Execution-time Prediction for Dynamic Streaming Applications with Task-level Parallelism

Peter Poplavko[1,2], Twan Basten[2], Jef van Meerbergen[2,3]
[1]Magma Design Automation, Eindhoven, The Netherlands
poplavko@magma-da.com
[2]Electronic Systems Group, Eindhoven University of Technology, Eindhoven, The Netherlands
[3]Philips Research Laboratories, Eindhoven, The Netherlands

Abstract— **Programmable multiprocessor systems-on-chip are becoming the preferred implementation platform for embedded streaming applications. This enables using more software components, which leads to large and frequent dynamic variations of data-dependent execution times. In this context, accurate and conservative prediction of execution times helps in maintaining good audio/video quality and reducing energy consumption by dynamic evaluation of the amount of on-chip resources needed by applications. To be effective, multiprocessor systems have to employ the available parallelism. The combination of task-level parallelism and task delay variations makes predicting execution times a very hard problem. So far, under these conditions, no appropriate techniques exist for the conservative prediction of execution times with the required accuracy. In this paper, we present a novel technique for this problem, exploiting the concept of scenario-based prediction, and taking into account the transient and periodic behavior of scenarios and the effect of scenario transitions. In our MPEG-4 shape-decoder case study, we observe no more than 11% average overestimation.**

I. INTRODUCTION

In modern embedded systems, more and more streaming audio/video applications are implemented in software. This trend is driven by the increased hardware performance and the growing need to save in hardware costs. Important platforms for embedded system software are multiprocessor systems-on-chip (MP-SoC) ([13]).

The fact that certain software subroutines manifest data-dependent execution delays, leads to variations in *execution time*, i.e. the time it takes to process one frame. (Note that we use 'frame' with a more general meaning than 'video frame'.) Execution time variations pose a challenging problem, namely, the efficient use of the hardware resources given that the resource requirements change over time. This concern is especially relevant for embedded systems, having stringent low-cost low-power requirements.

An important approach to cope with this problem is *execution time prediction*. A special system unit, the resource or quality manager, predicts the execution times. When the execution times get smaller, the manager can deallocate resources or switch the involved processors into to a low-power mode. If execution times grow, the manager can allocate extra resources, or switch the application to a different quality mode with some degradation in perceived (video/audio) quality.

To take advantage of MP-SoCs, task-level parallelism must be exploited by running different tasks on different processors. The parallel realization of streaming applications is usually expressed using dataflow graphs, where the application tasks are modeled as graph nodes, called *actors*. Actors communicate by sending *data tokens* (blocks of data) through first-in-first-out (FIFO) channels. To process a frame, each actor performs multiple executions per frame (typically 10-1000 for video applications).

In this paper, we focus on so-called Homogeneous Synchronous Dataflow (*HSDF*) graphs [12], which are popular in multiprocessor scheduling [2]. We allow dynamic data-dependent actor execution delays, which makes HSDF graphs very useful to express dynamic streaming applications. Our reason to consider HSDF graphs is that they allow *analytical* performance estimation. However, for dynamic applications, the performance analysis by default yields only worst-case estimates, and the wider the dynamic range of execution-time variations the worse the accuracy of this approach.

We propose a novel analysis technique which greatly improves the accuracy of execution time estimates for dynamic multiprocessor applications compared to worst-case estimates, while still giving conservative results. Our approach is based on the *scenario-based* prediction paradigm [10, 21]. This paradigm is based on the observation that the dynamic behavior of an application is typically composed of a limited number of sub-behaviors, i.e., *scenarios*, that have similar resource requirements, i.e., similar actor execution delays in the context of this paper. The execution of a dynamic streaming application consists of a sequence of intervals in which some specific scenario is active. Within such an interval, the behavior converges to a steady-state periodic pattern after some initial transient phase, caused by the fact that the transitions from one scenario interval to another one typically occur in a pipelined manner and have a non-negligible duration. The key to the accuracy of our method is that it takes into account the transient and steady-state periodic behavior of scenarios and scenario transitions. To the best of our knowledge, our technique is the only conservative execution-time prediction technique with good accuracy that can handle dynamic task delays in combination with all forms of task-level parallelism, including pipelining and data parallelism.

The rest of this paper is organized as follows. Section II motivates our topic and goals. Section III analyzes related work. Our approach is introduced in Section IV. Section V works out our execution time prediction method in detail. Section VI describes a case study and gives experimental results. Section VII summarizes and concludes the paper.

II. PROBLEM CONTEXT

A. Implementation Trajectory

In this subsection, we place the execution time prediction problem into the context of a typical MP-SoC implementation trajectory. At design time, the trajectory runs a mapping flow, generating actor binding, scheduling and communication decisions. The mapping flow is beyond the scope of this paper; see [11, 16§5, 18], for examples. It is augmented with an analysis flow, which generates the execution time predictions and runs partially at design time and partially at run time. The analysis flow is the main topic of this paper. The execution time predictions generated by this flow may be used by any form of resource- and/or quality management, see e.g. [13].

We assume that an application is specified by its HSDF task graph and the periodic *frame deadlines*. A *frame* is thus defined as the data unit for which deadlines are specified. Deadlines for multimedia streaming applications commonly refer to coarse-grain data units, containing multiple elementary units. We therefore assume that a frame consists of multiple data tokens; e.g., a video frame consists of video blocks.

To support real-time constraints, we require that the MP-SoC platform can provide guaranteed computation, communication and memory budgets for applications. Task graph actors should be assigned to processors statically. When different actors share the same processor, we either enforce a static order of actors or use scheduling techniques giving a guaranteed percentage of the processor cycles to different actors (e.g. TDMA [17]). To avoid communication resource conflicts, we assume a distributed-memory MP-SoC architecture, consisting of processors coupled with their local memories, integrated as tiles by an interconnection network, such as a (segmented) bus, or a network-on-chip. Actors running at different tiles use separate local buses and memories, without conflicting with each other. Conflicts in the interconnection network can be avoided e.g. via reserved connections. Examples of architectures satisfying the sketched requirements are Cradle [7], PROPHID [17], and Hijdra [4].

The analysis flow is divided into an actor-level and a graph-level stage. This division is useful because execution delay variations of the same magnitude taking place at different actors can have different impacts on the timing behavior of the graph. The actor-level analysis exposes the data-dependent dynamic behavior of the actors. The graph-level stage integrates the individual behaviors of different actors to produce the overall frame execution time prediction.

The analysis flow takes into account the decisions on how the actors are mapped and scheduled on the multiprocessor platform and how the communication is organized. In our flow, we model the implementation decisions using a special HSDF graph, called an *IPC graph*, where 'IPC' stands for inter-processor communication [2, 14, 16§7.1]. In [14], it is shown that IPC graphs can model an application mapping onto an MP-SoC as assumed in this paper. Compared to the task graph, an IPC graph may contain extra actors and edges, which model, for instance, the ordering of actors per processor and the communication through the interconnection network.

It is important to mention that, although different applications may share some hardware resources, they are modeled by different independent IPC graphs. This is possible because guaranteed computation, memory, and communication budgets are assigned to applications.

B. Objectives: Conservatism and Accuracy

From the resource and quality management perspective, the results of performance prediction should be conservative and accurate. Conservatism (guaranteed, or with a high probability) is needed to avoid that a frame being processed misses its deadline, which implies a waste of computing resources. Any conservative method is necessarily based on some analytical approach. Accuracy is important because overestimations of execution time may for example lead to the unnecessary downscaling of video/audio quality, the unnecessary selection of high operating frequencies, or the allocation of too many resources. Conservatism and accuracy are to some extent conflicting requirements. In this paper, we emphasize conservatism and analytical reasoning, while achieving a considerable accuracy improvement over worst-case execution time predictions, which makes our work applicable for practical resource and quality management.

III. RELATED WORK

A. Execuction-time Prediction and Scenarios

One approach to execution time prediction is to extrapolate the frame execution time from past execution times, e.g., [20]. However, [3] claims that frame execution times of streaming applications like MPEG decoders cannot be extrapolated with sufficient accuracy. The work shows that to achieve satisfactory accuracy the input packet headers must provide some *a-priori* hints on the complexity of the upcoming frames. All approaches known to us that use a-priori hints (e.g. [3, 13]]) fit into the *scenario-based* prediction paradigm [10, 21], which we therefore choose as a foundation for our approach.

A scenario is a set of application execution behaviors with similar resource usage, processor cycles in our case. Scenario-based execution time prediction estimates the execution time via an algebraic expression in terms of *scenario coefficients,* i.e. the contributions of a scenario to the execution time, and *scenario parameters*, typically variables counting the number of invocations of the scenario. The coefficients are constant and depend on the platform hardware and scheduling, whereas the parameters are dynamic, platform-independent and specific for the given application. The parameters are, in fact, the necessary a-priori hints provided in the headers [3].

One important aspect in scenario-based execution-time prediction is *parameter identification*, i.e., defining scenario parameters. Identification is an implementation-independent – often manual – process, based on knowledge of the application; e.g., [3] identifies the number of video blocks of type 'inter' and 'intra' as parameters. [9] proposes an automated parameter identification technique that is able to detect parameters that are present in the source code.

The other challenge of scenario-based execution time prediction is *scenario characterization*, which includes

- finding an expression for the execution time, and
- calculating the scenario coefficients for that expression.

Characterization can be conservative or approximate. Conservative characterization necessarily has an analytical foundation. A shortcoming of most existing scenario characterization approaches (e.g. [3, 9]) is the lack of support for task-level parallelism and task scheduling.

Our work contributes to parameter identification and scenario characterization by generalizing these problems to and solving them for IPC graphs. We apply a scenario-based

approach both at the actor level and the graph level. For the most part, the actor-level analysis can be realized using existing techniques (Section V.B). We focus on identification of the graph-level scenario parameters, derivation of the execution-time prediction expression (Section IV), and characterization of graph-level scenario coefficients (Section V).

B. *Support for Task-level Parallelism in Multiprocessors*

The only multiprocessor-oriented scenario characterization technique we are aware of is task concurrency management (TCM) [13]. However, this work does not consider task-level pipelining and does not allow cyclic dataflow graphs, which limits the applicability of this approach.

We considered several performance estimation techniques that are suitable for characterizing a given scenario on a multiprocessor, namely, static HSDF throughput analysis [8], stochastic HSDF throughput analysis [16§7.6], Markov-chain analysis [19], and schedulability analysis [15]. These techniques provide analytical ways to calculate performance metrics of the modeled system in a state of equilibrium or the steady state.

From the perspective of dynamic resource management, the shortcoming of these steady-state analysis techniques is that they do not support dynamic parameterization. [8] requires constant actor execution delays, [16§7.6] requires a static probability distribution, [19] requires static statistical moments, and [15] requires static upper and lower task delay bounds. Consequently, these techniques can only provide conservative execution-time predictions when the parameters have constant – hence necessarily worst-case – values. For example, [8] provides fast and accurate HSDF *throughput* analysis algorithms when actor delays are constant, representing the worst-case of the actual, dynamic delays. One can use these algorithms to calculate the frame execution time as the number of data tokens in the frame divided by throughput. However, the wider the dynamic range of the actor execution delays, the worse the accuracy of this approach.

We propose an extension of HSDF throughput analysis with the elements needed for the scenario-based approach. Although our techniques work for general HSDF graphs, to illustrate their use in MP-SoC design, we explain them in the context of IPC graphs, briefly introduced in Section II.

IV. Approach: Scenario Intervals and Transitions

Each graph-level scenario is characterized by constant (and conservative) *scenario delay levels*, such that, by definition of a scenario, the real actor delay values in the given scenario stay close below the scenario delay levels. We split the execution of a frame into *scenario intervals*, such that in every interval the behavior of actors belongs to one scenario. Within each interval, we model actor delays to be equal to the scenario delay levels. We refer to this actor delay model as the *multi-scenario mode* of the IPC graph. In this mode, the execution times are (conservative) estimates of the real execution times.

IPC graph execution in multi-scenario mode is illustrated in Figure 1. By property of constant-delay HSDF graphs, inside every scenario interval, the processing of data tokens follows a periodic pattern (shown as rectangles, one per data token), which reflects the steady-state behavior of the graph in the given scenario. The pattern is characterized by the *latency* and *period*

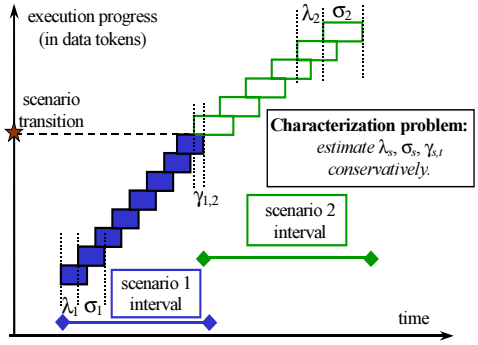

Figure 1. IPC execution in multi-scenario mode

of data token executions, denoted σ_s, and λ_s respectively, where s is the scenario identifier (see Figure 1).

At the borders of scenario intervals, called *scenario transitions*, one pattern is followed by a different one. In general, the new pattern is not established instantaneously, but after a few tokens have been processed. This initial part of the scenario interval is the *transient phase* (not illustrated in Figure 1). As shown later, we take the transient phase into account in such a way that one can safely assume that the periodic patterns are established instantaneously. To achieve good accuracy, even when scenario intervals are very short, we take into account the *timing overlap* between scenario intervals, denoted $\gamma_{s,t}$ (see again Figure 1). This overlap is caused by pipelined processing of consecutive data tokens.

With the above in mind, we can define the following estimation of the execution time 'Δ' of one frame:

$$\Delta = \sum_{\text{scenario } s} \lambda_s \cdot J_s + (\sigma_s - \lambda_s) \cdot L_s - \sum_{s,t} \gamma_{s,t} \cdot K_{s,t} \qquad (1)$$

where J_s is the total number of data tokens in scenario s over all intervals, L_s is the total number of intervals of scenario s, and $K_{s,t}$ is the number of transitions from scenario s to scenario t. The J_s, L_s, and $K_{s,t}$ are *graph-level scenario parameters*, which turn out to be a sufficient basis for accurate execution time prediction for IPC graphs (where the L_s depend on the $K_{s,t}$ and thus do not need to be provided explicitly).

Values σ_s, λ_s and $\gamma_{s,t}$ are *graph-level scenario coefficients*, and the characterization problem is their conservative calculation, based on the actor delay levels and IPC graph structure. In the sketched implementation trajectory of Section II, this problem is solved at the graph-level analysis stage. The next section presents the graph-level analysis stage in the broader context of the analysis flow.

V. The Analysis Flow

A. *Actor Behavior*

This subsection briefly introduces the timing behavior of HSDF dataflow actors, giving necessary background.

An HSDF graph **G** consists of actors v_k and edges (v_a, v_b). Edges represent channels, through which actors communicate tokens. Some edges may carry *initial tokens*. In the rest of the paper, we use the HSDF graph example of Figure 2, which is an IPC graph describing the MPEG-4 shape decoder application discussed in detail in Section VI. Initial tokens are depicted via black dots. The circled integer annotations in the figure are used later for identification purposes.

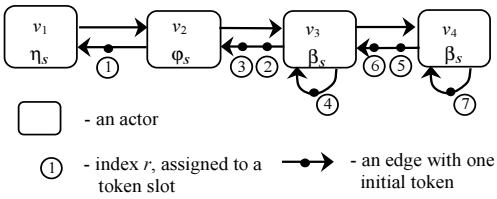

Figure 2. IPC graph of the case study

The timing behavior of an HSDF actor is described by a series of actor executions. Every execution has three stages: 1) waiting for and consumption of input tokens; 2) processing; 3) production of output tokens. At stage 1), the actor execution waits until there are input tokens in all input channels and consumes one token per input. For example, actor v_2 consumes input tokens at the edges from v_1 and v_3. Stage 2) takes a certain time, which is independent of the moment in time when the execution starts. The latter property ensures *monotonic* timing behavior of the whole graph in the sense that postponing some events in the graph can not lead to a decrease of the graph's execution time. At stage 3), the actor produces one output token per output. For example, actor v_2 produces tokens on the edges to v_1 and v_3.

The duration of stage 2) is called the *actor execution delay*. It is denoted $d(v_k, j)$, where j is an index of the actor execution. In the multi-scenario mode, the actor execution delay takes conservative values from a discrete set of scenario delay levels $\mathbf{D}(v_k) = \{d_s(v_k) | \ s \text{ a scenario}\}$. We have:

$$d(v_k, j) = d_s(v_k) \text{ if execution } j \text{ of } v_k \text{ is in scenario } s \qquad (2)$$

One can interpret the scenario levels as quantization levels. For example, our case study has three scenarios. For actor v_1, Figure 3 shows the real actor delay measured from simulation and the step-wise delay function we use to model the delays. The figure illustrates that our model is conservative. Due to the monotonic timing behavior of the HSDF graph, conservative actor delays imply conservative timing behavior of the graph. The determination of $\mathbf{D}(v_k)$ is covered in the next subsection.

An important aspect of an HSDF graph is the fact that its execution consists of an indefinite repetition of so-called *graph iterations*. In graph iteration with index j, all actors run actor executions with index j, performing different processing stages of the same input data. Thus, all actors are in the same scenario, and *the current scenario s is a function of the execution index j.*

B. *Actor-level analysis*

The purpose of the actor-level analysis is to calculate sets of scenario delay levels, $\mathbf{D}(v_k)$. Our actor-level analysis method is based on a combination of known execution delay modeling techniques, used in e.g. [3] and [21]. The resulting scenario delay levels provide input for the graph-level analysis, described in the later subsections.

The actor-level analysis begins with the identification of actor-level scenario parameters, ξ_ω, which relate to the processing of a data token. For example, actor v_1 has a parameter defining the number of bytes loaded into the input buffer when parsing the bit fields of the given data token. Actor-level parameter identification can be automated using the technique of [9] or done manually when specific application knowledge is needed.

A graph-level *scenario s* is then characterized by sets of values taken by a few most influential actor-level parameters; e.g. {' $\xi_1 \in [0,2]$ ', ' $\xi_1 \in [2,3]$ ', ' $\xi_1 = 4$ '} is a definition of

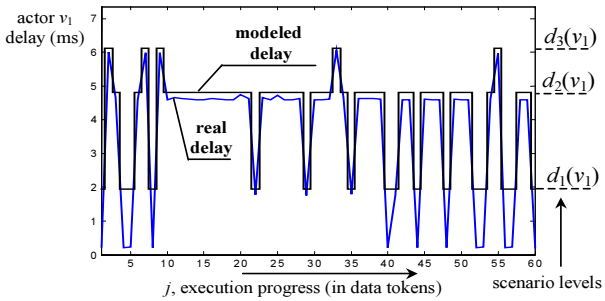

Figure 3. Actor delays in multi-scenario mode

three scenarios, assuming that ξ_1 is the most influential parameter of the given application. Subdividing the sets of parameter values into scenarios needs to be done manually, since no automated techniques exist to date [9]. This process is driven by similarities in execution delays. In Section VI, we explain how we define the scenarios in our case study.

In general, actor execution delays can be approximated as:

$$d_{\text{approx}}(v_k, j) = C_{k,0} + \sum_{\omega} C_{k,\omega} \cdot \xi_\omega(j) \qquad (3)$$

Using this approximation, we calculate the scenario delay levels $d_s(v_k) \in \mathbf{D}(v_k)$ as:

$$d_s(v_k) = C_{k,0} + \sum_{\omega} C_{k,\omega} \cdot \hat{\xi}_\omega(s) \qquad (4)$$

where the $\hat{\xi}_\omega(s)$ are the maximum (an therefore conservative) actor-level parameter values per (graph-level) scenario s.

Actor coefficients $C_{k,\omega}$ need to be calculated conservatively for the given implementation. We use design-time profiling combined with a linear regression technique with multiple variables [5], which, in theory, allows to obtain coefficients that are conservative with some required level of confidence. We use the upper bounds of 95%-confidence intervals of the regression variables. Together with the observation that incidental non-conservative coefficient values are compensated by overestimations elsewhere such as the maximization of parameters in (4), this provides a very high probability of conservative prediction for the frame execution time (100% in our experiments). This is in line with the objectives of this paper, in particular given the fact that multimedia applications can tolerate occasional deadline misses due to occasional underestimation of execution time.

Actor-level parameter values $\hat{\xi}_\omega(s)$ in (4) can be pre-calculated at design time or encoded in the input packet headers together with the graph-level parameters. The encoding in headers may allow for more accurate estimations in some cases. Equality (4) should then be applied at run time.

Linear regression to analyze task execution delays is used for instance in [3]. Calculation of scenario delay levels based on maximum values of the fine-grain parameters in the given scenario is applied for example in [21].

C. *Characterizing the Periodic Execution Pattern*

In this subsection, we analyze the timing behavior of an IPC graph **G** in scenario s, where the actor delays are modeled by conservative constant values, i.e., $d(v_k, j) = d_s(v_k)$. For convenience, we use the following notations in our example graph: $d_s(v_1) = \eta_s$, $d_s(v_2) = \varphi_s$, $d_s(v_3) = d_s(v_4) = \beta_s$. (The reason why the latter two are equal is explained in Section VI.)

Based on the actor delays, we obtain graph-level scenario coefficients σ_s and λ_s that characterize the periodic execution pattern established in a scenario interval $\mathbf{I}_p$, where p identifies the interval position in the sequence of scenario intervals. The interval consists of a set of subsequent graph iterations starting at index j_1 and belonging to the same scenario s. The number of iterations in the interval is called the *interval depth*, denoted J.

We take into account the transient phase of the graph behavior, preceding the periodic pattern at the beginning of the interval. We do that such that one can safely model the periodic pattern to be established instantaneously. To make this possible, we express the contribution of the interval to the total execution time as $\lambda_s \cdot J + (\sigma_s - \lambda_s)$ for any depth J (as if there were no transient phase) and set the latency value σ_s large enough such that the results are conservative.

To achieve our goal, we first express the timing behavior of graph **G** by a set of equations and then use these to derive σ_s and λ_s. To solve these equations, we need to know the times at which each initial token becomes available at the start of the interval. These times are called the *initial conditions*. In this subsection, we assume that at start of interval $\mathbf{I}_p$ all initial tokens are released simultaneously at a certain time T (*synchronous* initial conditions). In reality, usually only the first interval in a frame starts under synchronous conditions, because each subsequent interval depends on the previous intervals. In the next subsection, we show how to take the previous intervals into account.

Assume that graph **G** contains R initial tokens. For convenience of explanation, assume that every edge has as many slots to accommodate a token as it has initial tokens. Assume that these slots are indexed by index r=1…R. For instance, in Figure 2, the slots are annotated with r = 1…7. Then, a graph iteration can be seen as the transportation of all tokens from their current slots along the directed paths in the graph until they reach the next slot they meet along the way. We call a path along which a token moves between the slots a transfer path. For example, in Figure 2, one transfer path of the token at slot 3 is $v_2v_3v_4$, after which it ends up at slots 7 and 5, and another path is v_2, after which it ends up at slot 1. During a graph iteration, a token moves via consumptions and productions by the actors on the transfer path. If there are multiple slots that can be directly reached from the current slot, then the token multiplies and moves to all those slots. In our example, the token at slot 3 moves to slots 1, 2, 4, 5, and 7. If several tokens move to the same slot, they merge into one token. For example, tokens at slots 1 and 3 merge at slot 4.

Based on the concept of token slots, we can describe the timing behavior of the graph in interval $\mathbf{I}_p$ as follows:

for $r = 1..R$: $x_r(j_1 - 1) = T$ (initial conditions) (5)

for $r = 1..R, j=j_1..J-1$: $x_r(j) = \max_{q=1..R}\left(x_q(j-1) + \delta_{q\to r}\right)$ (6)

where $x_r(j)$ is the time a token moves into slot r in iteration j and $\delta_{q\to r}$ is the largest delay of a transfer path from slot q to slot r. The path delay is the sum of the actor delays along the path. In our example, $\delta_{6\to 5}$ is $2\beta_s$ (i.e. the sum of the delays of v_3 and v_4). If no transfer paths exist from q to r, then $\delta_{q\to r} = -\infty$. Note that the algorithmic cost to calculate $\delta_{q\to r}$ for all slot pairs using an all-pair shortest path algorithm is $O(R^3)$. Also note that in practice the right-hand sides of many equations in (6) are identical, which can be used to exclude some equations.

Equations (5) and (6) enable straightforward calculation of $x_r(j)$ for any j by first calculating them for j_1+1, then for j_1+2, etc. Based on these equalities, we obtain σ_s and λ_s for every scenario, using a fundamental theorem for HSDF graphs, stated for example in [16 §7.4]. That theorem implies that there are integers H>0 and W > 0 such that:

$j = j_1 + H \Rightarrow$ for any r,q = 1..R holds:

$$x_r(j) - x_q(j) = x_r(j-W) - x_q(j-W) \qquad (7)$$

Using Equalities (6), one can easily show that if (7) holds for $j = j_1 + H$, then it also holds for any $j \geq j_1 + H$. Let H and W be the smallest integers satisfying (7). Then, starting from iteration $j_1 + H - W$, the graph behavior is always repeated W iterations later, establishing the periodic pattern, whereas the iterations before $j_1 + H - W$ exhibit behaviors that are never repeated later (transient behavior).

Our algorithm calculates $x_r(j)$ for j = [j_1, j_1+1, …], using Equality (7) as stopping criterion. The algorithmic cost is $O(R^2 \cdot H)$ for applying Equalities (6) plus $O(R \cdot H \cdot \log H)$ for searching for a match to Equality (7). In the end, the algorithm calculates the values of σ_s and λ_s, as explained below.

Observe that, instead of the simple periodic pattern presented in Section IV, where the behavior repeats every iteration, in the general case, it repeats every W iterations, where W depends on the graph [16 §7.4]. Therefore, we calculate period λ_s as the average timing distance between iterations: for arbitrary r ,

$$\lambda_s = \left(x_r(j_1 + H) - x_r(j_1 + H - W)\right)/W \qquad (8)$$

In line with our goal for latency σ_s, we calculate it such that expression $\lambda_s \cdot J + (\sigma_s - \lambda_s)$ is not less than $x_r(j_1 + J - 1) - T$ for any r and J, using the following equality:

$$\sigma_s = \max_{r=1..R} \max_{j=j_1..j_1+H-1} x_r(j) - \lambda_s \cdot (j - j_1) - T \qquad (9)$$

For our example, we calculated $x_r(j)$ in algebraic form, yielding the following results:

$$\lambda_{s\,(\text{example})} = \max(\eta_s + \varphi_s, \beta_s) \qquad (10)$$

$$\sigma_{s\,(\text{example})} = \eta_s + \varphi_s + 2\beta_s \qquad (11)$$

In general, to get λ_s and σ_s, one has to run the algorithm for each set of delay levels. The total algorithmic cost is $O(S \cdot (R^3 + R \cdot H_S \cdot \log H_S + R^2 \cdot H_S))$, where S is the number of scenarios and H_S is the maximum H over all scenarios. Note, that more efficient algorithms exist for period λ_s (see [8]), but we are not aware of more efficient algorithms for σ_s or for any good approximation thereof.

D. Characterizing the Scenario Transitions

Due to multiprocessor pipelining, the subsequent graph iterations overlap in time. Coefficients σ_s and λ_s take into account overlap inside scenario intervals, but not the $\gamma_{s,t}$ overlap at scenario transitions (see Figure 1). This subsection is dedicated to the conservative calculation of $\gamma_{s,t}$.

Consider two consecutive intervals, $\mathbf{I}_{p-1}$ and $\mathbf{I}_p$, running in scenarios s and t respectively. In the previous subsection, we assumed that at the start of interval $\mathbf{I}_p$ all token slots get an initial token simultaneously, which is, in reality, not necessarily true. In this subsection, we consider a special moment in time T, such that $\mathbf{I}_p$ can be *conservatively* assumed to start at time T via the simultaneous arrival of all initial tokens at that time. Based on that time moment, we calculate the timing overlap $\gamma_{s,t}$ Note that

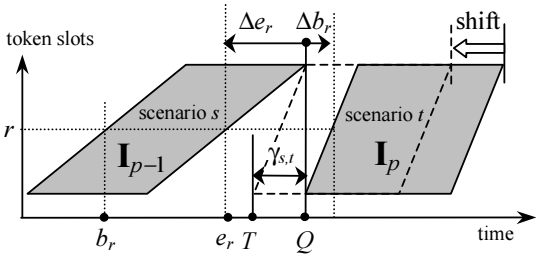

Figure 4. Finding the overlap between two time shapes

conservative assumptions about the arrival of initial tokens are allowed due to the monotonic timing behavior of HSDF graphs..

This idea is illustrated in Figure 4, where intervals $\mathbf{I}_{p-1}$ and $\mathbf{I}_p$ are represented by two time shapes, shown as parallelograms (for illustration purposes). The vertical axis corresponds to the set of R token slots, indexed by r. The figure ignores the fact that this axis is discrete. The horizontal axis corresponds to time. A horizontal section of a time shape represents a time interval between two events: event b_r, i.e. the consumption of a token at slot r at the beginning of the interval, and event e_r, i.e. the transfer of the token to slot r in the interval's last iteration.

Figure 4 shows how the time shapes would be arranged if there were no overlap between them and $\mathbf{I}_p$ were postponed until time Q, when $\mathbf{I}_{p-1}$ completes its execution. In such an arrangement, there is a gap between the time shapes. We reduce this gap by shifting $\mathbf{I}_p$ to the left from point Q to point T, where the shapes touch each other. The shifting distance (Q–T) is in fact the overlap value $\gamma_{s,t}$.

We calculate $\gamma_{s,t}$ based on the time shapes. Suppose that shape $\mathbf{I}_p$ is again located at starting point Q. For token slot r, let Δe_r be the distance between the right border of $\mathbf{I}_{p-1}$ and the reference point Q (see Figure 4). Let Δb_r be the distance between Q and $\mathbf{I}_p$'s left border. From Figure 4, it is obvious that time shape $\mathbf{I}_p$ can be shifted to the left by at most:

$$\gamma_{s,t} = \min_r(\Delta e_r + \Delta b_r) \qquad (12)$$

In the rest of this subsection, we focus on the calculation of Δe_r and Δb_r. We first build an HSDF graph, the *transition graph* $\mathbf{G}_{\mathbf{trans}}$, whose nodes model the actor executions of IPC graph $\mathbf{G}$ in the neighborhood of the scenario transition. Let M be the maximum number of initial tokens on any edge of IPC graph $\mathbf{G}$. The transition graph is obtained from $\mathbf{G}$ by unfolding it with a factor of $2M$. Each actor v_k in $\mathbf{G}$ is represented by $2M$ actors in $\mathbf{G}_{\mathbf{trans}}$: $v_k[1]$, $v_k[2]$, … $v_k[2M]$. There is an edge in $\mathbf{G}_{\mathbf{trans}}$ from $v_x[f]$ to $v_y[f+m]$ if and only if $\mathbf{G}$ contains an edge (v_x, v_y) with m initial tokens. The result for our example (where $M = 2$) is given in Figure 5.

Let j_1 be the first iteration of interval $\mathbf{I}_p$. Actors $v_k[1]$, $v_k[2]$, … $v_k[2M]$ in fact model the IPC actor executions in the range from $(j_1 - M)$ to $(j_1 + M - 1)$. When assigning the delay values to the transition graph actors, we take into account that iteration $(j_1 - 1)$ is in scenario s and iteration j_1 is in scenario t, because the depths of scenario intervals $\mathbf{I}_{p-1}$ and $\mathbf{I}_p$ are at least 1. It is not known which scenarios are active more than one iteration away from the transition. Therefore, to obtain conservative (small enough) Δe_r and Δb_r, for those iterations,

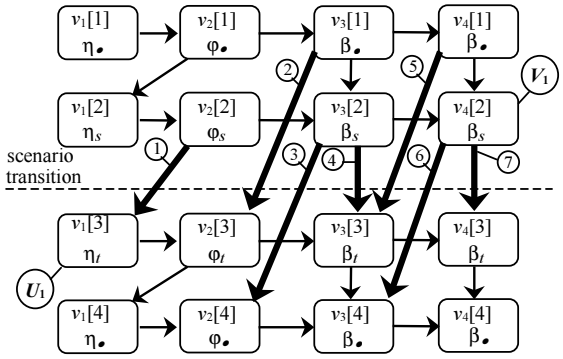

Figure 5. The transition graph for the case study

we use minimal delays. Using notation 'd_{trans}' for the delays of the transition graph actors, we have:

$$d_{\text{trans}}(v_k[M]) = d_s(v_k) \qquad (13.1)$$

$$d_{\text{trans}}(v_k[M+1]) = d_t(v_k) \qquad (13.2)$$

for $f < M$ or $f > M+1$,

$$d_{\text{trans}}(v_k[f]) = \min_{\text{scenario } z}\{d_z(v_k)\} \qquad (13.3)$$

For our example in Figure 5, we use notations $\eta_\bullet, \varphi_\bullet$, and $\beta_\bullet$ respectively for the minimal values of η_s, φ_s, and β_s.

Partition the transition graph at the scenario transition into an upper and a lower part. We observe that there is a one-to-one correspondence between each edge crossing the partition boundary and an initial token r. We call such edges the *edges of interest*; we show them in Figure 5 using bold arrows and we index them with index r as well.

Δe_r and Δb_r can now be calculated using the transition graph. Δb_r is the 'as soon as possible' (*asap*) time when the consumer node of edge r is ready to consume a token. The *asap* time is relative to the time when the lower part of the graph starts its first actor execution. To calculate this *asap* time, we find the nodes in the lower part of the graph that are the first to start, referring to them as the *sources of interest* U_i. A source of interest is recognized by the property that it has solely edges of interest as incoming edges. In Figure 5, the only source of interest is $U_1 = v_1[3]$. The *asap* time of a (consumer) node $v_k[f]$ in the lower part of the transition graph is equal to the largest delay of a graph path from any source of interest to node $v_k[f]$, not including the delay of that node. For example, the *asap* time of node $v_3[4]$ is $\eta_t + \varphi_t + \max(\eta_\bullet + \varphi_\bullet, \beta_t)$, and it is equal to Δb_6, because $v_3[4]$ is the consumer of edge 6. The right boundary Δe_r can be calculated via the same line of reasoning, except that we look at the upper part of the graph, we compute the 'as late as possible' (*alap*) relative times, we use *sinks of interest* V_i, which have solely edges of interest as outgoing edges, and the paths propagate from the producer node of the edge of interest (not including its delay) to a sink of interest. In our example, node $V_1 = v_4[2]$ is the only sink of interest and, for example, Δe_2 is the *alap* time of $v_3[1]$, which is $2\beta_s$.

To calculate *asap* and *alap* times, a longest path algorithm can be used, and the algorithmic cost is $O(M\cdot(V+E))$ per scenario, where V and E are the numbers of IPC graph actors and edges, and M the maximum number of initial tokens on any edge. The cost of Equality (12) is $O(S^2\cdot R)$. Therefore, the total algorithmic cost to calculate the overlap values for all scenario transitions is $O(S\cdot M\cdot(V+E) + S^2\cdot R)$.

For our example, we obtained an algebraic expression:

$$\gamma_{s,t} = \min(2\beta_s, \eta_t + \varphi_t + \min(\beta_s, \beta_t)) \quad (14)$$

E. Algorithmic Complexity

The total algorithmic cost of the graph-level analysis is the sum of the total costs of the algorithms presented in the previous two subsections. Among all variables contributing to cost, the only concern is H_S because it has a value that is in the worst-case exponential in the representation of the IPC graph. In our case study, we saw that H_S could change by a factor of 2 due to a 0.00001% change in a scenario delay level, which could be explained by the fact that multiple bits are required to represent that change accurately, H_S being worst-case exponential in that number. Nevertheless, when we represent the coefficients with a reasonable accuracy of 0.1%, in our case study and our previous work on static-delay HSDF graphs, we have never seen the value of H_S to exceed 10. To improve the robustness of our method, finding approximations of σ_s with polynomial algorithmic cost is an important future work topic.

VI. Case Study

A. Implementation and Actor-level Analysis

In our case study, we mapped an MPEG-4 decoding application for an arbitrarily-shaped video object [6] onto two ARM7 processors [1], P_1 and P_2, running at 100MHz. P_1 served as the main compute engine. P_2 served as a memory controller for computing the addresses and accessing the large video memory containing the shape data of the video object.

The IPC graph in Figure 2 models the mapping and the scheduling of this application. It is essentially a pipeline reading data in actor v_1 and storing the processing results in v_4. The tokens are 16x16 pixel video blocks. Actor v_1 models the video block entropy decoding on P_1 and reading the video memory for the reference video block, done by P_2. These operations are enwrapped into one actor because the application does not allow executing them in parallel. (Observe that they could not be mapped onto the same processor due to the constraint that P_2 is the memory controller.) Actor v_2 sends the decoding results through a network channel to processor P_2. Actor v_3 models the network channel and actor v_4 models storing the output in the video memory, executed by P_2 in a separate thread of execution. The edges between v_2, v_3, and v_4 model two FIFO buffers with space for two tokens each.

The computation and communication budgets were tuned such that both processors were fully loaded (in the worst case) and the load was properly balanced. For example, balancing the budgets for actors v_3 and v_4 leads to the same delay β_s for both.

The actor-level analysis considered only actor v_1; the other actors have constant delays which could be simply measured. We split actor v_1 into several subroutines. Different subroutines had up to 8 actor-level scenario parameters. We applied linear regression to the subroutines separately. This yielded 17 actor-level parameters and 18 coefficients in total.

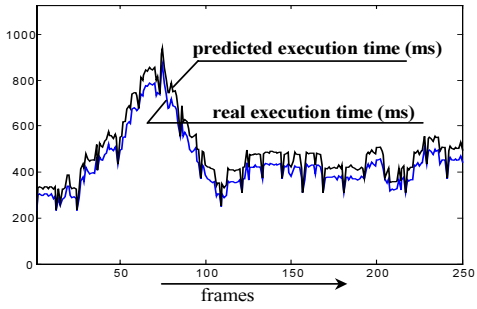

Figure 6. Execution time prediction results for Stream 1

To define scenarios, we identified 8 major video block types defined in the decoding algorithm, considering each type as a scenario candidate as it showed similar decoding delays for different blocks and corresponded to a certain combination of values of the most influential actor-level parameters. The candidates with similar contribution to execution time were merged, yielding three scenarios (see Figure 3).

To illustrate the usage of prediction, we implemented a simple *frame-skipping quality manager* that estimates whether the current frame or any frame depending on it can meet its deadline. If the answer is negative, then the manager skips the frame and the frames depending on it, and the decoding continues with the next frame that is estimated to meet its deadline.

B. Graph-level Analysis Results

The major goals of our graph-level analysis evaluation are:

- to measure the accuracy and to check conservatism;
- to explore the overhead-accuracy trade-off;
- to study the importance of accuracy for visual quality;
- to compare our results to worst-case analysis results.

We ran our execution time prediction method for two available sample video streams, containing 250 video frames each (with 45-255 blocks per frame). To calculate the scenario coefficients, we used expressions (10), (11) and (14), with the set of values $\mathbf{D}(v_1) = \{\eta_s\}$ being updated for every frame using Equality (4), values $\hat{\xi}_\omega(s)$ for that equality being obtained from the frame headers and φ_s and β_s being constant. The results are summarized in Table I. The columns show per stream the results for ideal prediction and the prediction using a certain number of scenarios. The first two lines show the average and maximum error with respect to simulations (ran on a multiprocessor extension of the Armulator simulator [1]). For our default setting (3 scenarios), our method yields 11% and 10% average error for the two sample streams. Figure 6 shows execution time curves for Stream 1. Our prediction turned out to be strictly conservative (although this is theoretically not guaranteed), which is in line with our objectives. The measured prediction error can be mostly explained by overprediction due to our scenario-based actor delay model (Figure 3).

TABLE I. Results for MPEG-4 Shape Decoding

	Stream 1				Stream 2			
scenario count	***ideal***	**3**	**2**	**1**	***ideal***	**3**	**2**	**1**
avg error, %	0	11	25	56	0	10	18	60
max error, %	0	17	36	77	0	14	24	89
quality, %	77	64	46	28	77	70	65	33
overhead (bytes)	-	20	12	5	-	19	13	6
overhead, %	-	5	3	1.3	-	1	0.7	0.3

The more scenarios are used in the prediction, the larger the overhead, because the more actor-level and graph-level parameter values need to be encoded in the frame header. We measure the overhead needed to encode the difference between the parameter value in the current and the previous frame using Shannon's entropy metric. The results are shown in the last two rows of Table I. The relative overhead differs considerably between the streams because they have different average frame size, 400 bytes and 2000 bytes. The absolute overhead is almost independent of the frame size. The results show that the relative overhead is limited if frame sizes are not too small. Note that overhead can be further reduced, e.g. by applying quantization to the least sensitive parameters.

We see that using less scenarios reduces the overhead but leads to poor accuracy. As we see in Table I, reducing the number of scenarios from 3 to 2 (by merging the two highest delay levels in Figure 3) results roughly in twice the error. Having only 1 scenario leads to even larger error increase (by a factor of 5 to 6). The one-scenario approach is in fact similar to using worst-case throughput [8], which shows the big advantage of our scenario-based approach over that technique.

Differences in prediction accuracy have a big impact on the visual quality, because more frames are skipped if the overprediction grows. For Stream 1, we set the frame deadline to 400ms, which produces significant processor overload. For Stream 2, we set the deadline such that similar overload was achieved. This choice is intentional, showing a situation where the quality manager has to actively control the quality. For our frame-skipping quality manager, Table I shows the quality results, measured in the percentage of frames presented to the user. From the table, for Stream 1, we observe a significant quality drop, from 64% for our approach to only 28% for the worst-case approach. A similar observation holds for Stream 2.

Note that the frame-skipping is not typical for advanced video decoders; neither is the frame rate of 2.5 frames per second (i.e. a 400 ms deadline). We made those assumptions due to practical limitations in the experimental setup. The results and conclusions carry over to realistic settings with faster processors and more advanced quality control methods.

VII. CONCLUSIONS

In this paper, we have introduced an execution time prediction method with adequate support for task-level parallelism in multiprocessors. Our method is oriented to multimedia streaming applications. It yields accurate and conservative resource utilization predictions needed for run-time resource and quality management in low-power embedded multimedia systems. To produce conservative results, we base our method on analytical techniques. We could not apply existing performance analysis techniques directly as they can only accurately analyze the behavior when performance metrics are static. For truly data-dependent dynamic systems, the most that these techniques can provide is worst-case performance estimates, which is in general insufficiently accurate for the intended application domain.

Our approach introduces support for dynamic data-dependent behavior into IPC (inter-processor communication) graphs [2], which constitute an important performance analysis framework for streaming applications running on MP-SoCs. IPC graphs belong to the class of HSDF graphs [12]. Our techniques are not limited to IPC graphs, but apply to arbitrary HSDF graphs with data-dependent actor execution delays. The key to our approach is the extension of the analysis to multiple scenarios, where a scenario captures runtime behaviors of an application with similar resource usage [10, 21]. For that, we developed novel analytical techniques for characterizing transient behavior and scenario transitions in HSDF graphs.

Application of our approach to an MPEG-4 shape decoder shows that our method can give good accuracy, at acceptable costs, where a standard worst-case analysis approach would fail to provide a good prediction.

In future work, we would like to make the transient behavior analysis robust against possible occasional high algorithmic complexity costs, and we plan to apply our results in MP-SoC quality and resource management.

REFERENCES

[1] www.arm.com

[2] N. Bambha, V. Kianzad, M. Khandelia, S.S. Bhattacharrya, Intermediate Representations for Design Automation of Multiprocessor DSP Systems. In Design Automation for Embedded Systems, vol. 7, 307-323, , 2002.

[3] A.C. Bavier, A.B. Montz, L.L. Peterson, Predicting MPEG Execution Times, Proc. of ACM SIGMETRICS'98, pp. 131-140, 1998.

[4] M. Bekooij, et al, Predictable Embedded Multiprocessor System Design, In Lecture Notes in Computer Science 3199, pp. 77-91, Springer 2004.

[5] S. Chatterjee, A.S. Hadi. Influential Observations, High Leverage Points, and Outliers in Linear Regression. Statistical Science, 1986. pp. 379- 416.

[6] N. Brady, MPEG-4 Standardized Methods for the Compression of Arbitrarily Shaped Video Objects, IEEE Transactions on Circuits and Systems for Video Technology, vol. 9, no. 8, pp. 1170-1189, 1999.

[7] Cradle Technologies, Inc., Multiprocessor DSPs: Next Stage in the Evolution of Media Processing DSPs, white paper, www.cradle.com

[8] A. Dasdan, R.K. Gupta. Faster Maximum and Minimum Cycle Algorithms for System-Performance Analysis, IEEE Trans. on CAD of Integrated Circuits and Systems, 17(10): 889-899, 1998.

[9] S.V. Gheorghita, T. Basten, H. Corporaal. Profiling Driven Scenario Detection and Prediction for Multimedia Applications. In Proc. IC-SAMOS 2006, pp. 63-70. IEEE CS Press, 2006

[10] S.V. Gheorghita, T. Basten, H. Corporaal. Application Scenarios in Streaming-Oriented Embedded System Design. In Proc. SoC 2006, pp. 175-178. IEEE, 2006.

[11] R. Lauwereins, M. Engels, M. Ade, J.A. Peperstraete, Grape-II: A System-Level Prototyping Environment for DSP Applications. In IEEE Computer, vol. 28, no. 2, 35-43, Feb. 1995.

[12] E.A. Lee, and D.G. Messerschmitt, Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. In IEEE Transactions on Computers, vol. 36, no. 1, pp. 24-35, 1987.

[13] Z. Ma, C. Wong, P. Yang, J. Vounckx, F. Catthoor. Mapping MPEG-4 visual texture decoder efficiently on a heterogeneous multi-processor platform. IEEE Signal Processing Magazine. 22:(3) 65-74, 2005.

[14] P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, B. Mesman, Task-level Timing Models for Guaranteed Performance in Multiprocessor Networks-on-Chip, Proc. CASES'03, pp. 63-72. ACM 2003.

[15] K. Richter, M. Jersak, R. Ernst, A Formal Approach to MP-SoC Performance Verification, IEEE Computer, 36(4): 60-67, 2003.

[16] S. Sriram, and S.S. Bhattacharyya, Embedded Multiprocessors: Scheduling and Synchronization, Marcel Dekker, Inc., 2002.

[17] M.T.J. Strik, A.H. Timmer, J.L. van Meerbergen, G.-J. van Rootselaar, Heterogeneous Multiprocessor for the Management of Real-time Video and Graphics Streams. In IEEE Journal of Solid-State Circuits, vol. 35, no. 11, 1722 –1731, 2000.

[18] S. Stuijk, T. Basten, M.C.W. Geilen, H. Corporaal. Multiprocessor Resource Allocation for Throughput-Constrained Synchronous Dataflow Graphs. Proc. DAC 2007, pp. 777-782, ACM 2007.

[19] B.D. Theelen, Performance Modeling for System-Level Design PhD. Thesis. Eindhoven Univ. of Technology, Eindhoven, the Netherlands.

[20] A. Varma, et al, A Control-Theoretic Approach to Dynamic Voltage Scheduling, Proc. CASES'03, pp. 255-266. ACM 2003.

[21] P. Yang, et al, Managing Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems, Proc. ISSS'02, pp. 112-119, ACM 2002.