Iterative Probabilistic Performance Prediction for Multi-Application Multiprocessor Systems

Akash Kumar, *Member, IEEE,* Bart Mesman, Henk Corporaal, *Member, IEEE,* and Yajun Ha, *Senior Member, IEEE*

***Abstract*—Modern embedded devices are increasingly becoming multiprocessor with the need to support a large number of applications to satisfy the demands of users. Due to a huge number of possible combinations of these multiple applications, it becomes a challenge to predict their performance. This becomes even more important when applications may be dynamically started and stopped in the system. Since modern embedded systems allow users to download and add applications at run-time, a complete design-time analysis is not always possible. This paper presents a new technique to accurately predict the performance of multiple applications mapped on a multiprocessor platform. Iterative probabilistic analysis is used to estimate the time spent by tasks during their contention phase, and thereby predicting the performance of applications. The approach is scalable with the number of applications and processors in the system. As compared to earlier techniques, this approach is much faster and scalable, while still improving the accuracy. The analysis takes 300 μs on a 500 MHz processor for ten applications. Since multimedia applications are increasingly becoming more dynamic, results of a case-study with applications with varying execution times are also presented. In addition, results of a case-study with real applications executing on a field-programmable gate array multiprocessor platform are shown.**

***Index Terms*—Heterogeneous multiprocessor, multiple applications, non-preemption, performance prediction, synchronous data flow graphs.**

I. Introduction

CURRENT DEVELOPMENTS in modern embedded devices like a set-top box and a mobile phone integrate a number of applications or functions in a single device, some of which are not known even at design time. Therefore, an increasing number of processors are being integrated into a single chip to build multiprocessor systems-on-chip. To achieve high performance in such systems, the limited computational resources must be shared causing contention. Modeling and analyzing this interference is essential to building cost-effective systems which can deliver the desired performance of the applications.

Manuscript received February 13, 2009; revised May 25, 2009 and October 13 2009. Current version published March 19, 2010. This paper was recommended by Associate Editor, Y. Paek.

A. Kumar and Y. Ha are with the Department of Electrical and Computer Engineering, National University of Singapore, Singapore 117576 (e-mail: akash@nus.edu.sg; elehy@nus.edu.sg).

B. Mesman and H. Corporaal are with the Eindhoven University of Technology (TUe), Eindhoven 5612AZ, The Netherlands (e-mail: b.mesman@tue.nl; h.corporaal@tue.nl).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2010.2042887

However, with an increasing number of applications running in parallel leading to a large number of possible *use-cases*, their performance analysis becomes a challenging task. (A *use-case* is defined as a possible set of concurrently running applications.) Future multimedia platforms may easily run 20 applications in parallel, resulting in up to 2^{20} potential use-cases. It is clearly impossible to verify the correct operation of all these situations through testing and simulation. 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.

Fig. 1 puts different approaches for performance evaluation in perspective. The way to obtain the most realistic performance estimates is measuring it on the real system. However, this is often not available until late in the design process. An alternative is simulating the (partitioned) application code on a multiprocessor simulation platform that models all the details, like a multi-processor ARM simulator. However, this is rather slow. System hardware prototypes on a field-programmable gate array (FPGA) are also a viable alternative that is faster once the platform is available. However, this often implies a high synthesis time making the approach infeasible for design space exploration (DSE). In order to reduce this time, application models may be derived that simulate the behavior of applications on a high level. These models may then be simulated using a transaction level simulator that also takes the architecture and mapping into account. Besides software, some hardware platforms are also available for this simulation [1]. The benefit of using such a simulator is that it is much faster than a cycle-accurate simulator or synthesizing a prototype for FPGA. However, when dealing with a large number of use-cases, this approach may still not be feasible for DSE, and certainly not for run-time implementation. To further speed performance estimation, analyzing models mathematically is the best.

The focus of this paper is on analyzing performance when multiple applications share a multiprocessor platform for a given mapping. While this analysis is well understood (and relatively easier) for preemptive systems [2]–[4], non-preemptive scheduling has received considerably less attention. However, for high-performance embedded systems (like cell-processing engine and graphics processor), non-preemptive systems are preferred over preemptive systems for a number of reasons [5]. Further, even in multiprocessor systems with preemptive processors, some processors (or coprocessors/ accelerators) are

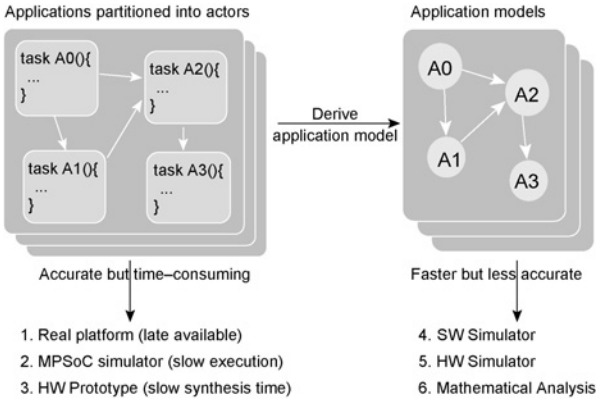


Fig. 1. Comparison of various techniques for performance evaluation.

usually non-preemptive; for such processors non-preemptive analysis is still needed.

A. Our Contribution

In this paper, we propose a new technique to accurately predict performance of multiple applications executing on a non-preemptive multiprocessor platform. In our analysis, we use the synchronous data flow (SDF) model since this is the most compact model that still allows analysis of properties easily and public tools are available for analyzing application performance in isolation [6]. Since the application is modeled as an SDF graph, the communication and memory access can also be modeled as an actor (or multiple actors, depending on the type of communication fabric) in the graph using techniques presented in [7]. The execution time of such actors corresponds to the delay during data communication or memory access.

While in this paper we have applied this approach to SDF graphs, it can be applied on any model of computation which allows analysis of performance parameters like throughput and buffer-requirement of independent applications. Some examples are cyclo-static data flow and homogeneous synchronous data flow. Recently, an extension to SDF has been proposed, known as a scenario aware data flow (SADF) graph, that allows modeling of dynamism in an SDF graph. Models like Kahn process networks cannot be used since the execution time is not known *a priori*.

When applications are modeled as SDF graphs, their performance on a (multiprocessor) system can be easily computed when they are executing *in isolation*. However, when they execute concurrently with other applications, there is contention for resources. Determining the time the individual tasks (or actors) have to wait for resources to become available is important in order to accurately estimate the overall application performance. In this paper, we present a technique to predict the time that tasks (or actors) have to spend during the contention phase for a resource. This technique evaluates the probability of a task blocking a resource by considering how often the task requests the resource, and how long it takes during each execution. Using this information, the expected waiting time for all tasks sharing a resource is computed. These waiting time estimates, together with the original execution times, are used to predict the performance of applications. This, in turn, affects the probability of blocking the resource, and the entire analysis is repeated until it converges. Therefore, we call this iterative probabilistic performance prediction (IP^3) technique. The approach is very fast and can be used at both design-time and run-time owing to its low implementation complexity, in contrast with simulating or executing the application on an FPGA or the models using a simulator.

Following are the key features of the IP^3 analysis presented.

1) *Accurate:* The observed accuracy in the experiments is between 2 and 15% on average.
2) *Fast:* The algorithm has the complexity of $O(n)$, where n is the number of actors on each processor.
3) *Scalable:* The algorithm is scalable in the number of actors per application, the number of processing nodes, and the number of applications in the system.
4) *Suitable for embedded systems:* The algorithm has been tested on an embedded processor on FPGA, and requires very few cycles to estimate the application period.

We also see the effectiveness of this approach when applied to dynamic execution times in the experiments section. Results of a case-study done with typical real-life applications in a mobile phone are also presented. Further, we compare the results of the analysis with an actual multiprocessor implementation. However, it should be added that this approach does not provide any timing guarantees. Further, contention caused by shared bus and input/output (I/O) devices is not considered enough.

The remainder of this paper is organized as follows. Section II gives an introduction to SDF graphs. Section III explains the iterative probability technique that is used to predict performance of multiple applications. Section IV checks the validity of the model assumptions by doing experiments and measuring the probability distribution. Section V presents and compares the results obtained with other state-of-the-art techniques. Section VI discusses related work about how performance analysis is traditionally done—for single and multiple applications, and finally, Section VII presents the major conclusions and gives directions for future work.

II. Synchronous Data Flow Graphs (SDFGs)

SDFGs (see [8]) are often used for modeling modern digital signal processing (DSP) applications [9] and for designing concurrent multimedia applications implemented on a multiprocessor system-on-chip. Both pipelined streaming and cyclic dependences 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 [10].

Fig. 2 shows an example of an SDF graph. There are three actors (also known as tasks) in this graph. As in a typical data flow graph, a directed edge represents the dependence between actors. Actors also need some input data (or control information) before they can start and they usually also produce some output data; such information is referred to as *tokens*. The number of tokens produced or consumed in one execution of actor is called *rate*. In the example, a_0 has an input rate of 1 and output rate of 2. Actor execution is also

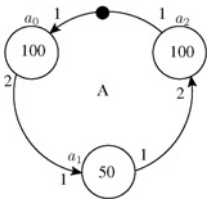


Fig. 2. Example of an SDF graph.

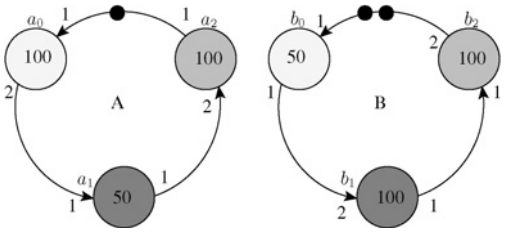


Fig. 3. Two application SDFGs A and B.

called *firing*. An actor is called *ready* when it has sufficient input tokens on all its input edges and sufficient buffer space on all its output channels; an actor can only fire when it is ready. When the actor gets ready to execute on a processor, it is also said to *arrive* on that processor. The edges may also contain *initial tokens*, indicated by bullets on the edges, as seen on the edge from actor a_2 to a_0 in Fig. 2.

One of the most interesting properties of SDFGs relevant to this paper is throughput. Throughput is defined as the inverse of the long term period, i.e., the average time needed for one iteration of the application. (An iteration is defined as the minimum non-zero execution such that the original state of the graph is obtained.) This is the performance parameter that we use in this paper. We now define the following properties of an SDF graph.

Definition 1 (Actor Execution Time): Actor execution time, $\tau(a)$ is defined as the execution time needed to complete execution of actor a on a specified node. $\tau(a)$ is also represented as τ_a interchangeably.

$\tau(a_0) = 100$, for example, in Fig. 2. When the actor represents a communication or a memory node, the execution time determines the time for data transfer or memory access, respectively.

Definition 2 (Repetition Vector): Repetition vector q of an SDFG A is defined as the vector specifying the number of times actors in A are executed for one iteration of SDFG A.

For example, in Fig. 2, q[a0 a1 a2] = [1 2 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 Fig. 2. (Note that actor a_1 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. When network and memory access is also modeled in the graph, then the throughput of the graph will also take such delay into account.

Determining the worst-case-execution time of an actor is one of the hardest things. A number of tools are available to do the same for the designer [11]. A number of techniques are present in the literature to do the partitioning of program code into tasks. Compaan is one such example that converts sequential description of an application into concurrent tasks by doing static code analysis and transformation [12]. Sprint also allows code partitioning by allowing the users to tag the functions that need to be split across different actors [13]. Yet another technique has been presented that is based on an execution profile [14]. For this paper, we shall assume that the analysis has already been done and the application is already modeled as an SDF graph.

Often an application can be associated with multiple quality levels as has been explained in existing literature [15]. Each quality of the application will, in that case, be depicted with a different task graph with (potentially) different requirements of resources and different performance constraints.

Further, we shall assume that all actors have an auto-concurrency of 1. Auto-concurrency of an actor implies how many instances of an actor can be active in parallel. Auto-concurrency of more than 1 implies that an actor is simultaneously executing on multiple processors. Allowing this in practice would lead to a number of complications and require a lot of hardware support including code-duplication, ensuring that data-tokens produced from different processors are still fed in the succeeding actor sequentially, and so on. However, having said that, any actor with auto-concurrency of more than 1, say n, can be represented with n actors each with auto-concurrency of 1. Therefore, assuming auto-concurrency of 1 in the analysis is sufficient and practical.

III. Probabilistic Analysis

When multiple applications execute in parallel, they often cause contention for shared resources. A probabilistic model can be used to predict this contention. The time spent by an actor in contention is added to its execution time, and the total gives its response time

$$t_{\text{resp}} = t_{\text{exec}} + t_{\text{wait}}. \tag{1}$$

The t_{wait} is the time that is spent in contention when waiting for a processor resource to become free. (This time may be different for different arrivals of a repetitive task.) The response time, t_{resp}, indicates how long it takes to process an actor after it arrives at a node. When there is no contention, the response time is simply equal to the execution time. Using only the execution time gives the maximum throughput that can be achieved with the given mapping. At design-time, since the run-time application-mix is not always known, it is not possible to exactly predict the waiting-time, and hence the performance. In this section, we explain how an estimate is obtained using a probabilistic approach.

We now refer to SDFGs A and B in Fig. 3. Say a_0 and b_0 are mapped on a processor $Proc_0$. a_0 is active for time $\tau(a_0)$ every $Per(A)$ time units (since its repetition entry is 1). $\tau(a_0) = 100$ time units and $Per(A) = 300$ time units on average. Assuming the process of executing tasks is stationary and ergodic, the probability of finding $Proc_0$ in use by a_0 at a random moment in time equals $\frac{1}{3}$. We now assume that the arrivals of a_0 and b_0 are stationary and independent; thus, the probability of $Proc_0$

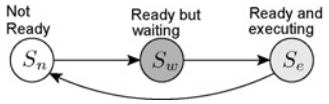


Fig. 4. Different states an actor cycles through.

being occupied when b_0 arrives is also $\frac{1}{3}$.[1] 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, given the fact that a_0 is executing, is uniformly distributed from [0, 100]. Therefore, the expected waiting time is 50 time units and b_0 has to wait for 50 time units on average on a long-run execution whenever it finds $Proc_0$ blocked due to a_0. Since the probability that the resource is occupied is $\frac{1}{3}$, the average time actor b_0 has to wait is given by $\frac{50}{3} \approx 16.7$ time units. The average response time of b_0 will therefore be 66.7 time units.

A. Formal Analysis

Fig. 4 shows that any actor of an SDF graph has to go through three different states. When the actor does not have enough input data or output space, i.e., sufficient tokens on all of its incoming edges and available buffer capacity on all of its output edges, it is not ready. This state is denoted by S_n. When the data is available, the actor becomes ready. However, if the required resource is busy then the actor may still have to wait. We denote this state of ready but waiting for the resource to become available as S_w. When the processor or another resource becomes available, the actor starts executing and this state is denoted as S_e. For an actor whose execution time is constant, the time spent in the executing state S_e does not change, and is simply equal to its execution time $\tau(a)$. The time spent during waiting state S_w depends on the available resources. If there is no other actor mapped on a particular resource, then this time is simply zero. The time spent during not-ready state S_n depends on the graph structure and the period of the graph.

We can define the state of the task (Fig. 4) as a stochastic process $S(t)$. We assume that this process is ergodic and stationary. The total probabilities of finding an actor in any of these states are clearly 1. Thus, we obtain

$$P(S(t) = S_n) + P(S(t) = S_w) + P(S(t) = S_e) = 1 \qquad (2)$$

where $S(t)$ denotes the state at time t. We will see that the steady-state probabilities of an actor being in the states described above can be computed by considering the graph structure, the actor execution time, and some properties of other actors mapped on the sharing resource. The probability of finding an actor a in executing state S_e can be computed by considering how often it executes, i.e., its repetition vector entry $q(a)$, and its execution time $\tau(a)$. To put it precisely, the actor a executes $q(a)$ times every period $Per(A)$ of the application A to which a belongs, and each time it spends $\tau(a)$ cycles in the state S_e. Thus, the total time spent is $q(a).\tau(a)$ every $Per(A)$. Thus, because of the stationarity of the process, the steady-state probability of finding actor a in the executing state is given by the following equation:

$$P(S(t) = S_e) = \frac{q(a).\tau(a)}{Per(A)}. \qquad (3)$$

When the actor is sharing resources with other actors it may also have to wait for the resource to become available. If the average waiting time is denoted by $t_{\text{wait}}(a)$, then the total time spent in the waiting state, on average, is given by $q(a).t_{\text{wait}}(a)$ every $Per(A)$. Thus, the steady-state probability of finding actor a in the waiting state is given by the following equation:

$$P(S(t) = S_w) = \frac{q(a).t_{\text{wait}}(a)}{Per(A)}. \qquad (4)$$

Since the total probability for all the states should be 1, the probability of actor a being in the non-ready state can be computed as follows:

$$P(S(t) = S_n) = 1 - \frac{q(a).t_{\text{wait}}(a)}{Per(A)} - \frac{q(a).\tau(a)}{Per(A)}. \qquad (5)$$

The actor a only blocks the resource when it is either waiting or executing at the resource. (Blocking is defined as occupying a resource when another actor requests for it.) Thus, we define two types of blocking probabilities.

Definition 4 (Execution Blocking Probability): Execution Blocking Probability, $P_e(a)$, is defined as the probability that actor a of application A blocks the resource it is mapped on, *and is being executed*. $P_e(a) = P(S(t) = S_e)$.

Definition 5 (Waiting Blocking Probability): Waiting Blocking Probability, $P_w(a)$, is defined as the probability that actor a of application A blocks the resource it is mapped on *while waiting for it to become available*. $P_w(a) = P(S(t) = S_w)$.

When other actors, say b and c are sharing resources with a, it is important to know how long they may need to wait due to contention with a. This clearly depends on which of the three states a is in when these actors are queued, and the arbiter used. For our analysis, we shall assume a first-come-first-serve (FCFS) arbiter unless otherwise stated, since it is one of the most commonly used dynamic arbiter. With FCFS if b arrives when a is in the non-ready state S_n, then a does not contribute to the waiting time of b for that particular time. If b arrives when a is also waiting in the queue, i.e., state S_w, then b goes behind a in the queue (since we have FCFS), and it has to wait for the whole execution of a to finish before b can get its turn. When a is executing, the waiting time for b depends on where a is in its execution. If it is about to finish then b has to wait for a short while, but if a has just started, then b has to wait for almost the entire execution time of a.

Assuming that the arrival time of b is completely independent of the different states of a, the probability of b finding a in a particular state is simply the stationary probability of a being in that state. (See Footnote 1.) Further, our assumption also implies that when b arrives and finds a in a particular state, a may be anywhere, with uniform distribution, in that state. Thus, if b finds a in the S_e state, then the remaining execution time is uniformly distributed. Since the probability of finding a in a particular state is directly related to the waiting time of b, we obtain the probability distribution for waiting time of b as shown in Fig. 5.

[1]We know that in reality these are not independent since there is a dependence on resources. This assumption is made in order to simplify the analysis and keeping it composable. We study the impact of this assumption on the accuracy of the prediction made by this probabilistic model in Section IV.

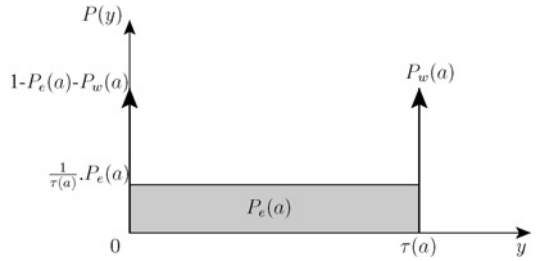


Fig. 5. Probability distribution of the waiting time added by actor a to another actor when actor a is mapped on the resource.

As shown in Fig. 5, the time actor b has to wait depends on the state of actor a when b arrives. When b arrives in the S_w state of a, it has to always wait for $\tau(a)$. This gives the $\delta-$function of $P_w(a)$ at $\tau(a)$. On the other extreme we have the $\delta-$function at origin due to b arriving in the S_n state of a. The probability of this is simply equal to the probability of a being in this state, as mentioned earlier. In the middle we have a uniform distribution with the total probability of $P_e(a)$, i.e., a being in S_e state.

If Y denotes how long actor b has to wait for the resource it shares with actor a, the probability density function, $P(y)$ of Y can be defined as follows:

$$P(y) = \begin{cases} 0 & y < 0 \\ \delta(y).(1 - P_e(a) - P_w(a)) & y = 0 \\ \frac{1}{\tau(a)}.P_e(a) & 0 < y < \tau(a) \\ \delta(y - \tau(a)).P_w(a) & y = \tau(a) \\ 0 & y > \tau(a). \end{cases} \tag{6}$$

The average waiting time due to actor a for b, $E(Y)$ can now be computed as follows:

$$\begin{aligned} E(Y) &= \int_{-\infty}^{\infty} y\, P(y)\, dy \\ &= \int_0^{\tau(a)} y\, \tfrac{1}{\tau(a)}.P_e(a)\, dy + \tau(a).P_w(a) \\ &= \tfrac{1}{\tau(a)} P_e(a) \left[\tfrac{y^2}{2}\right]_0^{\tau(a)} + \tau(a).P_w(a) \\ &= \tfrac{\tau(a)}{2} P_e(a) + \tau(a).P_w(a) \\ &= \tau(a)\left(\tfrac{P_e(a)}{2} + P_w(a)\right). \end{aligned} \tag{7}$$

If $\tau(a)$ is not constant but varying, $E(Y)$ also varies with $\tau(a)$. In such cases, $E(Y)$ can be computed as follows:

$$\begin{aligned} E(Y) &= E\left((\tau(a)\left(\tfrac{P_e(a)}{2} + P_w(a)\right)\right) \\ &= E(\tau(a))\left(\tfrac{P_e(a)}{2} + P_w(a)\right). \end{aligned} \tag{8}$$

Thus, an actor with variable execution time within a uniform distribution is equivalent to an actor with a constant execution time, equal to the mean execution time.[2] If $\tau(a)$ is uniformly distributed between $\tau_{\min}(a)$ and $\tau_{\max}(a)$, the overall average waiting time is given below

$$E(Y) = \left(\frac{\tau_{\min}(a) + \tau_{\max}(a)}{2}\right)\left(\frac{P_e(a)}{2} + P_w(a)\right). \tag{9}$$

Since (7) represents the waiting time of one actor due to another actor, when there are more actors mapped on a resource each of the mapped actors causes a given actor to wait. For the total waiting time due to n actors, we get the following equation:

$$t_{\text{wait}} = \sum_{i=1}^{n} \left(\frac{\tau_{a_i}}{2} P_e(a_i) + \tau_{a_i} P_w(a_i)\right). \tag{10}$$

[2]It is equivalent only in terms of its expected value, not of its distribution.

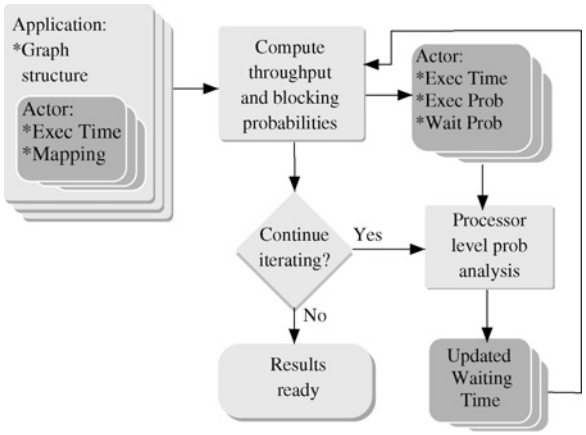


Fig. 6. Iterative probability method. Waiting times and throughput are updated until needed.

B. Iterating the Analysis

A key observation from the analysis provided above is that the periods of the applications change from the initial estimate of executing in isolation. For example, the period of application A in isolation was 300 time units, but is now estimated to be 358 time units. This in turn modifies the execution and waiting probabilities of all the actors. Thus, the waiting times of actors have to be recomputed, which in turn may change the period. Thus, the entire analysis needs to be repeated to update the period of the applications. Fig. 6 shows the flow for the iterative probability approach. The inputs to this flow is the application structure for each application, and the execution time and mapping of each actor in all the applications. These are first used to compute the base period (i.e., the minimum period without any contention) and the execution blocking probability of the actor. Using the mapping information, a list of actors is compiled from all the applications and grouped according to their resource mapping. For each processor, the probability analysis is done according to (10). The waiting times thus computed are used again to compute the throughput of the application and the blocking probabilities. Applying this analysis for the example in Fig. 3 updates the period of both applications to 362.7, 364.1, 364.2 and 364.2 time units, thereby converging at 364.2. Fig. 7 shows the updated application graphs after the iterative technique is applied.

The main reason why the analysis technique is fast is that it ignores the resource dependences that are created when actors from different applications share a resource. This very property can also become its weakness and result in arbitrarily bad estimation when cases are carefully constructed such that the resulting order on a processor does not suffer from any contention, or always suffers from the worst-case contention. In such cases, the average waiting time is no longer

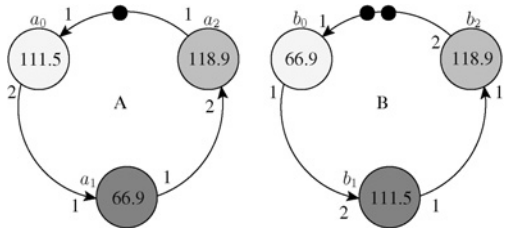


Fig. 7. SDF application graphs A and B updated after applying the iterative analysis technique.

Fig. 8. Probability distribution of waiting time another actor has to wait when actor a is mapped on the resource for the conservative iterative analysis.

applicable and may lead to erroneous estimates. In order to take such cases into account, one has no choice but to resort to considering all the data and resource dependences in one big SDF graph, and consider all possible executions of all the graphs involved, avoiding which was the very motivation behind this article. However, such cases are mostly artificial and unrealistic. In the large number of experiments, we never came across any example with this behavior. Further, when the execution times of actors are dynamic, it is even more unlikely for these cases to occur.

C. Terminating Condition

While the analysis can be repeated for a fixed number of iterations, it can also be based on the convergence of some parameters. Some candidates for testing convergence are provided below.

1) *Application period:* When the application period for all the applications does not change more than a predefined percentage, the analysis can be said to have been converged. In our experiments, we observed that just after six iterations all applications had a change of less than 1%.
2) *Processor utilization:* The analysis termination can also be based on the change in processor utilization. The utilization of processors varies with the load predicted by the algorithm. The load on a processor is defined as the sum of the probabilities of execution, $P_e(a)$, of all actors mapped on it. When the algorithm has converged, the load on the processor does not change.

We have reason to believe that the algorithm converges, since in all the experiments we conducted so far (over a thousand use-cases), it always converged. Further, a particular use-case always gave the same answer, irrespective of how far off the starting estimate was. In order to formally prove it, fixed-point arithmetic theory could be applicable [16]. However, we did not yet succeed in proving convergence.

D. Conservative Iterative Analysis

For some applications, the user might be interested in having a more conservative bound on the period, i.e., it is better to have a less accurate pessimistic estimate than an accurate optimistic estimate; a much better quality than predicted is more acceptable as compared to even a little worse quality than predicted. In such cases, we provide here a conservative analysis using our iterative technique.

In earlier analysis, when an actor b arrives at a particular resource and finds it occupied by say actor a, we assume that a can be anywhere in the middle of its execution, and therefore, b has to wait on average half of execution time of a. In the conservative approach, we assume that b has to always wait for full execution of a. In the probability distribution as presented in Fig. 5, the rectangular uniform distribution of $P_e(a)$ is replaced by another delta function at $\tau(a)$ of value $P_e(a)$. This is shown in Fig. 8. The waiting time equation is therefore updated to the following:

$$t_{\text{wait}} = \sum_{i=1}^{n} \tau_{a_i} \Big(P_e(a_i) + P_w(a_i) \Big). \tag{11}$$

Applying this analysis to the example in Fig. 3, we obtain the period as 416.7, 408, 410.3, 409.7, and 409.8. Note that in our example, the actual period will be 300 in the best case and 400 in the worst case. The conservative iterative analysis correctly finds the bound of about 410, which is only 2.5% more than the actual worst case. If we apply real worst-case analysis in this approach [using (14)], then we get a period of 600 time units, which is 50% over-estimated.

This analysis can be either applied from the original period directly, or only after the basic iterative analysis is already converged and terminated. The latter has the benefit of using a realistic period, instead of a conservative period. Since a conservative period is generally higher than the corresponding realistic period, the execution and waiting probability is correspondingly lower when using the conservative period. Thus, using a realistic period with a conservative analysis for the last iteration gives the most conservative results. In the experiments section, we present results of both approaches.

E. Parametric Throughput Analysis

Throughput computation of an SDF graph is generally very time consuming. Lately, techniques have been presented in [17] that can compute throughput of many multimedia applications within milliseconds. However, those results have been taken on a high-end computer while assuming fixed actor execution times. Therefore, throughput computation of an SDF graph is generally done off-line or at design-time for a particular graph. However, if the execution time of an actor changes, the entire analysis has to be repeated. Recently, a technique has been proposed to derive throughput equations for a range of execution times (defined as *parameters*) at design-time, while these equations can be easily evaluated at run-time to compute the critical cycle, and hence the period [18]. This technique greatly enhances the usability of the iterative analysis. With this the iterative analysis can be applied at both design-time and run-time.

For example, for application A shown in Fig. 3, there is only one critical cycle. If the execution times of all actors of A are variable, the following parametric equation is obtained (assuming auto-concurrency of 1):

$$Per(A) = \tau(a_0) + 2 \times \tau(a_1) + \tau(a_2). \quad (12)$$

Thus, whenever the period of application A is needed, the above equation can be computed with the updated response times of actors a_0, a_1 and a_2. This technique makes the iterative analysis suitable for run-time implementation.

F. Intra-Task Dependences

There are two ways of handling the situation when more than one actor of the same application are mapped on the same resource, depending on how it is handled in the real system. One way is to serialize (or order) executions of all actors of a given application. This implies computing a static-order for actors of a given application such that maximum throughput is guaranteed. This can be done using SDF^3 tool [6]. Once the static-order is computed, the partial order of actors mapped on the same resource can be extracted. The arbiter has to ensure that at any one point in time the actors of an application are executed in this pre-computed order. This ensures that actors of the same application are not queued at the same time. Thus, there is no waiting time added from these actors. For example, in Fig. 3 if actors a_0 and a_2 are mapped on the same processor, the static schedule for that processor will be $(a_0 a_2)^*$. A static order adds an extra dependency on actors a_0 and a_2, ensuring that they cannot be ready at the same time, and hence cannot cause contention for the actors mapped on the same processor. Equation (10) for an actor of application A can then be updated for this case as follows:

$$t_{\text{wait}} = \sum_{i=1, a_i \notin A}^{n} \left(\frac{\tau_{a_i}}{2} P_e(a_i) + \tau_{a_i} P_w(a_i) \right). \quad (13)$$

The above approach however, requires extra support from the arbiter. The easiest approach from the arbiter perspective is to treat all the actors mapped on the resource identically and let the actors of the same application also compete with each other for resources. The latter is evaluated in the experiments section.

G. Handling Other Arbiters

The above analysis has been presented for FCFS arbitration. For static-order schedulers like round-robin or another arbitrary order derived from SDF^3 [6], the schedule can be directly modeled in the graph itself. Other dynamic-order schedulers, like priority-based, can be easily modeled in the probability approach. One key difference between a priority-based scheduler as compared to FCFS is that in FCFS, once the actor arrives, it always has to wait for actors ahead of it in the queue. In a priority-based system, if it is preemptive, a higher priority actor can immediately preempt a lower priority actor, and if it is non-preemptive, it has to only wait for lower priority actors if they are executing. Let us define the priority of an actor a by $Pr(a)$, such that a higher value of $Pr(a)$ implies a higher priority. Equation (10), that is presented for FCFS, can be rewritten as

$$t_{\text{wait}, Pr} = \sum_{i=1}^{n} \left(\frac{\tau_{a_i}}{2} P_e(a_i) \right) + \sum_{i=1, Pr(a_i) \geq Pr(a)}^{n} \left(\tau_{a_i} P_w(a_i) \right).$$

It shows the waiting time for an actor a when sharing a resource with actors a_1 to a_n. Note that the waiting time contributed by the arrival of actor a during the queuing phase of an actor with a priority lower than that of a, is not added in the equation. Similarly, the conservative waiting time for priority-based schedulers is given by

$$t_{\text{wait}, Pr, C} = \sum_{i=1}^{n} \left(\tau_{a_i} P_e(a_i) \right) + \sum_{i=1, Pr(a_i) \geq Pr(a)}^{n} \left(\tau_{a_i} P_w(a_i) \right).$$

It can be seen that the above equations are a generalized form of earlier (10) and (11), respectively, since in FCFS the priorities of all actors are equal, i.e., $Pr(a) = Pr(a_i)\ \forall\ i = 1, 2, ...n$. It should be further noted, that since the priorities are only considered for local analysis on a specific processor (or any resource), different processors (or resources) can have different arbiters.

IV. Model Validation

In this section, we describe our experimental setup and study the impact of assumptions used in the probabilistic model. We start with experiments to show the impact of our assumptions on the predictions of our probabilistic model for arrival of actors on a resource. This is followed by comparing the predicted waiting time with measured waiting time on two processors—one fully utilized, and one less utilized.

A. Setup

Ten random SDFGs named A–J are generated with eight to ten actors each using the SDF^3 tool [6], mimicking DSP and multimedia applications. These are referred to as applications A–J. The execution time and the rates of actors are also set randomly. The edges of these graphs were randomly generated, resulting often in very complex graph topology. Some sample graphs used in this paper are available online [19]. A ten-processor heterogeneous system is used for simulation and analysis. The SDF^3 tool is used to analytically compute the periods of the graphs. Simulations are performed using parallel object oriented specification language (POOSL) [20] to measure the actual performance of the SDF models with a given mapping.

POOSL allows the designer to model both the application and the architecture up to an arbitrary level of detail. In our set up, we model each processor with a first-come-first-serve arbiter. Applications are modeled as SDF graphs with a number of actors having data dependences with other actors. When all the input data for an actor is available, the actor is queued at the mapped processor arbiter. When it gets access to the resource, it produces the output data after a given amount of delay, as specified in the model. The model is very useful for performance prediction when multiple applications share multiprocessor systems.

Besides our iterative technique, two other analysis techniques are used—the worst-case-waiting-time approach [21]

and the exponential probability approach [22]. The worst-case-waiting-time for non-preemptive systems for FCFS as mentioned in [21] is computed by using the following formula:

$$t_{\text{wait}} = \sum_{i=1}^{n} t_{\text{exec}}(a_i) \tag{14}$$

where actors a_i for $i = 1, 2, \ldots n$ are mapped on the same resource. The waiting time according to the exponential probability analysis presented in [22] is computed using the following formula ($P(a)$ is represented as P_a for brevity.):

$$t_{\text{wait}} = \sum_{i=1}^{n} \mu_{a_i} P_{a_i} \left(1 + \sum_{j=1}^{n-1} \frac{(-1)^{j+1}}{j+1} \prod_j (P_{a_1} \ldots P_{a_{i-1}} P_{a_{i+1}} \ldots P_{a_n})\right) \tag{15}$$

where

$$\prod_j (x_1, \ldots, x_n) = \sum_{1 \le k_1 < k_2 \ldots < k_j \le n} (x_{k_1} x_{k_2} \ldots x_{k_j}).$$

$\prod_j (x_1, \ldots, x_n)$ is an elementary symmetric polynomial defined in [23]. In simple terms, it is the summation of all products of j unique terms in the set $(x_1, \ldots, x_n)$. The number of terms clearly increases exponentially with increasing n. The total number of terms in the symmetric polynomial in (15) is given by $\binom{n-1}{j}$, i.e., $\frac{(n-1)!}{j!(n-1-j)!}$. As the number of actors mapped on a node increases, the complexity of the analysis also becomes high. To be exact, the complexity of the above formula is $O(n^{n+1})$, where n is the number of actors mapped on a node. Since this is done for each actor, the overall complexity becomes $O(n^{n+2})$. This high complexity arises from the fact that the approach in [22] looks at all possible combinations of other actors blocking a particular actor. Thus, while there are only three potential combinations when there are two other actors, there are 1023 combinations with ten other actors. Equation (15) is constructed by adding the contribution of individual actors (note the a_i before the first bracket). The terms inside the outer bracket are the probabilities of an actor being ahead in the queue, being there with at least one other actor, being there with at least two other actors, and so on and so forth. Since the case of being with at least two other actors is included in the case of at least one other actor, we get $(-1)^{j+1}$ to take care of the extra probabilities that are added and subtracted alternately. In our experiments a fourth-order approximation of 15 is used as a good compromise between complexity and accuracy, as proposed by the authors in [22]. The fourth-order approximation implies that all probabilities of a particular actor being in the queue with up to at least three other actors are considered.

B. Arrivals During Actor Execution

In order to check the accuracy of the probabilistic distribution of waiting times presented in Fig. 5, we let all the applications execute concurrently, and measured exactly when actors arrive when sharing a processor (or another resource) with another actor. For every execution of an actor a, three events are recorded in the processor log file—queuing time (t_q), execution start-time (t_s), and execution end-time (t_e). When other actors arrive between t_q and t_s, they have to wait for the entire execution of a. When they arrive between t_s and t_e, the waiting time depends on where a is in its execution. When the actors arrive between t_e and the next t_q, a does not have any effect on their waiting time. This was measured and summarized for the entire simulation for all the actors. Here we present results of two actors—one randomly chosen from a processor with high utilization and another with low utilization. This is done in order to check if the model still holds as the utilization of the processor approaches 1.

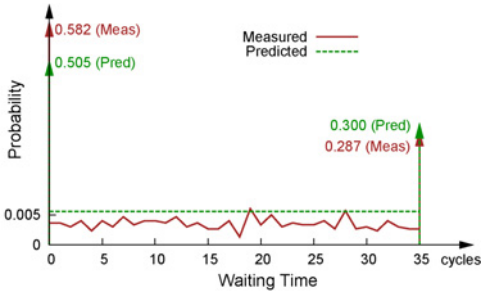


Fig. 9. Probability distribution of the time other actors have to wait for actor $a2$ of application F. $a2$ is mapped on Processor 2 with a utilization of 0.988. The average waiting time measured is 12.13 cycles, while the predicted average time is 13.92 cycles.

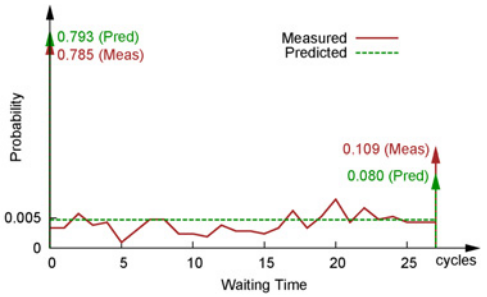


Fig. 10. Probability distribution of the time other actors have to wait for actor $a5$ of application G. $a5$ is mapped on Processor 5 with a utilization of 0.672. The average waiting time measured is 4.49 cycles, while the predicted average time is 3.88 cycles.

Fig. 9 shows the distribution of this waiting time for actor $a2$ of application F mapped on Processor 2. Processor 2 has a high utilization of almost 1. The distribution is obtained from about three thousand arrivals. This actor takes 35 cycles to execute. The distribution of actor arrival times assumed in the model is also shown in the same figure for comparison. A couple of observations can be made from this figure. The distribution between 0 and 35 is more or less uniform, though the probability in this uniform distribution is a little lower. The number of arrivals of other actors when $a2$ is not in the queue is somewhat higher than that assumed in the model, and the arrivals in the queuing time of $a2$ are rather accurate. If we look at the total waiting time contributed by $a2$, the prediction using the assumed arrival model is 13.92, whereas the measured mean delay contributed by $a2$ is 12.13—about 15% lower. The conservative analysis predicts the waiting time to be 17.94 due to $a2$. Fig. 10 shows a similar distribution for actor $a5$ of application G mapped on Processor 5. This processor has comparatively low utilization of 0.672.

Thus, we see that our assumption of the probability distribution in Fig. 5 consisting of two delta functions and a uniform distribution in the middle holds rather well and gives a good estimate of the waiting time.

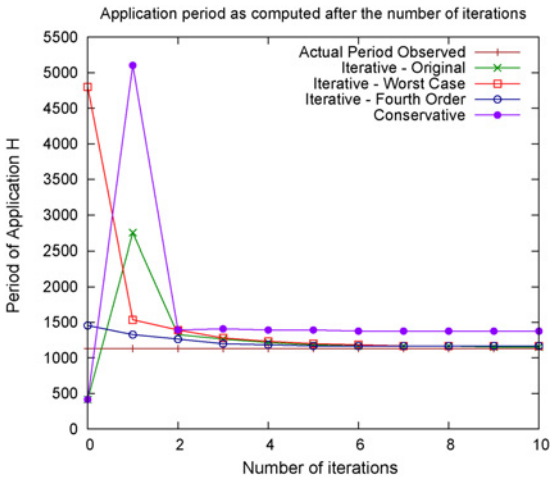


Fig. 11. Change in period computed using iterative analysis with increase in the number of iterations for application H.

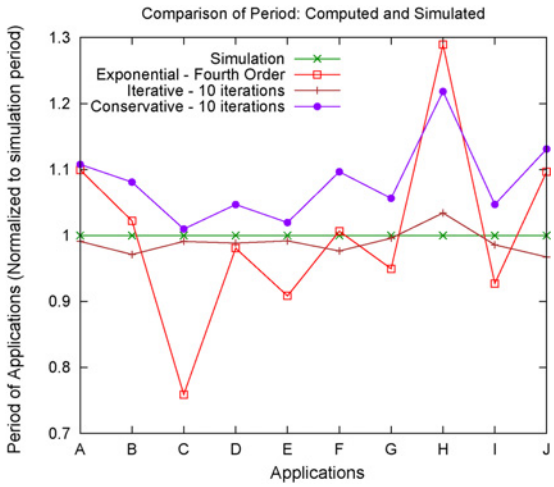


Fig. 12. Comparison of periods computed using iterative analysis techniques as compared to simulation results (all ten applications running concurrently).

V. Performance Evaluation

In this section, we present some results obtained for the iterative analysis as compared to simulation, and other analysis techniques. The overall application throughput predicted is compared with the measured throughput. Comparison with other analysis techniques (including our previous work) is also presented. Case-studies with dynamic execution times and mapping multiple actors of the same application on a node are presented. We also present case-studies with applications in a mobile phone, and comparison with Sobel and JPEG encoders executing on an FPGA multiprocessor platform. The section concludes with implementation of our technique on an embedded processor—Microblaze, and comparison of its complexity with existing techniques.

A. Application Throughput

We consider the same set of ten applications that is used in the earlier section for model validation. All ten applications are executed together on a ten-processor platform to measure the performance of these applications when executing concurrently. The iterative analysis is also used to compute the performance of all the applications. The results of other techniques, namely the worst-case and exponential technique to fourth order, are also computed, as per (14) and (15), respectively. An iterative technique is also applied to the results of worst-case and exponential approach, to study the effectiveness of the iterative approach. The effectiveness of the conservative iterative technique is also studied.

Fig. 11 shows the results of iterative analysis with an increasing number of iterations for application H. For this particular application, the original period (i.e., when running in isolation) is 416. When running concurrently in this use-case, the period is 1130 time units. The fourth-order approximation estimates the performance as 1456, while the worst-case estimate is 4800. The iterative approach when applied from the original period after five iterations predicts a period of 1200, and after ten iterations 1184 time units. After ten iterations there is no change in the estimate of applications.

The figure shows some very interesting results. First, we can see that the iterative approach is converging. Regardless of how far and at which side the initial estimate of the application behavior is, it converges within a few iterations close to the actual value. Second, the final value estimate is independent of the starting estimate. The graph shows that the iterative technique can be applied from any initial estimate and still achieve accurate results. We note that ten iterations are more than sufficient to achieve a good and stable estimate. Further, we see that the conservative analysis converges on a value slightly higher than the simulation value, as expected.

Fig. 12 shows the estimates and measured periods of all ten applications used in this case-study for different techniques. The estimates are normalized to the results achieved in the simulation. The results of the worst-case-waiting-time (14) are not shown on this graph, since they are more than two or three times the simulation results and putting them on the same scale makes the other results unreadable.

The figure shows that the iterative analysis is accurate for all the applications in this use-case. After ten iterations, the maximum error that can be seen is about 3% (in Application H), and the average error is less than 2%. On the other hand, in the exponential approach prediction, the average error is 10%, and the maximum error is 29% in the same application. Another observation we can make is that the estimate provided by the conservative iterative technique is always higher than the simulation result. On average, the conservative approach overestimates the period by about 8%—a small price to pay when compared to the worst-case bound that is 162% over-estimated.

The error in the iterative analysis (defined as mean absolute difference) is presented in Table I. Both the average and the maximum error are shown. Different starting points for the iterative analysis are taken. A couple of observations can be made from the table. Regardless of the starting estimate, the iterative analysis always converges. If we define 2% error margin as acceptable, we find that the fourth-order estimate requires only four iterations to converge while others require six iterations. However, obtaining the estimate of the fourth-order analysis is computationally intensive. Using the worst-case or the original period itself as the starting point for the iterative analysis saves the initial computation time, but takes a little longer to converge. Another observation we can make is

TABLE I
MEASURED INACCURACY FOR PERIOD IN % AS COMPARED WITH SIMULATION RESULTS FOR ITERATIVE ANALYSIS

Iterations	Fourth Order	Worst Case	Original	Conservative
0	9.9 (28.9)	72.6 (83.1)	163 (325)	72.6 (83.1)
1	6.7 (17.6)	88.4 (144)	12.6 (36)	252 (352)
2	3.5 (11.9)	6.3 (17.6)	6.7 (23.2)	7.9 (23.2)
3	2.9 (6.2)	4.5 (11.9)	4.3 (13.3)	8.8 (24.7)
4	**2** (4.8)	2.5 (7.7)	3.1 (9.1)	8.4 (23.2)
5	1.9 (3.9)	2.2 (4.8)	2.5 (6.2)	8.3 (23.2)
6	1.6 (3.6)	**1.7** (3.4)	**2** (4.8)	8.1 (21.8)
7	1.9 (4)	1.8 (3.4)	1.7 (3.9)	8 (21.8)
8	1.7 (3.6)	1.7 (3.4)	1.8 (3.6)	8 (21.8)
9	1.9 (3.4)	1.7 (3.6)	1.7 (3.4)	8 (21.8)
10	1.7 (3.4)	1.3 (3.1)	1.9 (3.4)	8.1 (21.8)

The maximum error is shown in brackets.

that in general, there is not much change after five iterations. Thus, five iterations present a good compromise between the accuracy and the execution time.

B. Dynamic Execution Times

Many applications are dynamic in nature. When there is a variation in the execution time of the application tasks, the SDF graph is not able to capture their exact behavior. The techniques that are conventionally used to analyze the application behavior give an even more pessimistic bound. To evaluate the performance of our technique, we re-ran the simulation by using dynamic execution time of the application tasks. Two sets of experiments were done—one with a uniform variation of up to 40% from the mean execution time and another with up to 80% deviation. Fig. 13 shows the results of experiments with dynamic execution times. We observe that the period of applications when execution time is allowed to vary does not change too much. In our experiments it varies by at most 2%. Clearly, it may be possible to construct examples in which it does vary significantly, but this behavior was not observed in our applications. Further, the conservative analysis still gives results that are more than the period of applications with variable execution times. In this figure, we also see the difference between applying conservative analysis throughout the ten iterations, and applying this analysis for only the last iteration. While in the former case, the prediction is sometimes very close to the measured results (Application C) and sometimes very far (Application H), in the latter the results make a nice envelope that is on average 10% more than the measured results.

C. Mapping Multiple Actors

So far we only considered cases when one actor per application is mapped on one processor. Since each application in the experiment contained up to ten actors, we needed ten processors. Clearly, this is not always efficient. Therefore, we mapped all actors of an application randomly on a 4-processor systems and checked if the iterative approach still works in that case. Since we do not consider intra-task dependency, the analysis remains the same, except that there are potentially more actors on any processor causing contention. For this experiment, we used four processors. Fig. 14 shows the comparison of the predicted results with the measured performance. The average error (mean absolute deviation) in this experiment is just 1%, while the maximum deviation is 3%. This shows that the approach is effective even when multiple actors of the same application are mapped on a resource. Further, in this experiment some processors had up to 30 actors mapped. This shows that the approach scales well with the number of actors mapped on a processor.

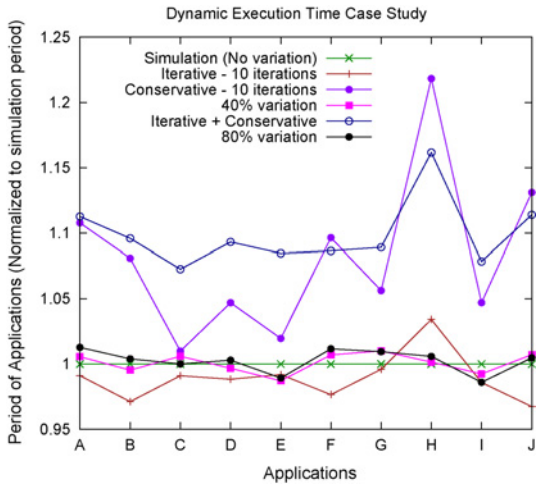


Fig. 13. Comparison of periods with variable execution time for all applications. A new conservative technique is applied; the conservation mechanism is used only for the last iteration after applying the base iterative analysis for ten iterations.

D. Mobile Phone Case-Study

In this section, we present results of a case-study with real-life applications. We did not do any optimization to the application specifications and granularity obtained from the literature to avoid favoring our approach. We consider five applications—video encoding (H263) [21], video decoding [7], JPEG decoding [24], modem [25], and a voice call scenario. These applications represent a set of typical applications—often executing concurrently—on a modern mobile phone. Sufficient buffer-space is assumed to be present among all channels in the applications, such that applications do not deadlock due to lack of buffer-space. This buffer-space on each channel (just enough to avoid deadlock) and auto-concurrency of one was modeled in the application graphs to compute the initial throughput using the SDF^3 tool.

This set of applications poses a major challenge for performance prediction since they consist of tasks with varying granularity of execution times, e.g., the *anti-aliasing* actor of *MP3 decoder* takes 40 time-units while its *sub-inversion* actor requires 186 500 time units. Further, the repetition vectors of these applications vary significantly. While the sum of repetition vector entries of *JPEG* is 26, i.e., actors of *JPEG* have to compete for processor resources to become available 26 times for one iteration, the sum of repetition vector entries of *H263 decoder* is 1190. Further, the number of tasks in each application vary significantly. While *H263 decoder* has only four tasks, the modem application has a total of 14 tasks. For this case-study, one task was mapped to one processor for each application, since multiple actor mapping options would have resulted in a huge number of potential mappings. This implied that while some processors had up to five actors, some

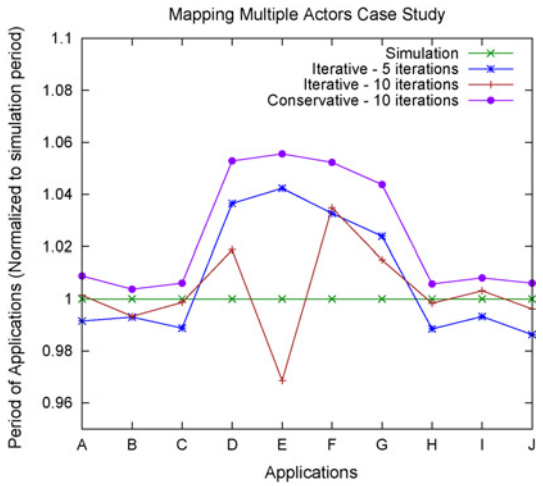


Fig. 14. Comparison of application periods when multiple actors of one application are mapped on one processor.

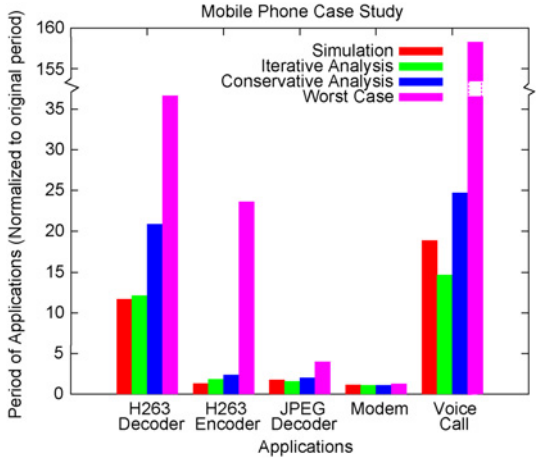


Fig. 15. Comparison of performance observed in simulation as compared to the prediction made with iterative analysis for applications in a mobile phone.

processors had only one actor. Thus, this case-study presents a big challenge for any performance prediction mechanism, and our iterative probabilistic technique was used to predict performance of these applications executing concurrently.

Fig. 15 shows the comparison between the prediction of the iterative analysis and the simulation result.[3] The results of the bound provided by the worst-case estimate are also shown for comparison. A couple of observations can be made from the graph. First of all, the period of applications increases in different proportions when executing concurrently with other applications. While the period of *modem* application increases by only 1.1 times, the period of *H263 decoder* increases by about 12 times, and that of a *voice call* by about 18 times. This depends on the granularity of tasks, the number of tasks a particular application is divided into, and the mapping of tasks on the multiprocessor platform. The modem application consists of about 14 tasks, but only six of them experience contention. The remaining eight tasks have a dedicated processor, and therefore have no waiting time. Further, the six tasks that do share a processor are only executed once per application iteration. In contrast, the *inverse-quantization* actor of the *H263 decoder* executes 594 times per iteration of the decoder, and has to wait for the processor to become available each time. This causes significant degradation in its performance. The second observation we can make is that the iterative analysis is still very accurate. The average deviation in throughput estimate is about 15%, and the maximum deviation is in the *voice call* application of 29%. The worst-case estimate in contrast is up to 18 times overly pessimistic. It should be mentioned that in this experiment FCFS arbitration was used. A different arbitration mechanism and a better mapping can distribute the resources more evenly.

[3]For these results, a bar chart is used instead of lines to make the graph more readable. Using a line would squeeze all the points of the *modem*, for example, to a single point. Further, it is difficult to make the gap in *y*-axis (needed for *voice call*) meaningful using lines.

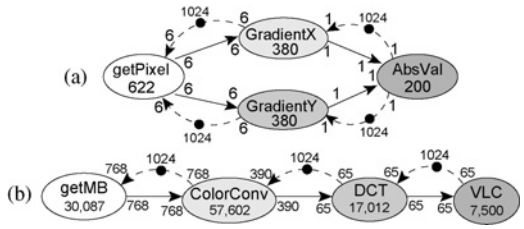


Fig. 16. SDF model of Sobel algorithm for one pixel, and JPEG encoder for one macroblock. (a) Sobel. (b) JPEG.

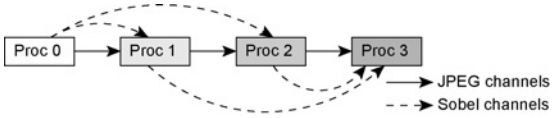


Fig. 17. Architecture of the generated hardware to support Sobel and JPEG encoder.

E. Comparison With an FPGA Multiprocessor Implementation

In addition to POOSL and the analysis approaches, we also used a prototyping approach (as presented in Fig. 1) to test performance of multiple applications on a real hardware multiprocessor platform.[4] A Microblaze-based multiprocessor platform was built using the MAMPS tool [19], [26]. This tool generates desired architecture for Xilinx-based FPGAs using their soft-processor (Microblaze) and point-to-point connections for data transfers using fast simplex links—FIFOs. Application C-code for the corresponding processors is then used and performance is measured for multiple applications. Here, we present results for Sobel (edge-detection algorithm) and JPEG encoding applications.

Fig. 16 shows the SDF model for Sobel and JPEG encoders. The Sobel model is based on pixel-level granularity while the JPEG model is based on macro-block granularity. The execution times shown in this figure are obtained by profiling the C-code of the corresponding applications on Microblaze processors, and include the communication delay for sending and receiving the data as well. As can be seen, the two applications have very different granularity of actors and poses a challenge for any analysis algorithm. Fig. 17 shows the generated hardware platform to support these two applications. The dedicated point-to-point links generated are shown by

[4]The applications presented earlier are too big to be accommodated in our FPGA multiprocessor platform.

TABLE II
PERIOD OF CONCURRENTLY EXECUTING SOBEL AND JPEG ENCODER APPLICATIONS AS MEASURED OR ANALYZED

Application	FPGA	POOSL		Iterative P^3	
		Period	Error	Period	Error
Sobel	17 293	17 134	0.92%	16 589	4%
JPEG Enc.	103 672	104 451	0.75%	103 686	0.01%

TABLE III
NUMBER OF CLOCK CYCLES CONSUMED ON A MICROBLAZE PROCESSOR DURING VARIOUS STAGES, THE PERCENTAGE OF ERROR (BOTH AVERAGE AND MAXIMUM), AND THE COMPLEXITY

Algorithm/Stage	Clock Cycles	Error in % Avg (Max)	Complexity
Load from CF card	1 903 500	–	$O(N.n.k)$
Throughput computation	12 688	–	$O(N.n.k)$
Worst case	2090	72.6 (83.1)	$O(m.M)$
Exponential—fourth order	1 740 232	9.9 (28.9)	$O(m^4.M)$
Iterative—one iteration	15 258	12.6 (36)	$O(m.M)$
Iterative—one iteration*	27 946	12.6 (36)	$O(m.M + N.n.k)$
Iterative—five iterations*	139 730	2.2 (3.4)	$O(m.M + N.n.k)$
Iterative—ten iterations*	279 460	1.9 (3.0)	$O(m.M + N.n.k)$

* Including throughput computation time.
N: number of applications; n: number of actors in an application; k: number of throughput equations for an application; m: number of actors mapped on a processor; M: number of processors.

arrows. All links had a buffer-capacity of 1024 integers. This buffer limitation is modeled as a back-edge in the application graph in Fig. 16. Table II shows the period of both the applications as measured on this FPGA platform. The estimates obtained by simulating the SDF models using POOSL and our proposed iterative technique are also shown. The errors in these estimates in comparison with the results measured on the FPGA board are also shown. The error in estimates from POOSL is less than a percent while the maximum error in our iterative technique results is 4%.

F. Implementation Results on an Embedded Processor

One of the main benefits of this approach is the speed and accuracy that makes it ideal for run-time resource management. In order to precisely compute the delay on an embedded processor, the proposed algorithms were ported to Microblaze—a soft-core provided by Xilinx. This required some rewriting to optimize the implementation for timing and reduced memory use. The default time taken for the exponential approach to fourth-order approximation, for example, was 72M cycles. Table III shows the time taken during various stages and algorithms *after* rewriting. The algorithmic complexity of each stage and the error as compared to the simulation result is also shown.

The error in various techniques as compared to the performance achieved is also shown in Table III. As can be seen, the exponential analysis with fourth order gives an average error of about 10% and a maximum error of 29%. The iterative technique after just five iterations predicts a performance that is within 2% of the measured performance on average and has only 3% maximum deviation in the entire set of applications.

The loading of application properties from the compact flash (CF) card took the most amount of time. However, this is only done once at the start of the system, and hence does not cause any bottleneck. On a system operating at 500 MHz, it takes about 4 ms to load the applications-specification. Parametric throughput computation is quite fast, and takes about 12 K cycles for all ten applications. For the iterative analysis, each iteration takes only 15 K cycles. If five iterations are carried out, it takes a total of 140 K cycles for all ten applications, including the time spent in computing throughput. This translates to about 300 μs on a 500 MHz processor when the performance of all ten applications is computed. Since starting a new application is likely to be done only once in every few minutes, this is a small overhead. In comparison, the exponential approach takes about 3.5 ms, i.e., about 12 times more time.

VI. RELATED WORK

In [27], the authors propose to analyze the performance of a *single application* modeled as an SDF graph by decomposing it into a homogeneous SDF graph (HSDFG) [9]. The throughput is calculated based on analysis of each cycle in the resulting HSDFG [28]. However, this can result in an exponential number of vertices [29]. Thus, algorithms that have a polynomial complexity for HSDFGs have an exponential complexity for SDFGs. Algorithms have been proposed to reduce average case execution time [17], but it still takes $O(n^2)$ in practice where n is the number of vertices in the graph. When mapping needs to be considered, extra edges can be added to model resource dependences such that a complete analysis taking resource dependences into account is possible. However, the number of ways this can be done even for a single application is exponential in the number of vertices [30]; for multiple applications the number of possibilities is infinite. Further, only static order arbitration can be modeled using this technique.

For *multiple applications*, an approach that models resource contention by computing worst-case-response-time (WCRT) for time division multiple access scheduling (requires preemption) has been analyzed in [31]. This analysis gives a very conservative bound. Further, this approach requires preemption for analysis. A similar worst-case analysis approach for round-robin is presented in [21], which also considers non-preemptive systems, but suffers from the same problem of lack of scalability. WCRT is computed by adding the execution times of all the actors mapped on a resource. However, as the number of applications increases, the bound increases much more than the average case performance, as also shown in Section V. Real-time calculus has also been used to provide worst-case bounds for multiple applications [32]–[34]. Besides providing a very pessimistic bound owing to cyclic dependences in an SDF graph, the analysis is very intensive and requires a very large design-time effort. On the other hand, our approach is very simple. However, we should note that the above approaches give a worst-case bound that is targeted at hard-real-time (RT) systems.

A common way to use probabilities for modeling dynamism in application is to use stochastic task execution times [35]–[37]. 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 and provide equivalent model for dynamic task execution times with uniform distribution. To the best of our knowledge, there is no efficient approach of analyzing multiple applications on a non-preemptive heterogeneous multiprocessor platform. A technique has been presented in [22] to also model and analyze contention, but the approach in this paper is much better. The technique in [22] looks at all possible combinations of actors blocking another actor. Since the number of combinations is exponential in the number of actors mapped on a resource, the analysis has an *exponential* complexity. The IP^3 approach, on the other hand, computes how much a particular actor contributes to the waiting time of the other actors. This has *linear* complexity in the number of actors.

Queuing theory also allows computing the waiting times when several processes are being served by a resource [38]; it has been applied for networks [39] and processor-sharing [40]. However, this is not applicable in our scenario for a number of reasons. First, since we have circular dependences in the SDF graphs, feedback loops are created that cannot be handled by the queuing theory. Second, the execution time of tasks on a processor does not follow a common distribution. Each task may have an independent execution time distribution. Therefore, a general expression for the service time for tasks mapped on a processor cannot be determined. The same reason makes it hard to apply queuing theory when applications are modeled as petri-nets.

VII. Conclusion and Future Work

In this paper, we presented a new probabilistic technique to estimate the performance of applications when sharing resources. An iterative analysis is presented that can predict the performance of applications very accurately. Besides, a conservative flavor of the iterative analysis is presented that can also provide conservative predictions for applications for which the mis-prediction penalty may be high.

An experiment with ten random applications concurrently executing shows the average error in prediction using iterative probability to be less than 2% and the maximum error as 3%. Further, it takes about four to six iterations for the prediction to converge. The execution-time complexity of the algorithm is low—it takes only 300 μs with ten applications on a 500 MHz processor. The implementation results on an embedded processor show that the iterative technique outperforms the earlier exponential technique in [22]—it requires 12 times less compute time, and shows better accuracy. The accuracy of the approach is validated with Sobel and JPEG encoder applications executing concurrently on an FPGA multiprocessor system. However, it should be mentioned that this system only supports point-to-point connections; therefore, the network contention is limited.

Further, we presented results of a case-study of applications commonly used in a mobile phone. The models of these applications vary in the number of tasks, granularity of tasks, and also their repetition vectors differ largely. Even in this particular use-case, the prediction by iterative analysis is close to the simulation result. This shows the robustness of the technique. We also see that applications with coarser task granularity perform better in the first-come-first-serve arbitration as compared to applications that have a finer granularity. This occurs since the tasks with finer granularity have to compete for resources more often. Different arbitration mechanisms can potentially alleviate this problem, and more research should be done into that. One of the limitations of this approach is that it does not provide any guarantees. In future, we intend to extend our technique to provide probabilistic guarantees for soft real time tasks. Yet another limitation is that we only consider contention for processor(s) shared by multiple tasks. Contention caused by shared bus and I/O devices is not considered enough. Such contention is left as future work.

Acknowledgment

The authors would like to thank the reviewers for their valuable feedback which has led to an improved paper. They would also like to thank M. Geilen for discussions regarding probability, and A. Shabbir for his help in FPGA implementation of multiprocessor platform and profiling.

References

[1] J. Wawrzynek, D. Patterson, M. Oskin, S. Lu, C. Kozyrakis, J. Hoe, D. Chiou, and K. Asanovic, "RAMP: Research accelerator for multiple processors," *IEEE Micro*, vol. 27, no. 2, pp. 46–57, Mar.–Apr. 2007.

[2] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[3] S. Davari and S. K. Dhall, "An on line algorithm for real-time tasks allocation," in *Proc. IEEE Real-Time Syst. Symp.*, 1986, pp. 194–200.

[4] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, Jun. 1996.

[5] K. Jeffay, D. Stanat, and C. Martel, "On non-preemptive scheduling of periodic and sporadic tasks," in *Proc. 12th IEEE Real-Time Syst. Symp.*, 1991, pp. 129–139.

[6] S. Stuijk, M. Geilen, and T. Basten, "SDF3: SDF for free," in *Proc. 6th Int. Conf. Applicat. Concurrency Syst. Design (ACSD)*, 2006, pp. 276–278.

[7] S. Stuijk, "Predictable mapping of streaming applications on multiprocessors," Ph.D. dissertation, Dept. Electron. Syst., Eindhoven Univ. Technol., Eindhoven, The Netherlands, 2007.

[8] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, Jan. 1987.

[9] S. Sriram and S. Bhattacharyya, "Background terminology and notation," in *Embedded Multiprocessors: Scheduling and Synchronization*. New York: Marcel Dekker, 2000, pp. 31–53.

[10] S. Stuijk, M. Geilen, and T. Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous data flow graphs," in *Proc. Design Autom. Conf.*, 2006, pp. 899–904.

[11] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—Overview of methods and survey of tools," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, pp. 1–53, Apr. 2008.

[12] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprette, "System design using Kahn process networks: The Compaan/Laura approach," in *Proc. Design Autom. Test Europe Conf.*, 2004, pp. 340–345.

[13] J. Cockx, K. Denolf, B. Vanhoof, and R. Stahl, "Sprint: A tool to generate concurrent transaction-level models from sequential code," *EURASIP J. Appl. Signal Process.*, vol. 2007, no. 1, p. 213, Jan. 2007.

[14] S. Rul, H. Vandierendonck, and K. De Bosschere, "Function level parallelism driven by data dependences," *ACM SIGARCH Comput. Architecture News*, vol. 35, no. 1, pp. 55–62, Mar. 2007.

[15] P. Yang, P. Marchal, C. Wong, S. Himpe, F. Catthoor, P. David, J. Vounckx, and R. Lauwereins, "Managing dynamic concurrent tasks in embedded real-time multimedia systems," in *Proc. 15th Int. Symp. Syst. Synthesis*, 2002, pp. 112–119.

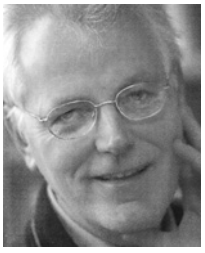
[16] R. Burden and J. Faires, "Error analysis for iterative methods," in *Numerical Analysis*, 8th ed. Pacific Grove, CA: Cole, 2005, ch. 2.4, pp. 75–83.
[17] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. G. Bekooij, "Throughput analysis of synchronous data flow graphs," in *Proc. 6th Int. Conf. Applicat. Concurrency Syst. Design*, 2006, pp. 25–36.
[18] A. H. Ghamarian, M. C. W. Geilen, T. Basten, S. Stuijk, "Parametric throughput analysis of synchronous data flow graphs," in *Proc. Design Autom. Test Europe*, 2008, pp. 116–121.
[19] MAMPS. (2008). "Multiple applications multiprocessor synthesis" [Online] Available: http://www.es.ele.tue.nl/mamps
[20] B. D. Theelen, O. Florescu, M. C. W. Geilen, J. Huang, P. H. A. van der Putten, and J. P. M. Voeten, "Software/hardware engineering with the parallel object-oriented specification langauge," in *Proc. Int. Conf. Formal Methods Models Codesign*, 2007, pp. 139–148.
[21] R. Hoes, "Predictable dynamic behavior in NoC-based MPSoC" M.S. thesis, Eindhoven Univ. Technol., Eindhoven, The Netherlands, 2004.
[22] A. Kumar, B. Mesman, H. Corporaal, B. Theelen, and Y. Ha, "A probabilistic approach to model resource contention for performance estimation of multifeatured media devices," in *Proc. Design Autom. Conf.*, 2007, pp. 726–731.
[23] D. Terr and E. W. Weisstein. (2008). "Symmetric polynomial" [Online]. Available: mathworld.wolfram.com/SymmetricPolynomial.html
[24] E. de Kock, "Multiprocessor mapping of process networks: A JPEG decoding case-study," in *Proc. 15th Int. Symp. Syst. Synthesis*, 2002, pp. 68–73.
[25] S. Bhattacharyya, P. Murthy, and E. Lee, "Synthesis of Embedded Software from Synchronous Dataflow Specifications," *J. VLSI Signal Process.*, vol. 21, no. 2, pp. 151–166, 1999.
[26] A. Kumar, S. Fernando, Y. Ha, B. Mesman, and H. Corporaal, "Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, no. 3, pp. 1–27, Jul. 2008.
[27] N. Bambha, V. Kianzad, M. Khandelia, and S. S. Bhattacharrya, "Intermediate representations for design automation of multiprocessor DSP systems," *Design Automat. Embedded Syst.*, vol. 7, no. 4, 2002, pp. 307–323.
[28] A. Dasdan, "Experimental analysis of the fastest optimum cycle ratio and mean algorithms," *ACM Trans. Design Autom. Electron. Syst.*, vol. 9, no. 4, pp. 385–418, Oct. 2004.
[29] J. Pino and E. Lee, "Hierarchical static scheduling of data flow graphs onto multiple processors," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, vol. 4. 1995, pp. 2643–2646.
[30] A. Kumar, B. Mesman, B. Theelen, H. Corporaal, and Y. Ha, "Analyzing composability of applications on MPSoC platforms," *J. Syst. Archit.*, vol. 54, nos. 3–4, pp. 369–383, Mar.–Apr. 2008.
[31] 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*. Berlin, Germany: Springer, 2005, pp. 81–108.
[32] K. Richter, M. Jersak, and R. Ernst, "A formal approach to MPSoC performance verification," *Computer*, vol. 36, no. 4, pp. 60–67, Apr. 2003.
[33] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2000, pp. 101–104.
[34] S. Kiinzli, F. Poletti, L. Benini, and L. Thiele, "Combining simulation and formal methods for system-level performance analysis," in *Proc. Design Autom. Test Eur.*, vol. 1. 2006, pp. 1–6.
[35] L. Abeni and G. Buttazzo, "QoS guarantee using probabilistic deadlines," in *Proc. 11th Euromicro Conf. Real-Time Syst.*, 1999, pp. 242–249.
[36] S. Manolache, P. Eles, and Z. Peng, "Schedulability analysis of applications with stochastic task execution times," *ACM Trans. Embedded Comput. Syst.*, vol. 3, no. 4, pp. 706–735, Nov. 2004.
[37] S. Hua, G. Qu, and S. S. Bhattacharyya, "Probabilistic design of multimedia embedded systems," *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 3, p. 15, 2007.
[38] L. Takacs, *Introduction to the Theory of Queues*. New York: Oxford Univ. Press, 1962 (reprinted by Greenwood Press in 1982).
[39] T. Robertazzi, "Stochastic Petri nets," in *Computer Networks and Systems: Queueing Theory and Performance Evaluation*. Berlin, Germany: Springer, 2000, pp. 237–275.
[40] J. E. G. Coffman, R. R. Muntz, and H. Trotter, "Waiting time distributions for processor-sharing systems," *J. ACM*, vol. 17, no. 1, pp. 123–130, 1970.

Akash Kumar (M'09) received the B.Eng. degree in computer engineering from the National University of Singapore (NUS), Singapore, in 2002. He received the joint Master of Technological Design degree in embedded systems from NUS and the Eindhoven University of Technology (TUe), Eindhoven, The Netherlands, in 2004, and received the joint Ph.D. degree in electrical engineering in the area of embedded systems from TUe and NUS, in 2009.

In 2004, he was with Philips Research Labs, Eindhoven, The Netherlands, where he worked on Reed Solomon codes as a Research Intern. From 2005 to 2009, he was with TUe as a Ph.D. student. Since 2009, he has been with the Department of Electrical and Computer Engineering, NUS, currently as a Visiting Fellow. He has published over 25 papers in leading international electronic design automation journals and conferences. His current research interests include analysis, design methodologies, and resource management of embedded multiprocessor systems.

Bart Mesman received the B.Eng. and M.Eng. degrees in electrical engineering, and the Ph.D. degree from the Eindhoven University of Technology, Eindhoven, The Netherlands, in 1993, 1995, and 2001, respectively. His thesis discusses an efficient constraint-satisfaction method for scheduling operations on a distributed very long instruction word processor architecture with highly constrained register files with stringent timing requirements.

From 1995 to 2005, he was with Philips Research Labs, Eindhoven, The Netherlands, as a Research Engineer where he worked on digital signal processing processor architectures and compilation. He is currently with the Eindhoven University of Technology, Eindhoven, as a Researcher. His current research interests include (multi)processor architectures, compile-time and run-time scheduling, and resource management in multimedia devices.



Henk Corporaal (M'09) received the B.S. degree in mathematics and natural science, the M.S. degree in theoretical physics from the University of Groningen, Groningen, The Netherlands, in 1977 and 1982, respectively, and the Ph.D. degree in electrical engineering, in the area of computer architecture, from the Delft University of Technology, Delft, The Netherlands, in 1995.

He has been teaching at several schools of higher education, and has been an Associate Professor with Delft University of Technology from 1996 to 2001 in the field of computer architecture and code generation, a Joint Professor appointment with National University of Singapore from 2001 to 2005, the Scientific Director of the joined NUS–TUE Design Technology Institute from 2003 to 2005, a Research Fellow with NEC CCRL, Princeton, in 1999, and a Visiting Professor with IISc, Bangalore, India, from 1997 to 1998. Currently, he is a Professor in embedded system architectures with the Einhoven University of Technology (TUe), Einhoven, and a Member of Netherlands Institute for Research on Information and Communication Technology (ICT), the Dutch Institute for Research in ICT, and in the management of PROGRESS, the Dutch applied research program on embedded systems. He has co-authored over 200 international journal and conference papers in the multiprocessor architecture and embedded system design area. His current research interests include the predictable design of soft and hard real-time embedded systems.

Yajun Ha (SM'09) received the B.S. degree in electrical engineering from Zhejiang University, Hangzhou, China, in 1996, the M.Eng. degree in electrical engineering from the National University of Singapore (NUS), Singapore, in 1999, and the Ph.D. degree in electrical engineering from Katholieke Universiteit Leuven, Leuven, Belgium, in 2004.

He has been an Assistant Professor with the Department of Electrical and Computer Engineering, NUS, since 2004. Between 1999 and 2004, he did his Ph.D. research project at the Interuniversity Microelectronics Center, Leuven. He has held a U.S. patent and published more than 50 internationally refereed technical papers in his interested areas. His current research interests include embedded system architecture and design methodologies, particularly in the area of reconfigurable computing.