times. In our approach we assume fixed execution time, though it is easy to extend this to varying task execution times as well. To the best of our knowledge, there is no efficient approach of analyzing multiple soft-RT applications on a non-preemptive heterogeneous multi-processor platform.

3. ANALYZING CONTENTION

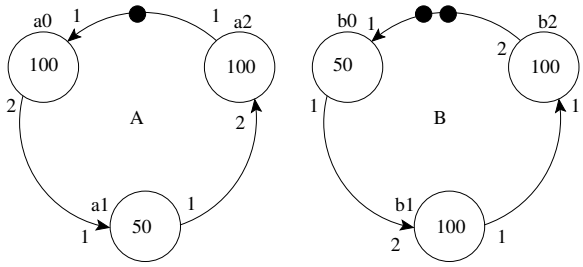

Figure 2: Two application SDFGs A and B

In this section we explain how the contention between actors for processing resources is analyzed. We start the analysis by defining some terms.

Definition 1. (Actor Execution Time) *Actor execution time, $\tau(a)$ is defined as the time needed to complete execution of actor a on a specified node. $\tau(a_0) = 100$, for example, in Figure 2.*

Definition 2. (Repetition Vector) *Repetition Vector q of an SDFG A is defined as the vector specifying the number of times an actor in A is executed for one iteration of A. For example, in Figure 2, q[a0 a1 a2] = [1 2 1] and q[b0 b1 b2] = [2 1 1].*

Definition 3. (Application Period) *Application Period $Per(A)$ is defined as the time SDFG A takes to complete one iteration on average. $Per(A) = 300$ in Figure 2. (Note that actor a1 has to execute twice.) This is also equivalent to the inverse of throughput. An application with a throughput of 50 Hz takes 20 ms to complete one iteration.*

We now refer to SDFGs A and B in Figure 2. Say a_0 and b_0 are mapped on a processor $Proc_0$ and others have dedicated resources. a_0 is active for time $\tau(a_0)$ every $Per(A)$ time units (since its repetition entry is 1). In the example shown above, $\tau(a_0) = 100$ time units and $Per(A) = 300$ time units, which can be computed using MCM analysis techniques [4] or state-space exploration technique [5].

The probability that $Proc_0$ is used by a_0 at any given time is $\frac{100}{300} = \frac{1}{3}$, since a_0 is active for 100 cycles out of every 300 cycles. Since arrival of a_0 and b_0 are independent, this is also the probability of $Proc_0$ being occupied when b_0 arrives at it. Further, since b_0 can arrive at any arbitrary point during execution of a_0, the time a_0 takes to finish after b_0 arrives on the node is uniformly distributed from $[0, 100]$. Therefore, b_0 has to wait for 50 time units on average if $Proc_0$ is found blocked. Since the probability that the resource is occupied is $\frac{1}{3}$, the average time actor b_0 has to wait is given by $\frac{50}{3} \approx 17$ time units. The response time (defined as sum of execution time and waiting time) of b_0 will therefore be ≈ 67 time units.

3.1 Generalizing the Analysis

This sub-section generalizes the analysis presented before by means of an example. As we can see in the above analysis, each actor has two attributes associated with it: 1) the

probability that it blocks the resource and 2) the average time it takes before freeing up the resource it is blocking. In view of this we define the following terms before proceeding:

DEFINITION 4. (BLOCKING PROBABILITY) *Blocking Probability, $P(a)$ is defined as the probability that actor a of application A blocks the resource it is mapped on. $P(a) = \tau(a).q(a)/Per(A)$. $P(a_0) = \frac{1}{3}$ in Figure 2. $P(a)$ is also represented as P_a interchangeably.*

DEFINITION 5. (AVERAGE BLOCKING TIME) *Average Blocking Time, $\mu(a)$ is defined as the average time before the resource blocked by actor a is freed given the resource is found to be blocked. Again, $\mu(a)$ is also represented as μ_a interchangeably. $\mu(a) = \tau(a)/2$ for constant execution time. In Figure 2, $\mu(a_0) = 50$.*

If X is defined as the random variable that denotes how long an actor b has to wait if the resource it is requesting is being blocked by actor a, the probability density function, $w(x)$ of X can be defined as follows.

$$w(x) = \begin{cases} 0, & x \leq 0 \\ \frac{1}{\tau(a)}, & 0 < x \leq \tau(a) \\ 0, & x > \tau(a) \end{cases} \quad (1)$$

The average time b has to wait, or μ_a is therefore,

$$\begin{aligned} t_{wait}(b) = \mu_a &= \int_{-\infty}^{\infty} x\, w(x)\, dx \\ &= \int_0^{\tau(a)} x\, \frac{1}{\tau(a)}\, dx \\ &= \frac{1}{\tau(a)} \left[\frac{x^2}{2}\right]_0^{\tau(a)} \\ &= \frac{\tau(a)}{2} \end{aligned} \quad (2)$$

Let us revisit our example in Figure 2. Let us now assume actors a_i and b_i are mapped on $Proc_i$ for i = 0, 1, 2. The blocking probabilities for actors a_i and b_i for i = 0, 1, 2 is

$$P(a_i) = \frac{\tau(a_i).q(a_i)}{Per(A)} = \frac{1}{3} \text{ for } i = 0, 1, 2.$$
$$P(b_i) = \frac{\tau(b_i).q(b_i)}{Per(B)} = \frac{1}{3} \text{ for } i = 0, 1, 2.$$

The average blocking time of actors in Figure 2 is

$$[\mu_{a_0}\ \mu_{a_1}\ \mu_{a_2}] = [50\ 25\ 50] \text{ and } [\mu_{b_0}\ \mu_{b_1}\ \mu_{b_2}] = [25\ 50\ 50]$$

In this case, since only one other actor is mapped on every node, the waiting time for each actor is easily derived.

$$t_{wait}(b_i) = \mu(a_i).P(a_i) \text{ and } t_{wait}(a_i) = \mu(b_i).P(b_i)$$
$$t_{wait}[b_0\ b_1\ b_2] = [\frac{50}{3}\ \frac{25}{3}\ \frac{50}{3}] \text{ and } t_{wait}[a_0\ a_1\ a_2] = [\frac{25}{3}\ \frac{50}{3}\ \frac{50}{3}]$$

Figure 3 shows the response time of all actors taking waiting times into account. The new period of SDFG A and B is computed as 359 time units for both. Clearly, the period that these application graphs would achieve in practice is only 300 time units. However, it must be noted that in our entire analysis we have ignored the inter-graph actor dependency. For example, if the cyclic dependency of SDFG

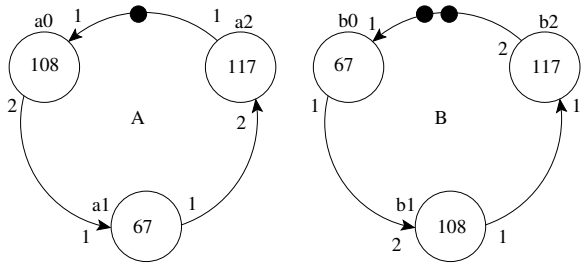

Figure 3: SDFGs A and B with response times

1: a_{ij} is actor j of application A_i
2: **for all** actors a_{ij} **do**
3: $P(a_{ij})$ = GetBlockingProb($\tau(a_{ij})$, $q(a_{ij})$, $Per(A_i)$)
4: **end for**
5: //Now use this to compute waiting time
6: **for all** Applications A_i **do**
7: **for all** Actors a_{ij} of A_i **do**
8: $t_{wait}(a_{ij})$ = ComputeWaitingTime(τ, P)
9: $\tau(a_{ij}) = \tau(a_{ij}) + t_{wait}(a_{ij})$
10: **end for**
11: $Per(A_i)$ = ComputeNewPeriod(A_i)
12: **end for**

Figure 4: Algorithm for estimating Period using blocking probabilities

B was changed to clockwise, all the values computed above would remain the same while the period of the graphs would change. The new period as measured through simulation is 400 time units. The probabilistic estimate we have now obtained in this simple graph is roughly equal to the mean of period obtained in either of the cases.

Further, in this analysis we have assumed that arrival of actors on a node is independent. In practice, this assumption is not always valid. Resource contention will inevitably make the independent actors dependent on each other. The assumption becomes even weaker, when actors within an application are considered. Even so, the approach we have works well, as we see in Section 5.

A rough sketch of the algorithm used in our approach is outlined in Figure 4. Blocking probability of all the actors is first computed (Steps 2-4). This is used to compute the waiting time for each actor which is then added to its own execution time (Steps 7-10). The function *ComputeWaitingTime* takes blocking probability and execution time of all the other actors mapped on the same node. The new execution times are used to compute new period of the application (Step 11). The procedure is done for each application.

Computing the waiting time (Step 8) when there is only one other actor mapped on the node is rather trivial as demonstrated in the example. In the next subsection, we extend the same for an arbitrary number of actors.

3.2 Extending to N Actors

Let us assume actors a, b and c are mapped on the same node, and that we need to compute the waiting time for c. c may be blocked by either a or b or both. Analyzing the case of c being blocked by both a and b is slightly more

complicated. There are two sub-cases for it - one in which a is being served and b is queued, and another in which b is being served and a is queued. We therefore have four possible cases.

Blocking only by a

$$t_{wait}(c_1) = \mu_a.P_a.(1 - P_b)$$

Blocking only by b

$$t_{wait}(c_2) = \mu_b.P_b.(1 - P_a)$$

a being served, b queued

$$t_{wait}(c_3) = \frac{1}{2}.P_a.P_b.(\frac{\tau(a)}{2} + \tau(b)) = \frac{1}{2}.P_a.P_b.(\mu_a + 2\mu_b)$$

b being served, a queued

$$t_{wait}(c_4) = \frac{1}{2}.P_a.P_b.(\frac{\tau(b)}{2} + \tau(a)) = \frac{1}{2}.P_a.P_b.(2\mu_a + \mu_b)$$

The total probability of both actors requesting the resource when c arrives is $P_a.P_b$. The individual probability is taken as half of that, since it is equally probable for either of them to be ahead in the queue. The time that c may need to wait, however, varies depending on which actor is being served. For example, if a is ahead in the queue, c has to wait for $\frac{\tau(a_i)}{2}$ due to a, since a is being served. However, since the entire b remains to be served after a is finished, c needs to wait $\tau(b_i)$ for b. One can also observe that the waiting time due to actor a is $\mu_a.P_a$ when it is in front, and $2.\mu_a.P_a$ when behind. Adding all the above equations, we get

$$\begin{aligned} t_{wait}(c) &= \frac{1}{2}.P_a.P_b.(\mu_a + \mu_b) + \mu_a.P_a + \mu_b.P_b \\ &= \mu_a.P_a.(1 + \frac{1}{2}P_b) + \mu_b.P_b.(1 + \frac{1}{2}P_a) \end{aligned}$$

The above can be also computed by observing that whenever an actor a is in the queue, the waiting time is simply $\mu_a.P_a$, i.e. the probability of a being in the queue (regardless of other actors) and the waiting time due to it. However, when it is behind some other actor, there is an extra waiting time μ_a, since the whole of a has to be executed. The probability of a being behind b is $\frac{1}{2}.P_a.P_b$ and hence the total waiting time due to a is $\mu_a.P_a.(1 + \frac{1}{2}P_b)$. The same follows for the contribution due to b.

Queue	Probability (excl P_a)	Extra waiting prob
a	$(1 - P_b)(1 - P_c)$	
ab	$P_b(1 - P_c)/2$	
ba	$P_b(1 - P_c)/2$	$P_b(1 - P_c)/2$
ac	$P_c(1 - P_b)/2$	
ca	$P_c(1 - P_b)/2$	$P_c(1 - P_b)/2$
abc-acb	$P_b.P_c/3$	
bca-cba bac-cab	$\frac{2}{3}P_b.P_c$	$\frac{2}{3}P_b.P_c$
Total		$\frac{1}{2}(P_b + P_c) - \frac{1}{3}P_b.P_c$

Table 1: Probabilities of different queues with a

For three actors waiting in the queue, it is best explained using a table. Table 1 shows all the possibilities of queue with a in it. The first column contains the ordering of actors in the queue, where the leftmost actor is the first one in the queue. All the possibilities are shown in it together with their probabilities. Please note that since a is in all the queues, the probability component P_a has been excluded. For the cases when a is not in front, the waiting time is increased by $\mu_a.P_a$, and therefore, those probability terms are added again. The same can be easily derived for other actors too. We therefore obtain the following equation.

$$\begin{aligned} \mu_{abc}.P_{abc} =& \mu_a.P_a.\Big(1 + \frac{1}{2}(P_b + P_c) - \frac{1}{3}P_b.P_c\Big) \\ &+ \mu_b.P_b.\Big(1 + \frac{1}{2}(P_a + P_c) - \frac{1}{3}P_a.P_c\Big) \\ &+ \mu_c.P_c.\Big(1 + \frac{1}{2}(P_a + P_b) - \frac{1}{3}P_a.P_b\Big) \end{aligned} \quad (3)$$

We can also understand intuitively, to an extent, how the terms in the equation derived above contribute to the delay (extra waiting time) in the analysis. The first term of each of the three lines (for instance $\mu_a.P_a$ in first line) denotes the delay due to the respective actors. The terms that follow are the probabilities of the actor being in the queue; being there with at least one more actor but behind; and then with at least two more actors and so on and so forth. Since the third probability term (≥ 2 *actors*) is included in the second probability term (≥ 1 *actor*), the last term is subtracted.

Equation 3 is now generalized using the same reasoning for n actors $a_1, a_2, \ldots a_n$ mapped on a resource to give

$$\begin{aligned} \mu_{a_1 \ldots a_n} P_{a_1 \ldots a_n} =& \sum_{i=1}^{n} \mu_{a_i} P_{a_i} \Big(1 + \sum_{j=1}^{n-1} \frac{(-1)^{j+1}}{j+1} \\ & \prod\nolimits_j (P_{a_1} \ldots P_{a_{i-1}} P_{a_{i+1}} \ldots P_{a_n})\Big) \end{aligned} \quad (4)$$

where $\prod_j (x_1, ..., x_n)$ is an elementary symmetric polynomial defined in [16]. We observe that as the number of actors mapped on a node increases, the complexity of analysis also becomes high. To be exact, the complexity of the above formula is $O(n.n^n)$, where n is the number of actors mapped on a node. Since this is done for each actor, the overall complexity becomes $O(n^2.n^n)$. In the next section we see how this complexity can be reduced.

4. COMPLEXITY REDUCTION

The complexity of the analysis plays an important role when putting an idea to practice. In this section we see how we can reduce the complexity of the proposed approach, and also present two different approaches of employing our idea.

4.1 Approximating for Implementation

The total complexity for analysis in Equation 4 is $O(n^2.n^n)$. Using some clever techniques for implementation, the complexity can be reduced to $O(n^2+n^n)$ i.e. $O(n^n)$, which is still infeasible and not scalable. An important observation that can be made is that higher order terms start to appear in our analysis. The number of these terms in Π_j in Equation 4 increases exponentially. Since these terms are products of probabilities, higher order terms can be neglected. To limit the computational complexity, we provide a second order approximation of the formula.

$$\mu_{a_1 \ldots a_n} P_{a_1 \ldots a_n} \approx \sum_{i=1}^{n} \mu_{a_i} P_{a_i} \Big(1 + \frac{1}{2} \sum_{j=1, j \neq i}^{n} (P_{a_j})\Big) \quad (5)$$

Using this approximation the complexity reduces to $O(n^2)$ (using clever implementation). In general, the complexity can be reduced to $O(n^m)$ for $m \geq 2$ by using m-th order approximation. In Section 5 we present results of second and fourth order approximations of Equation 4.

4.2 Composability-based Approach

In this approach, two actors are *composed* into one actor such that the properties of this new actor can be approximated by the sum of their individual properties. In particular, if we have two actors a and b, we would like to know their combined blocking probability P_{ab}, and combined waiting time due to them $\mu_{ab}.P_{ab}$. We further define this composability operation for probability by $\oplus$ and for waiting time by $\otimes$. We therefore get,

$$P_{ab} = P_a \oplus P_b = P_a + P_b - P_a.P_b \tag{6}$$

$$\mu_{ab}.P_{ab} = \mu_a.P_a \otimes \mu_b.P_b = \mu_a.P_a.(1+\frac{P_b}{2}) + \mu_b.P_b.(1+\frac{P_a}{2}) \tag{7}$$

(Strictly speaking $\otimes$ operation also requires individual probabilities of the actors as inputs, but this has been omitted in the notation for simplicity.) Associativity of $\oplus$ is easily proven by showing $P_{abc} = P_{ab} \oplus P_c = P_a \oplus P_{bc}$. Operation $\otimes$ is associative only to second order approximation. This can be proven in a similar way by showing $\mu_{abc}P_{abc} = \mu_{ab}P_{ab} \otimes \mu_c P_c = \mu_a P_a \otimes \mu_{bc} P_{bc}$.

Associative property of these operations reduces the complexity even further. Complexity of Equation 6 and 7 is clearly $O(1)$. If waiting time of a particular actor is to be computed, all the other actors have to be combined giving a total complexity of $O(n^2)$, which is equivalent to the complexity of second-order approximation approach. However, in this approach the effect of actors is incrementally added. Therefore, when a new application has to be added to the analysis and new actors are added to the nodes, the complexity of the computation is $O(n)$ as compared to $O(n^2)$ in the case of second-order approximation, for which the entire analysis has to be repeated.

Computing inverse of Formulae

The complexity of this *Composability-based* approach can be further reduced when we can compute the inverse of the formulae in Equation 6 and 7. When the inverse function is known, all the actors can be *composed* into one actor by deriving their total blocking probability and total average blocking time. To compute the individual waiting time, only the inverse operation with their own parameters has to be performed. The total complexity of this approach is $O(n) + n.O(1) = O(n)$. The inverse is also useful when applications enter and leave the analysis, since only an incremental *add* or *subtract* has to be done to update the waiting time instead of computing all the values.

The inverse for both operations are given below.

$$\begin{aligned} & P_{a_1 \ldots a_n b} = P_{a_1 \ldots a_n} \oplus P_b \\ \Rightarrow & P_{a_1 \ldots a_n} = P_{a_1 \ldots a_n b} \oplus^{-1} P_b = \frac{P_{a_1 \ldots a_n b} - P_b}{1 - P_b} \quad (P_b \neq 1) \end{aligned} \tag{8}$$

$$\begin{aligned} & \mu_{a_1 \ldots a_n b} P_{a_1 \ldots a_n b} = \mu_{a_1 \ldots a_n} P_{a_1 \ldots a_n} \otimes \mu_b P_b \\ \Rightarrow & \mu_{a_1 \ldots a_n} P_{a_1 \ldots a_n} = \mu_{a_1 \ldots a_n b} P_{a_1 \ldots a_n b} \otimes^{-1} \mu_b P_b \\ \Rightarrow & \mu_{a_1 \ldots a_n} P_{a_1 \ldots a_n} = \frac{\mu_{a_1 \ldots a_n b} P_{a_1 \ldots a_n b} - \mu_b.P_b(1 + \frac{P_{a_1 \ldots a_n}}{2})}{1 + \frac{P_b}{2}} \end{aligned} \tag{9}$$

It should be mentioned that the inverse formula can only be applied when $P_b \neq 1$.

5. PERFORMANCE EVALUATION

In this section we present the results of above analysis obtained as compared to simulation results for a number of use-cases. For this purpose, ten random SDFGs were generated with eight to ten actors each using the SDF^3 tool [14], mimicking DSP or a multimedia application, and was a strongly connected component i.e. every actor in the graph can be reached from every actor. The execution time and the rates of actors were also set randomly. The SDF^3 tool was also used to analytically compute period of the graphs. Using these ten SDFGs, over a thousand use-cases (2^{10}) were generated. Simulations were performed using POOSL [17] to give actual performance achieved for each use-case. Three different probabilistic approaches were used - second order approximation of Equation 4, fourth order approximation of the same and composability approach using Equation 7. Results of worst-case-response-time analysis [6] for non-preemptive systems are also presented for comparison.

The simulation of all possible use-cases, each for 500,000 cycles took a total of 23 hours on a Pentium 4 3.4 GHz with 3 GB of RAM. In contrast, analysis for all four approaches was completed in only about 10 minutes. Computing waiting times due to probabilistic estimates takes negligible time. The only significant time is spent in computation of throughput for each use-case. Throughput computation took about 6ms on average for each application for each use-case. Overall there were about 5000 (5 applications in each use-case on average) throughputs to be computed, making it about 30 seconds for each estimation technique. The rest of the time was spent in file handling by the operating system.

Figure 5 shows a comparison between periods computed analytically using different approaches as described in the paper, and the simulation result. The use-case for this figure is the one in which all applications are executing concurrently. This is the case with maximum contention. The period shown in the figure is normalized to the original period of each application that is achieved in isolation. The worst case observed in simulation is also shown.

A number of observations can be made from the figure. We see how the period is much higher when multiple applications are run. For application C, the period is six times the original period, while for application H, it is only three-fold (simulation results). The analytical estimates computed using different approaches are also shown in the same graph. The estimates using the worst-case-response-time [3] is much higher than that achieved in practice and therefore, overly pessimistic. The estimates of all the three probabilistic approaches are very close to the observed performance.

We further notice that the second order estimate is always more conservative than the fourth order estimate, which is expected, since it overestimates the contention for resources.

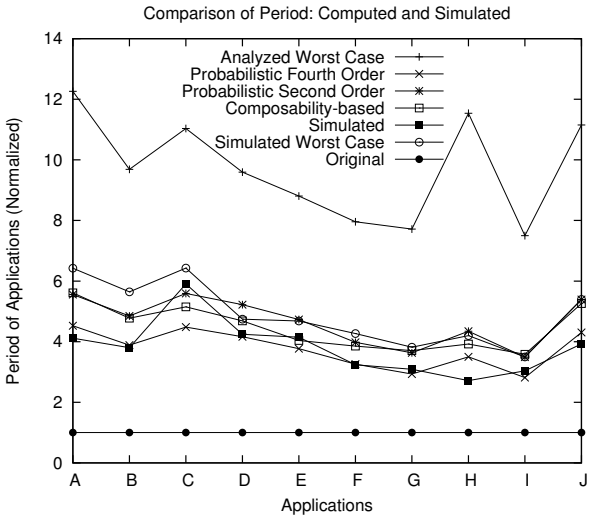

Figure 5: Comparison of period computed using different analysis techniques as compared to simulation result (all 10 applications running concurrently)

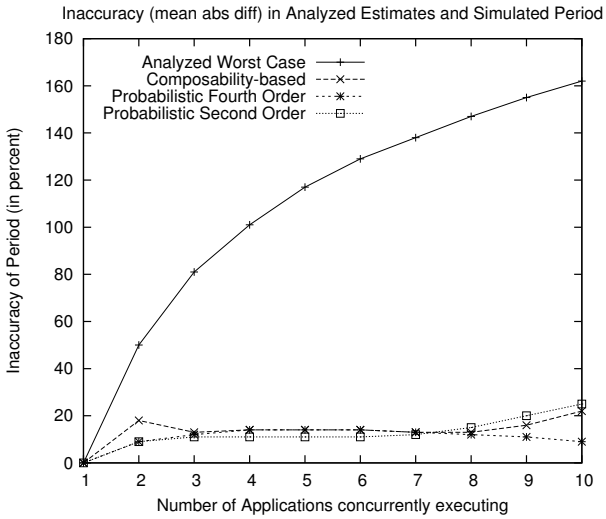

Figure 6: Inaccuracy in application periods obtained through simulation and different analysis techniques

Method	Inaccuracy in Percent Throughput	Period	Complexity
Worst Case	49.0	112.1	$O(n)$
Composability	4.0	13.8	$O(n)$
Fourth Order	0.7	13.1	$O(n^4)$
Second Order	2.8	11.2	$O(n^2)$

Table 2: Measured inaccuracy for throughput and period as compared with simulation results. The complexity of all the algorithms is also shown.

The fourth order estimates of probability is the closest to the simulation results except in applications C and H.

Table 2 shows a summary of the measured inaccuracy using different estimation techniques as compared to simulated case. These results are taken as the mean absolute difference between the estimated and measured results, and averaged over all the use-cases. The corresponding complexity of the approach is also shown. As can be seen, the worst-case approach provides estimates that are much higher than those found through our approaches. Estimates of the fourth-order approximation are the best in terms of throughput. However, the inaccuracy of the other two probabilistic approaches is also negligible. The complexity of the worst-case based approach is indeed the least. However, the composability-based approach can also achieve the same complexity if none of the probabilities are 1. The complexity of the the other two approaches is higher, but that comes with the advantage of much better accuracy over the worst-case approach.

Figure 6 shows the variation in period that is estimated and observed as the number of applications simultaneously executing in the system increases. The metric displayed in the figure is the mean of absolute differences between estimated and observed period. When there is only one application active in the system, the inaccuracy is zero for all the approaches, since there is no contention. As the number of applications increases, the worst-case-response-time estimate deviates a lot from the simulation result. This indicates why this approach is not scalable with number of applications in the system. For the other three approaches, we observe that the variation is usually within 20% of simulation result. We also notice that the second order estimate is almost exactly equal to the composability-based approach - both of which are more conservative than the fourth-order approximation. The maximum deviation in the fourth order approximation is about 14% as compared to about 160% in the worst-case approach - a ten-fold improvement.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a novel mechanism to analytically derive throughput of concurrently running applications. Two different approaches based on that mechanism are presented, which offer trade-offs between accuracy and efficiency in implementation. These mechanisms are compared with the worst-case estimation technique - the state of the art in performance estimation for mapping multiple applications on a multi-processor platform. The approach is verified with real performance achieved during simulation. The approach is scalable with the number of applications - it is extremely fast, yet accurate for a large number of applications, and up to ten times better than the worst-case estimation approach.

Since the approach is fast, it is feasible to employ this technique for run-time admission control. The approach can benefit even more by using the run-time throughput of the applications to better estimate the effect of incoming applications. The application, for example, can be admitted only if its expected throughput is above the desired throughput. Further, the approach can be easily extended to varying execution times, for example, in data dependent executions where execution times are not fixed but follow a probabilistic distribution. In future, we intend to extend our approach to take task dependencies in a graph into consideration.

7. REFERENCES

[1] L. Abeni and G. Buttazzo. QoS guarantee using probabilistic deadlines. In *11th ECRTS*. IEEE, 1999.

[2] N. Bambha *et al.* Intermediate Representations for Design Automation of Multiprocessor DSP Systems. In *Design Automation for Embedded Systems*, volume 7, pages 307–323. Springer, 2002.

[3] M. Bekooij *et al.* 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.

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

[5] A. Ghamarian *et al.* Throughput Analysis of Synchronous Data Flow Graphs. In *6th ACSD*. IEEE, 2006.

[6] R. Hoes. Predictable Dynamic Behavior in NoC-based MPSoC. Masters Thesis, Tech. Univ. Eindhoven, 2004.

[7] A. Kumar *et al.* Global Analysis of Resource Arbitration for MPSoC. In *9th DSD*. IEEE, 2006.

[8] S. Kunzli *et al.* Combining Simulation and Formal Methods for System-Level Performance Analysis. In *DATE*. IEEE, 2006.

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

[10] S. Manolache, P. Eles, and Z. Peng. Schedulability analysis of applications with stochastic task execution times. *Trans. on Embedded Computing Sys.*, 3(4):706–735, 2004.

[11] J. Pino and E. Lee. Hierarchical static scheduling of dataflow graphs onto multipleprocessors. In *ICASSP*. IEEE, 1995.

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

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

[14] S. Stuijk, M. Geilen, and T. Basten. SDF3: SDF For Free. In *6th ACSD*. `http://www.es.ele.tue.nl/sdf3`, 2006.

[15] S. Stuijk, M. Geilen, and T. Basten. Exploring Trade-Offs in Buffer Requirements and Throughput Constraints for Synchronous Dataflow Graphs. In *43th DAC*. ACM, 2006.

[16] D. Terr and E. W. Weisstein. Symmetric Polynomial. Available from: `mathworld.wolfram.com/SymmetricPolynomial.html`.

[17] B. Theelen *et al.* Software/Hardware Engineering with the Parallel Object-Oriented Specification Langauge. In *5th MEMOCODE*. IEEE, 2007.

[18] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *ISCAS*. IEEE, 2000.

[19] M. Wiggers *et al.* Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure. In *4th CODES+ISSS*. ACM, 2006.