technische universiteit eindhoven

Run-time Prediction of Execution Times of Stream-oriented Applications in Multiprocessors On-chip

P. Poplavko, M. Pastrnak, T. Basten, J. van Meerbergen, M. Bekooij, P. de With

ES Reports
ISSN 1574-9517

ESR-2005-06
8 July 2005

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems

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

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

Run-time Prediction of Execution Times of Stream-oriented Applications in Multiprocessors On-chip

P. Poplavko[1,2], M.Pastrnak[1,3], T. Basten[1], J. van Meerbergen[1,2], M.Bekooij[2], and P. de With[1,3]
[1]Eindhoven University of Technology, [2]Philips Research, [3]LogicaCMG Nederland
E-mail: p.poplavko@tue.nl

Abstract

For stream-oriented applications, it is a challenging problem to predict the total execution time of a loop consisting of multiple tasks with data-dependent task execution delays executed in a pipeline-like manner on a multiprocessor system on-chip. Embedded applications can profit from such prediction at run-time, e.g. for power and quality-of-service management. For this purpose, we propose a generic loop execution time estimate giving a tight upper bound and taking parallelism into account. Our estimate is an algebraic expression of a few a priori parameters describing the frequency and the value of changes of the task execution delays. Our method is based on a timing analysis of the loop. To illustrate how our method can be applied in practice, we use an MPEG-4 algorithm for decoding video object shape as a case study.

1. Introduction

Our work focuses on stream-oriented applications with real-time constraints, e.g. video/audio coding, for on-chip multiprocessors. To cope with the complexity of the system design we require timing predictability from the multiprocessor platform. Given a predictable hardware architecture and predictable real-time scheduling, we develop methods to predict the overall system-on-chip performance, characterized by the *execution time*, i.e. the time it takes to process a given number of data items. The execution time is inversely proportional to the throughput.

In general, an execution time is data-dependent. If the execution time exceeds the deadline specified in the real-time constraints, a preventive action should be taken to correct the situation. For example, voltage/frequency can be increased, e.g. [11], or the quality of service level can be reduced, e.g. [2]. Many run-time management methods require the execution time to be predicted in advance so that they can take the appropriate action on time.

Existing prediction methods estimate execution times using an algebraic expression on the parameters of the input data [1] known *a priori*. However, they assume sequential execution, whereas, for stream-oriented applications, pipeline-like execution on several processors is very important for achieving the required throughput. Furthermore most existing methods do not provide any guarantees on meeting the deadlines.

To support guarantees, the existing literature offers schedulability analysis for embedded multiprocessors, e.g. [9]. However, such methods only perform the worst-case analysis, which is not enough for run-time management.

In this paper, we propose a run-time execution time estimate that supports pipeline-like execution and performance guarantees. We consider a generic loop – we call it the *loop of*

interest – that has to perform a specific number of *iterations* within a certain deadline. This set of iterations is called the *loop execution*. The total time to complete the loop execution is called the *loop execution time*. The body of the loop consists of tasks with data-dependent execution delays.

We propose an algebraic expression that yields an estimate of the loop execution time. The estimate is *conservative*, i.e. it gives a provable upper bound. Our method is based on formal timing analysis and the timing model we introduced in [6, 7] and the scenario-based approach we proposed in [12]. However, so far the accuracy of our method could be severely compromised by dynamic data-dependent changes in task execution delays. The main contribution of this paper is an *overlap analysis technique*, which effectively and conservatively predicts the effect of those changes on the execution time of the pipeline. It makes our approach applicable, in terms of accuracy, to a broader class of data-dependent applications. It also enables a trade-off between prediction accuracy and run-time overhead of the prediction.

We start the paper by providing a motivating example, which clarifies our assumptions about the problem that needs to be solved and the appropriate means to solve it. Afterwards, we give an overview of our method and of the target abstract multiprocessor platform. Section 3 describes the timing models used in the analysis. In Section 4, we study the analysis problem and introduce the overlap analysis technique. In Section 5, we do a case study and analyze the results in terms of accuracy. Section 6 summarizes the contributions of this paper.

2. Problem context

2.1 Motivating example

A user application may involve multiple video objects, like cars, people, etc. Therefore, the MPEG-4 video standard introduced arbitrarily shaped video objects. Each video object is defined by an external source, providing the encoded data for the object. For each object, a video decoder is activated. It can be split into the binary shape decoding part and texture decoding part. The decoder produces a sequence of video object planes (VOPs), which correspond to video frames. Each VOP is an HxW matrix of J blocks (see Figure 1(a)). The blocks produced by the shape decoder contain binary pixel values and are called BABs (binary alpha blocks). The decoding of the complete shape information of a VOP is done by the block-level loop, block-by-block.

Video standards define the *tasks* that may be executed for each block. Examples are variable length decoding, motion vector decoding, etc. Each task reads the block information, processes it and passes the output to other tasks.

Different types of blocks are defined in video standards; for example, MPEG-4 defines transparent, boundary and opaque BABs (Figure 1(a)). The time it takes a task to process a block may strongly depend on the block type. For example, transparent BABs require the least time to

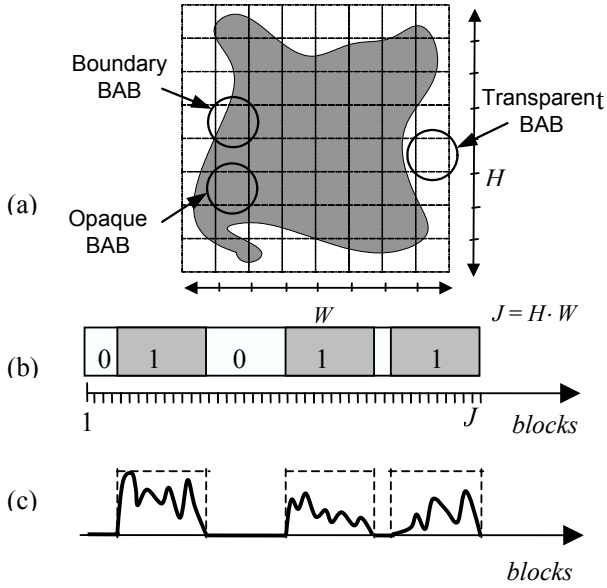

Figure 1: Video object shape decoding
(a) VOP binary shape and block types
(b) Block types: 0/1
(c) Task processing times and the block types

be processed. Figure 1(b) shows an example where the decoder sees consecutive blocks of two different types and Figure 1(c) shows a possible evolution of the processing times of a task.

VOPs have periodic decoding deadlines. For example, it may be required to produce one VOP every 30 ms. Therefore, we are interested in the estimation of the VOP decoding times. Our methodology should perform a sort of integration operation over the loop execution interval 1..*J* applied to the processing times of all tasks, e.g. the function in Figure 1(c).

The strong influence of the block types on task delays is important in execution time prediction. Bavier et al [1] studied the run-time prediction of decoding times of MPEG-2 video frames on a single processor. Their main result is that satisfactory predictions are only obtained when taking into account 3 different block types (I, P, and B). They introduced 3 *parameters*, namely, the number of blocks of type 'I', 'P' and 'B' in a frame. The execution time was evaluated as a linear expression on the parameters. The coefficients of the expression were computed at run time. The parameter values were known *a priori*, being included in the *header* of the video packet.

2.2 Problem, approach and related work

This section outlines our approach and compares it to related work.

In our method, the application designer provides the specification of the loop of interest in the form of a *process network* [5], consisting of processes, communicating through FIFO channels. Multiple concurrent processes may run on the same processor or on different processors. Each

process executes a fixed subset of tasks in a local loop on a given processor. The problem is then to conservatively estimate the overall execution time of the loop of interest.

Similarly to the work of Bavier et al [1], we require that the headers of the input stream contain a few parameters, characterizing the complexity of the loop execution a priori. In video applications, the video encoder can easily produce the values of such parameters. The major contributions of our method are that it:

- provides a conservative estimate,
- supports parallel execution of tasks,
- allows trade-off between accuracy and run-time overhead.

Our approach works as follows. First, we derive a timing model of the loop of interest [6, 7] based on homogeneous synchronous dataflow graphs (HSDF), although a bit restrictive, but often used to model multiprocessor applications. Then we construct an algebraic expression that estimates the loop execution time. It is obtained from the timing model using a new analysis technique that makes our approach better applicable for data-dependent applications.

In an HSDF, the tasks are modeled as dataflow *actors*. Timing analysis techniques have been proposed in the past employing similar timing models, e.g. for asynchronous circuits [13]. However, such techniques apply only for the static data-independent actor delay annotations (worst-case), which may result in overly pessimistic execution time estimations [7]. The analysis techniques working with probabilistic actor delays normally use assumptions that are not valid for stream-oriented applications [8, §7]. Therefore, we consider an important alternative, which can solve this challenging problem in many cases.

2.3 Architecture and scheduling

The multiprocessor architecture assumed in our methodology consists of *processing tiles* and an *interconnection network*. A processing tile is a self-contained embedded computer, with one processor and a local memory system. Each processor has a local scheduler managing multiple processes on a single processor. A process runs a sequence of tasks in a fixed order. For communication, tasks access first-in-first-out (FIFO) buffers in the local memories.

The *processing time* of a task is defined as the total time the processor spends on running that task only, not including the blocking times on synchronization. We require that the processing times of tasks be ‘predictable’, meaning that it can be accurately computed given that the task’s input data is known a priori. Therefore we assume that real-time tasks use no caching and all their data and instruction code are prefetched into the local memory in advance.

We adopt a time-division multiple access (TDMA) scheduling for processes, where time is divided into periods of length T, and each period is split into several time slots of possibly different size assigned to different processes.

The *response time* takes task preemption into account [4]:

$$R(t, T_B, T) = t + \left\lceil \frac{t}{T_B} \right\rceil \cdot (T - T_B) \quad (1)$$

where t is the task processing time, T_B is the time slot reserved for the corresponding process; the ratio T_B / T determines the processor cycle budget. The 'ceiling' part of the expression gives the worst-case number of preemptions, and $T - T_B$ gives the preemption delay.

We also require predictable timing in communication. The interconnection network should provide channels with bandwidth reservation. Further, we assume that channels are unidirectional and that they have at each side a FIFO buffer. A comprehensive example of such a network is the ÆTHEREAL network-on-chip [10]. By analogy to processes, which execute tasks, channels execute *data transfers*. A data transfer is also characterized by a response time. We omit details on the ÆTHEREAL network-on-chip because they are not relevant for the rest of the paper.

3. IPC model

3.1 Case study

In this section, from the process network of a loop of interest we construct a timing model, called an *IPC* (interprocessor communication) model, which is a conservative model of the loop's timing behavior. For illustration purposes, we use the shape decoder case study.

The process network of the shape decoder contains three processes assigned to two processors. Process 'Main' performs most of the computations for shape decoding on a CPU. The other two processes, 'Load' and 'Store' execute both on a memory controller

The tasks executed by the process are all represented in the IPC graph, so we describe them briefly here. Task '`Ini`' is for initialization; task '`DecMV`' decodes the coordinates – a motion vector ('MV') – of the BAB used as a reference for decoding. Task '`Load`' fetches the reference BAB from the video memory; task '`DecBAB`' finishes the decoding of the block and task '`WrBAB`' sends the decoded BAB to process 'Store', which stores the block in the video memory.

3.2 IPC graph

An IPC model consists of an IPC graph [7, 8], a processing time model and budget relations.

An *IPC graph* is an HSDF graph. It can be directly derived from a process network where the source code contains explicit annotations of the start of each task and the size of each data transfer. Each process should be annotated with a processor cycle budget, and each channel should be annotated with the reserved bandwidth and FIFO buffer sizes. The IPC graph of the shape decoder is presented in Figure 2.

The nodes of the graph are called *actors* A_k. We denote the actor set $\mathbf{A}$, and it has two subsets: $\mathbf{A_T}$ for the task actors, shown with white nodes in Figure 2, and $\mathbf{A_D}$ for the data transfer actors,

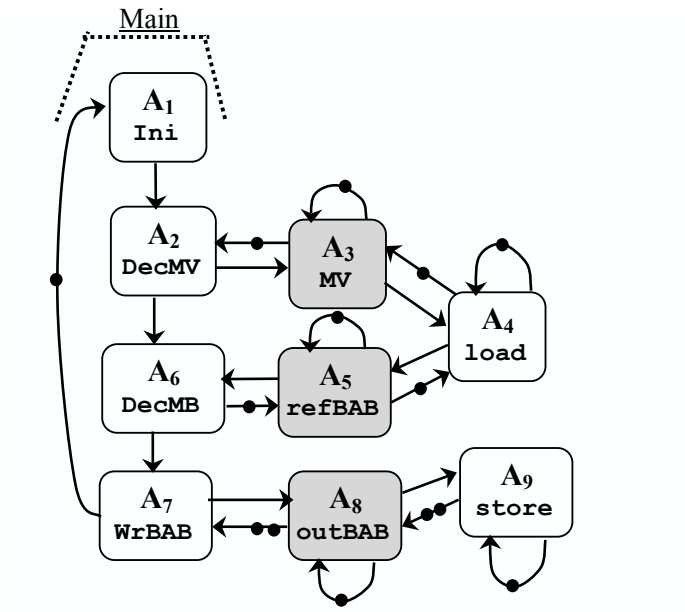

Figure 2: IPC graph of the shape decoder at BAB

shown in grey. The *edges* of the graph carry tokens from one actor to another in the FIFO order. Each edge may contain one or a few *initial tokens*.

The execution of an IPC graph is a series of iterations. All actors execute exactly once per iteration, and some may start a new iteration while the others are still busy with previous iterations. An actor starts an execution immediately when it sees one token at each input, and we say that the actor *captures* one token at each input. After an execution is completed, one token is *consumed* at each input and one token is *produced* at each output. The duration of execution is called *actor execution delay*, denoted $d(A_k, j)$, where integer j is the iteration index. Because the actor execution delay is data-dependent, it may change every iteration.

To construct an IPC graph, for each process, a cycle is created which consists of actors that model the tasks belonging to the given process. For example, in Figure 2 we highlight the cycle of process 'Main'. The information carried by a token that goes around such a cycle represents the local state of the process; it is not physically transported but kept in the local memory. Similarly, each channel is represented by a cycle of data transfer actors.

The FIFO buffers of a channel are modeled with edges between the tasks and the data transfers. Each FIFO buffer is modeled with forward edge(s), e.g. (A_2, A_3), and backward edge(s), e.g. (A_3, A_2). The forward edge carries the data stored in the buffer. The backward edge has the same number of initial tokens as the number of data tokens that can fit in the buffer. For example, both buffers of channel 'outBAB' can only accommodate up to two BABs.

General information about the construction of an IPC graph can be found in [7].

3.3 Actor execution delays

Throughout the paper, for different purposes, we define and apply different timing models for the internal delays of the actors, whereby we only consider the *task actors*. The details of modeling the data transfer delays are outside the scope of this paper.

To define the delays of the task actors, we first define a model for task processing time. The basic model is called the *parametric function,* which can be expressed as follows:

$$t_{par}(A_k, j) = c_{0,k} + c_{1,k}\xi_{1,k}(j) + \ldots + c_{P(k),k}\xi_{P(k),k}(j) \qquad A_k \in \mathbf{A_T},\ \xi_{p,k} \in \mathbf{\Omega} \tag{2}$$

where $c_{p,k}$ are constant coefficients, called *actor coefficients*, $\xi_{p,k}$ are *actor parameters*, characterizing the internal processing of actors, and $\mathbf{\Omega}$ denotes the set of all parameters. The actor coefficients, measured in clock cycles, should have values high enough to ensure conservative estimation. They provide us with an abstraction of the hardware architecture.

Example (actor parameters): We estimate the processing time of actor 'decMV' in the clock cycles of an ARM7 processor with the following linear expression:

$$175+792\,\xi_{\delta} + 580\,\xi_{\delta d} +464\,\xi_{\delta d+} + 128\cdot\xi_{\mathrm{Nbyte}}\ ,$$

where, for example, ξ_{δ} is a Boolean value equal to 1 if the shape decoder loads a reference BAB and ξ_{Nbyte} gives the number of byte boundaries crossed when accessing the input bitstream. ♦

We set the actor execution delays in IPC models to the values of their response times. This is expressed in the relation between the execution delay, the processing time and the processor cycle budget. Using (1), we obtain:

$$d(A_k, j) = R(t(A_k, j), T_B(A_k), T) \tag{3}$$

where $A_k \in \mathbf{A_T}$, t is a processing time model; T_B and T have been defined in Section 2.3.

4. Execution time prediction

A big part of this section is dedicated to timing analysis. First, we describe the scenario model and basic execution time estimate we proposed in [12]. Upon this foundation, we build a new estimate of the execution time and the overlap analysis technique, which supports it.

4.1 The challenge of timing analysis

The execution time of J iterations of a cyclic homogeneous synchronous dataflow graph can be computed directly as the length of the longest path through this graph, which traverses some actors and edges multiple times. Under *length* of the path we understand the sum of the delays of all actors that lie on it. To compute the execution time this way, one would have to know the actor delays in all iterations of the loop execution. For Equality (2), this means that one would have to know all parameter values for all actors in every iteration of the loop of interest. This

would introduce too much overhead both in terms of the amount of information stored in the input stream headers and in terms of the time required to compute the estimate.

The idea of the timing analysis is to estimate the execution time of the loop without having to directly compute it.

4.2 Scenario model

In the particular case where all actor delays are *static*, i.e. constant in all iterations, the longest path is known to converge to a periodically repeating sequence of actors, and thus we bound the execution time from above by a linear expression $\lambda \cdot (J-1) + \sigma$, where λ – *period* – and σ – *lateness* – can be derived from the IPC model [7].

However, the actor delays, modeled by Equalities (2) and (3), can change considerably at run-time. For example, Figure 3 shows the delay of the shape decoding for subsequent blocks of a video sequence. We see that it varies by almost a factor of seven.

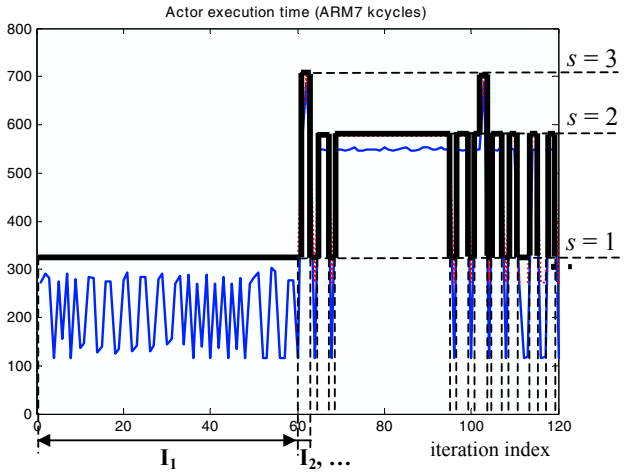

Figure 3: Scenario model for actor delay

In our approach we strive to reuse the results of the static analysis. To make it possible, we apply quantization to actor parameters $\xi_{p,k}$. The quantization levels are called scenarios. For each scenario s=1..S we specify the quantized values for all actor parameters: $\hat{\xi}_{p,k}(s)$. To obtain a conservative timing model, the rounding-up to the closest quantization level is used.

As a consequence of parameter quantization, the actor delay model also gets quantized and exhibits a more static behavior. We call such an actor delay model the *scenario model* $\hat{d}(s, A_k)$. For example, Figure 3 shows three scenarios we introduced for the shape-decoding case study, and the scenario model is shown with the bold curve.

Under the scenario model, the total interval of the loop iteration index 1…J can be split into a few *intervals* where actors have static delays. Thus, within such an interval, the IPC graph can be

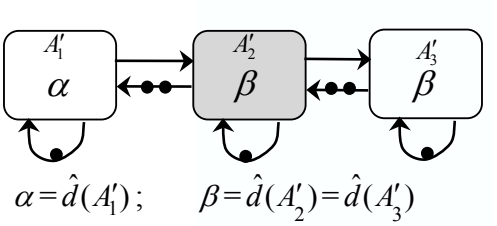

Figure 4(a) IPC graph

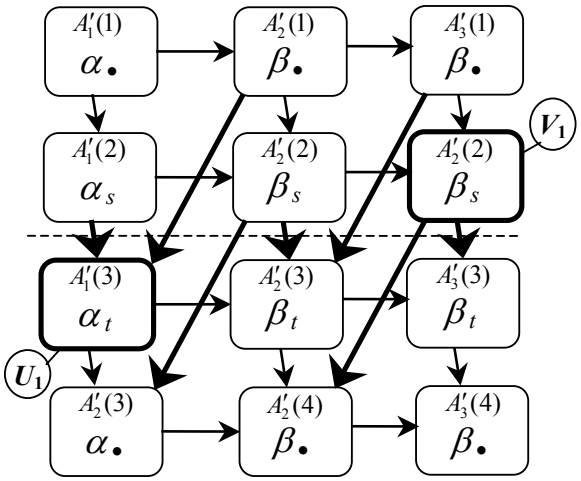

Figure 4(b) Unfolded graph and minimum overlap analysis

characterized by some λ and σ. Figure 3 shows on the horizontal axis the intervals corresponding to the given set of scenarios and the given video sequence.

In the consecutive iterations $j_1..j_2$ of each interval the parameters stay within the same scenario s, and we say that it *belongs* to scenario s. For example, in Figure 3, interval $\mathbf{I}_2$ belongs to scenario 3. At the interval boundaries, *scenario transitions* take place.

4.2 Basic execution time estimate

The scenario model enables us to express the loop execution time algebraically. In this subsection we give the basic execution time estimate of the scenario model.

In the algebraic expression of the estimate, we use parameters and coefficients that refer to the loop level of granularity. We call them *loop parameters* and *loop coefficients*. For the basic estimate, we introduce two loop parameters, namely, J_s giving the total number of loop iterations in scenario s and L_s giving the number of intervals belonging to scenario s.

By adding up the execution time estimates of all scenario intervals we obtain:

$$\hat{ET}_{basic} = \sum_s (\lambda_s \cdot J_s + (\sigma_s - \lambda_s) \cdot L_s) \qquad (4)$$

where λ_s and σ_s are the period and lateness of the IPC graph in scenario s.

The qauntization levels of the actor parameters $\hat{\xi}_{p,k}(s)$ have to be specified a priori in the input stream headers, as well as the values of the loop parameters. To obtain the loop coefficients – so far we classify λ_s and σ_s as such – analysis algorithms have to be applied to the IPC graph. For λ_s and σ_s a *general analysis rules* can be specified, applicable to any IPC graph, but we skip them for the space reasons. They can be found e.g. in our previous work [7]. Here we only specify the rules that work for a simple IPC graph in Figure 4(a).

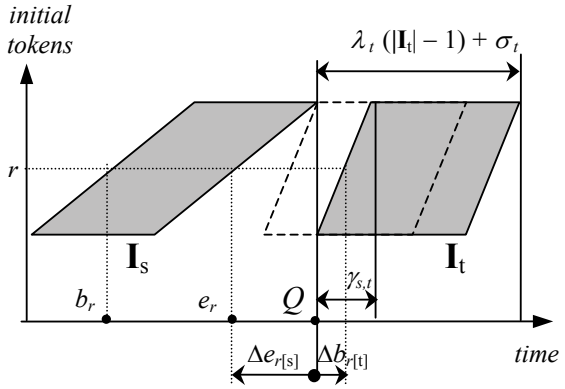

Figure 5: Minimum overlap of time

The period of the IPC graph in Figure 4(a) is determined by the actor that has the biggest delay. Thus we obtain:

Example Rule 1: $\lambda_s = \max(\alpha_s, \beta_s)$ ♦

The lateness of that graph is determined by the sum of actor delays:

Example Rule 2: $\sigma_s = \alpha_s + 2\beta_s$ ♦

4.4 Execution time estimate

The basic estimate does not take into account the time overlap between every two subsequent intervals at the scenario transitions. This can result in considerable loss of accuracy in case scenario transitions happen frequently. In this subsection, we describe a new execution time estimate, which takes the overlap into account.

Figure 5 shows an example of two iteration index intervals, $\mathbf{I}_s$ and $\mathbf{I}_t$, belonging to scenario s and t respectively. The intervals are represented by two *time shapes*, shown as filled parallelograms in Figure 5. The vertical axis corresponds to the set of all R initial tokens of the IPC graph, sorted in some order by index $r = 1 \ldots R$. For example, in Figure 4(a), $R = 7$. In fact, the vertical axis is discrete, although our drawings ignore this fact. The horizontal axis corresponds to time. A horizontal section of a time shape represents a time interval, whose *left border* and *right border* correspond to two events associated with the initial token r: the begin event b_r and the end event e_r. We define these events later.

The full horizontal length of the time shape $\mathbf{I}_s$ corresponds to its contribution to the total execution time estimate. It amounts to: $\lambda_s \cdot (|\mathbf{I}_s| - 1) + \sigma_s$, where $|\mathbf{I}_s|$ is the number of iterations in the iteration index interval, called the interval *depth*.

Figure 5 shows how the time shapes are arranged according to the assumptions of Equality (4). The main property of this arrangement is that we can distinguish a reference time

point Q where one can draw a vertical line through the latest e_r in interval $\mathbf{I}_s$ and the earliest b_r in interval $\mathbf{I}_t$. In such an arrangement, there is a gap between the time shapes.

We reduce the gap between the time shapes by shifting $\mathbf{I}_t$ to the left as far as we can such that the shapes do not cross each other. We denote the shift distance $\gamma_{s,t}$. The shifted position of the time shape $\mathbf{I}_t$ is shown in Figure 5 with dashed lines.

Let us compute the distance $\gamma_{s,t}$. For event r, let $\Delta e_{r[s]}$ be the position of $\mathbf{I}_s$'s right border relative to the reference point Q. Let $\Delta b_{r[t]}$ be the relative position of $\mathbf{I}_t$'s left border. From Figure 5 it is obvious that time shape $\mathbf{I}_t$ can be shifted to the left by at most:

$$\gamma_{s,t} = \min_r(\Delta e_{r[s]} + \Delta b_{r[t]}) \qquad (6)$$

In Section 4.5 we define the left and right borders such that they are conservative and not dependent on the position or depth of the intervals $\mathbf{I}_s$ and $\mathbf{I}_t$. Thus, it is valid to speak of $\gamma_{s,t}$ as the *minimum overlap* between scenario s and scenario t. We classify it as another loop coefficient. Thereby, we obtain a new execution time estimate:

$$\widehat{ET} = \widehat{ET}_{basic} - \sum_{s,t=1..S;s\neq t} \gamma_{s,t} \cdot K_{s,t} \qquad (7)$$

where $K_{s,t}$ is the number of transitions from scenario s to scenario t – loop parameter. We see that the overhead is quadratic in S, because the number of possible transitions is quadratic. We expect that in practice the overhead will be feasible because many scenario transitions will be impossible or unlikely to happen, enabling us to ignore them in Equality (7).

4.5 Analysis of minimum overlap

In this subsection, we work out the main idea already explained in the previous subsection and provide the general analysis rule for $\gamma_{s,t}$. To apply Equality (6), we have to define $\Delta e_{r[s]}$ and $\Delta b_{r[t]}$.

We do that in this subsection using an actor delay model d_{trans}, which characterizes the scenario transition. Suppose that j_1 is the last iteration of interval $\mathbf{I}_s$ and j_1+1 is the first iteration of interval $\mathbf{I}_t$. We call j_1 the transition point. Before the transition point, model d_{trans} considers only the actor executions that have direct influence on the starting times of actors after the transition point. After the transition point, model d_{trans} considers only the actor executions whose starting times can be directly influenced by actor executions before the transition point. Let M be the maximum number of initial tokens on any edge of the given IPC graph. We observe that for model d_{trans} it is enough to consider actor executions M iterations before the transition point and M iterations after. For convenience, we number the iterations of model d_{trans} starting from 1, therefore model d_{trans} spans the iteration index range $j = 1...2M$. With such a numbering, the scenario transition occurs between iterations M and M+1.

When filling the delay values into model d_{trans} we remember that we can only be sure that iteration *M* belongs to scenario *s* and iteration *M*+1 belongs to scenario *t`*, because the depths of intervals $\mathbf{I}_s$ and $\mathbf{I}_t$ are at least 1. It is not known which scenarios are active further than 1 iteration away from the transition point, so we have to fill in conservative values there. Remember that the final goal of model d_{trans} is computation of $\Delta e_{r[s]}$ and $\Delta b_{r[t]}$. For them, 'conservative' means 'small enough'. Therefore we put the minimal values from all scenarios. So, we have:

$$d_{trans}(A_k, j) = \hat{d}(\bullet, A_k), \quad j < M \text{ or } j > M+1 \quad (8.1)$$

where $\hat{d}(\bullet, A_k) = \min_{q=1..S}\left(\hat{d}(q, A_k)\right)$

$$d_{trans}(A_k, M) = \hat{d}(s, A_k) \quad (8.2)$$

$$d_{trans}(A_k, M+1) = \hat{d}(t, A_k) \quad (8.3)$$

Let us unfold the IPC graph $2M$ times and apply this model. Figure 4(b) shows the corresponding unfolded graph for the IPC graph in Figure 4(a), where $M = 2$. Each actor of the original graph is replicated $2M$ times. Each replica represents one actor execution of the original graph, and the edges between them represent the dependencies between actor executions. The *j*-th replica of actor A'_k gets notation $A'_k(j)$. Its execution delay is set to $d_{trans}(A'_k, j)$. The execution delays used in Figure 4(b) – $\alpha_\bullet, \alpha_s, \alpha_t, \beta_\bullet, \beta_s, \beta_t$ – comply with Equalities (8). Each edge of the original graph containing *m* initial tokens is replicated $2M - m$ times, the edge replicas joining the of the producer actors with index j and the consumer actors with index $j + m$.

We can partition this graph into two parts at the location of the scenario transition: upper part and 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 4(b) using the bold arrows and we implicitly index them with index *r* as well.

Now we are ready to define $\Delta e_{r[s]}$ and $\Delta b_{r[t]}$ using the unfolded graph. The left boundary $\Delta b_{r[t]}$ is defined as the 'as soon as possible' (*asap*) time between the moment when the lower part of the graph starts execution and the moment when it captures the token on the edge of interest *r*.

To describe the computation of *asap*, let us first find the nodes in the lower part of the graph that are the first to start, we call them the *sources of interest* U_i. A source of interest has solely edges of interest as incoming edges. In Figure 4(b), the only source of interest is $U_1 \equiv A'_1(3)$.

The *asap* time of a node $A_k(j)$ in the lower part of the graph is equal to the length of the longest path from any source of interest to node $A_k(j)$, not including the delay of that node. For example, the *asap* time of node $A'_2(4)$ is $\alpha_t + \max(\alpha_\bullet, \beta_t)$.

Finally, $\Delta b_{r[t]}$ is computed as the *asap* time of the consumer node of the edge of interest that corresponds to initial token *r*.

The right boundary $\Delta e_{r[s]}$ can be defined and computed by 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_l, which have solely edges of interest as outgoing edges, and the longest paths propagate in the reverse direction. For example, node $V_1 \equiv A_3'(2)$ is the only sink of interest in Figure 4(b) and the *alap* time of node $A_2'(1)$ is $2\beta_s$.

General Rule 3 (for γ): For each scenario, compute *asap* and *alap* values and then, for each scenario transition, compute $\gamma_{s,t}$ using Equality (6). ♦

This rule has algorithmic complexity quadratic in *S* and linear in *M*. This overhead can be reduced. For example, as mentioned before, it may be possible to ignore many scenario transitions.

After some algebraic manipulations we obtained the following analysis rule for our example:

Example Rule 3:

$\gamma_{s,t} = \min(2\beta_s, \alpha_t + \beta_s, \alpha_t + \max(\alpha_\bullet, \beta_t))$ ♦

5. Case study

In this section, we give an example of how our method can be applied in practice. We also provide experimental evidence of good accuracy due to the minimum overlap analysis. As a reference point to evaluate the accuracy we take the simulation results.

Our reference point itself has an important interpretation. In fact, we simulate an IPC model, using the real traces of processing time obtained for certain input. On the one hand, we admit that the IPC model introduces some error to ensure conservatism when compared to the real execution of the application in hardware. On the other hand, the simulation of the IPC model gives the best obtainable estimations of the guaranteed performance. Thus, we will evaluate the error of our algebraic expression in the estimation of the guaranteed system performance.

5.1 Experimental setup

We have developed a functional executable of an MPEG-4 shape decoder in C++. It implements the process network introduced in Section 3.1. We use two input video sequences 'singer' and 'dancers' from MoMuSys project. Both of them contain around 250 VOPs. We also use a trimmed version of 'singer' sequence (30 VOPs) as a *sample sequence* to derive the actor delay models as described in Section 5.3.

We have run the executable in a multiprocessor simulation environment, where the simulated processors are ARM7 RISC cores. The processors are modeled using an instruction-set simulator (ISS), integrated with a simulation backend based on SystemC. In our setup, a few ISS instances

can run separate processes. The simulation backend models the communication and manages the global simulation time model. The processor budgeting is modeled using Equality (3).

The simulation environment generates actor processing time traces, actor parameter traces and loop execution time traces. The former two are fed to the scripts that implement the execution time estimation for this case study and the latter is the reference point for accuracy evaluation.

5.2 Graph transformations

The number of actors and edges directly influence the algorithmic complexity of the analysis rules. It can be sometimes possible to transform the graph such that the complexity is reduced, although our method does not require that. For example, we simplify the graph in Figure 2 to obtain the graph in Figure 4(a).

First, we rename the nodes A_8 and A_9 into A'_2 and A'_3 respectively. Then, we observe that actors A_1, A_2 … A_6, always fire one after another and never overlap. Therefore, we can merge actors A_1, A_2…A_6 into one actor A_{10}, whereby the behavior of the rest of the graph is kept intact. The final step is to merge A_{10} and A_7 into actor A'_1. This step has only a small impact on the behavior, and it is conservative.

Our simplification yields extra analysis rules:

Example Rule 4: $\alpha_s = \sum_{k=1}^{7} \hat{d}(s, A_k)$ ♦

Example Rule 5: $\beta_s = \max(\hat{d}(s, A_8), \hat{d}(s, A_9))$ ♦

Thus, the delays α and β determine the performance of our case study application. This is manifested first of all in Example Rule 1, because the loop iteration period λ is a key performance factor.

5.3 Actor execution delays

As mentioned before, in order to define the execution delays of the task actors, we first construct parametric functions. For each actor, we define a set of actor parameters that contribute linearly to the processing times. The coefficients are obtained using the parameter and processing time traces of the sample sequence, using two alternative methods. First, if an actor exhibits a high variation and diversity of the processing times, we use the linear regression technique [6], with a correction for obtaining large enough values of coefficients. Second, sometimes it is justified to measure the maximum value of the processing time and to use it as the coefficient in a simple parametric function consisting of just one linear term.

Having defined the processing time models, we also have to choose the resource budgets for all the processes and, implicitly, the channels as well.

First, we predetermine that process ‘Main’ gets 100% budget of its processor, because it contains the heaviest actors. Consequently, for A_1, A_2, A_6, A_7, we have $\hat{d} = \hat{t}_{par}$.

The bandwidth reservation in the interconnection network gives us certain freedom to predetermine the delays of data transfers A_3 and A_5. We set the bandwidth such that these values

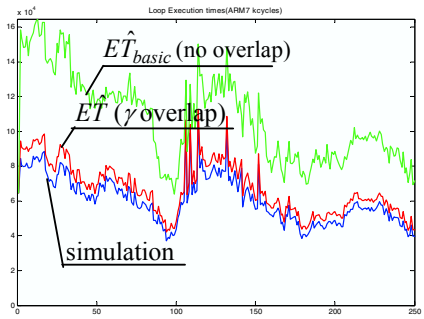

Figure 6: Loop-level execution

are at least one order of magnitude smaller than the other components of α in Example Rule 4. Therefore, these data transfers are sure not not cause a performance bottleneck.

Next, we consider actor A_4 'load', contributing to α, and actor A_9 'store', contributing to β. They run on the same processor. We have to split the processor cycle budget between A_4 and A_9. According to Example Rule 1, for good performance one has to balance α and β, and we set the budgets such that we obtain a reasonable balance.

Finally, we balance Example Rule 5 by adjusting the channel bandwidth such that the delay of the data transfer A_8 is roughly equal to the delay of task A_9.

5.4 Scenarios

Defining the scenarios for this case study determines the quantization levels of the delay of actor A_1', or α_s, whereas the other two actors have constant delays.

To define quantization levels of actor parameters, first we considered two of them having the highest impact on $d(A_1', j)$, namely, parameter ξ_δ, introduced in Section 3.2, and ξ_μ, which is a Boolean equal to 1 only if so-called 'context arithmetic decoding' is involved in the decoding of the BAB. We define each scenario by the value taken by one or both of those parameters. All the other parameter values are rounded up to their maximum observed in the given scenario.

Example Scenario 1: $\xi_\delta = 0$ ♦

Example Scenario 2: $\xi_\delta = 1$, $\xi_\mu = 0$. ♦

Example Scenario 3: $\xi_\delta = 1$. $\xi_\mu = 1$. ♦

In fact, it is $d(A_1', j)$ and its scenario model that is shown in Figure 3.

5.5 Accuracy of our method

Using the scenarios and rules defined above, we have applied our method for predicting the VOP decoding times for video sequences 'singer' and 'dancers'. Figure 6 shows the prediction

and simulation results for VOP decoding times for ‘dancers’. To verify that the computation of overlap is necessary, we show both the basic estimate and the estimate with overlap analysis.

We see that the estimate that does not take the overlap into account exhibits a huge error (over 70% on average for both sequences). Taking the overlap into account changes this picture considerably; it results in a tight upper bound of virtually the same shape (the error is less than 12% on average). We explain this difference by the fact that the overlap analysis has made this estimate ‘immune’ to the number of scenario transitions.

This case study is an example of an application where overlap analysis helps to obtain reasonable prediction accuracy, so it indeed extends the applicability of our method. Moreover, it potentially offers opportunity to improve the prediction accuracy by increasing the number of scenarios without introducing too much error caused by the extra scenario transitions.

6. Conclusions

We have presented an execution-time prediction method for stream-oriented applications. It can be used for run-time management of power consumption and quality-of-service. It is designed for an abstract multiprocessor platform that supports timing predictability.

The core of our method is an algebraic expression, which gives a conservative estimate of the execution time of a loop containing several data-dependent tasks distributed between multiple processors, whereby tasks execute in a data-driven as-soon-as-possible pipelined manner. Our execution time estimate is a function of parameters provided a priori in headers of the input stream. The coefficients of the expression provide abstraction from implementation details.

To the best of our knowledge, we are the first to provide an accurate execution time prediction method for a broad class of data-dependent stream-oriented applications running on multiprocessors. The accuracy is ensured by a novel timing analysis technique for the synchronous dataflow model of computation. The main mathematical difficulty that we overcame is the dynamic variation of task delays. First, we round the task delays to a few quantization levels, called scenarios, and split the loop execution into intervals corresponding to those levels. Then, we construct the time shapes of the intervals and tightly combine them together. This technique enables a trade-off between the prediction accuracy and the run-time overhead: increasing the number of quantization levels leads to better accuracy.

To illustrate how our ideas can be applied in practice, we considered a working example of an MPEG-4 shape decoder. This case study shows promising results, and we conclude that we have introduced essential ingredients for providing tight and conservative run-time performance estimation, which works even for dynamic application like MPEG-4.

In our future work, we will consider more case studies in the domain of video/audio decoding and try to exploit our prediction method in a quality-of-service manager. We may also look for methods to make the run-time overhead as low as possible and to perform as much analysis as

possible at design time. Some other interesting topics are still open, first of all, a general method to define scenarios given the application.

7. References

[1] A.C. Bavier, A.B. Montz, and L.L. Peterson, "Predicting MPEG Execution Times", *Proc. Of the ACM SIGMETRICS'98*, pp. 131-140, June 1998.

[2] J. Bormans, et al, "Terminal QoS: Advanced Resource Management for Cost-Effective Multimedia Appliances in Dynamic Contexts", Chapter in *Ambient Intelligence: Impact on Embedded System Design*, Kluwer, pp. 183-201, 2003.

[3] A. Dasdan, and R.K. Gupta, "Faster Maximum and Minimum Cycle Algorithms for System-Performance Analysis", in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10): 889-899, 1998.

[4] R. Hoes *Predictable Dynamic Behaviour in NoC-based Multiprocessor Systems-on-Chip*, Master's Thesis, Eindhoven University of Technology, 2004.

[5] E.A. de Kock, et al., "YAPI: Application Modeling for Signal Processing Systems", in *Proc. 37th Design Automation Conference*, ACM, pp. 402-405, 2000.

[6] M. Pastrnak, et al, "Data-flow Timing Models of Dynamic Multimedia Applications for Multiprocessor Systems.", in: *Proc. IWSOC-04*, IEEE, pp. 206-209, 2004.

[7] P. Poplavko, et al, "Task-level Timing Models for Guaranteed Performance in Multiprocessor Networks-on-Chip", in *Proc. CASES-03*, ACM, pp. 63-72, 2003.

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

[9] K. Richter, M. Jersak, and R. Ernst, "A Formal Approach to MP-SoC Performance Verification", *IEEE Computer,* vol. 36, no. 4, pp. 60-67, 2003

[10] E. Rijpkema, K.G.W. Goossens, and A. Radulescu, "Trade Offs in the Design of a Router with Both Guaranteed and Best-Effort Services for Networks on Chip", in *Proc. of DATE'03*, ACM, pp. 350-355, 2003.

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

[12] P. Poplavko, et al, "Mapping of an MPEG-4 Shape-Texture Decoder onto an On-chip Multiprocessor", in *Proc. PRORISC-2003*, Technology Foundation STW, The Netherlands, pp. 139-147, 2003.

[13] T. Amon, et al, "An Algorithm for Exact Bounds on the Time Separation of Events in Concurrent Systems", in: *Proc. ICCD-93*, pp. 166-173, IEEE, 1993.