Predicting the Throughput of Multiprocessor Applications under Dynamic Workload

Peter Poplavko, Marc Geilen and Twan Basten
Eindhoven University of Technology
[p.poplavko, m.c.w.geilen, a.a.basten]@tue.nl

***Abstract*— This work contributes to throughput calculation for real-time multiprocessor applications experiencing dynamic workload variations. We focus on a method to predict the system throughput when processing an arbitrarily long data frame given the meta-characteristics of the workload in that frame. This is useful for different purposes, such as resource allocation or dynamic voltage scaling in embedded systems. An accurate enough analysis is not trivial when two factors are combined: parallelism and dynamic workload variations. In earlier work, two analysis methods showed good accuracy for several application examples, but no comparative experiments were carried out. In this work, we contribute new propositions to the theoretical basis of the previous methods. Based on these propositions, we remove a potential problem in a common subroutine and propose a new analysis method. We compare the methods experimentally. The new method provides a significant reduction of the throughput prediction error, up to 12%.**

I. Introduction

In modern embedded systems, scalable multiprocessors play an increasingly important role. Multiple cores that are coupled to each other via busses, memories and switches pose challenging problems for programming these systems. One of the major challenges is predicting the performance, in order to meet the real-time constraints. For many streaming applications in the multimedia and communication domains, this problem means predicting the throughput, and this is the problem we address in this paper.

The main application of throughput prediction is timing constraint verification of different implementation choices. Examples of this process are resource allocation [3] and the management of limited system resources, e.g., quality scaling [9] and dynamic voltage scaling [8].

A throughput prediction method should analyze arbitrarily long execution runs with a finite overhead. For ensuring good quality, it should give conservative but tight estimations. It should be preferably based on an analytical method so that the results are reliable.

Such a method is difficult to realize when two factors are combined: multiprocessor parallelism and dynamic workload variations. To handle the parallelism with the above mentioned requirements, a so-called steady-state of the system should be detected and analyzed, which is done in many performance analysis approaches, e.g. [15]. However, these techniques typically assume that the system has static characteristics which never change and the same steady state is preserved forever. However, such assumptions do not fit to the dynamic workload situation, which prescribes us to use multiple (temporary) steady states and the transitions between them.

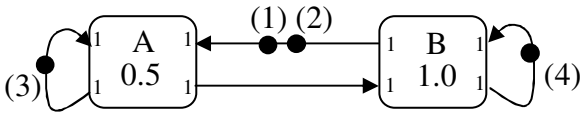


Fig. 1. An SDF example

The only throughput analysis methods known to us that satisfy the requirements are introduced in [6] and [13]. The methods focus on the synchronous dataflow (SDF) model of computation [10], which, as argued in Section II, fits the modeling of multiprocessor systems very well. The above mentioned methods show an excellent throughput prediction accuracy for several multiprocessor application examples.

In this paper, we contribute two important theoretical propositions. From the one proposition we derive a method with significantly improved prediction accuracy. The other proposition helps to eliminate a serious potential convergence problem from the methods of [6], [13] and the new method. First, we give background information by explaining the SDF model in Section II and the relevant previous work in Section III. In Section IV, we present the propositions. The new throughput analysis method is derived in Section V. In Section VI, we perform experimental evaluation to compare the methods, using a set of synthetic benchmarks and a real benchmark. Section VII summarizes the conclusions and looks at future work.

II. Synchronous Dataflow Graphs

The SDF model of computation [10] is represented by SDF graphs. An example is shown in Figure 1. An SDF graph is a directed graph, in which the nodes are called actors. They model the processing, scheduling and communication tasks. Every actor is connected to the graph edges by inputs and outputs. Every input and output has a data rate. In Figure 1 all rates are 1. Such graphs are called homogeneous SDF graphs (HSDF).

The edges of the graph are (potentially unbounded) queues for sending tokens between the actors. Some edges carry initial tokens. For the sake of this paper, we assume every initial token has an index. In Figure 1, the indices of the initial tokens are enclosed in parentheses. The tokens are dynamic data items, consumed and produced by actors in the course of execution.

Execution of every actor is a sequence of firings. An actor starts a new firing at the first moment when it has on each input a number of tokens at least the rate of the input. In this case, the actor is said to be enabled for firing. These tokens are consumed by the actor at the beginning of actor firing. For example, the first firing of actor A in Figure 1 is initially enabled because it can consume a token at each input. After the start of an actor firing, the firing is completed only after an interval called the *actor firing time*. In Figure 1, the firing times are constant (0.5 and 1.0) but in Section III.C we also represent firing time variations in our analysis model. At the firing completion, each output produces new tokens. The amount of tokens produced is equal to the output's rate.

An SDF graph *iteration* is a minimal non-empty set of firings of all actors such that in the end every edge has the same number of tokens as initially. For HSDF graphs, every actor fires exactly once per iteration, but in general different actors fire a different number of times, see [11] for details. Any SDF execution that does not deadlock and that does not build up an unbounded number of tokens in any of its edges is composed of such iterations.

To explain and analyze the graph's behavior, with every initial token j in iteration n ($n > 0$) we associate *release times* $x_{n,j}$. Time $x_{n,j}$ is defined as the moment when another token shifts into the place represented by the token index j in iteration n. For illustration purposes, we also use *capture time* i.e. the moment of time when the token departs from place j in iteration n.

Consider Figure 2(b) a few pages ahead. It illustrates the execution of the SDF graph of Figure 1 using a Gantt chart, where 'resources' (i.e. the points at the vertical axis) correspond to the four initial tokens. The 'tasks' that occupy the 'resources' are the periods of time between the token capture and release times in subsequent graph iterations. The odd iterations are shown with white tasks, the even ones with grey. This figure illustrates that different iterations may interleave with each other in time.

The SDF model of computation turns out to be very useful to model not only the application, but also the multiprocessor mapping, RTOS scheduling and communication. In the literature, a broad research has been carried out on this subject. A pioneering work in this direction assumes bus-based multiprocessors [4]. HSDF models for dedicated FIFO connections and on-chip networks were proposed in [12], for TDMA schedulers in [5] and for the general class of latency-rate schedulers in [17]. Because these models are composable, the interplay of all these components can be captured in one SDF model, which can act as an input for our performance analysis approach. Note that this SDF model is always represented by a strongly connected graph and we use this fact implicitly in our reasoning.

III. Predicting the Throughput by Scenarios

A. Parameter Function

A major timing metric of a multiprocessor application is the time required to process a *frame*, i.e. a given number of subsequent data samples. In terms of SDF graphs, it is the time required to perform a given set of subsequent iterations. We refer to this time as *frame execution time*, denoted Δ_N, where N is the number of graph iterations in the frame. The reciprocal value, N/Δ_N, is equal to the throughput. Therefore, we consider execution time prediction to be synonymous of throughput prediction.

In our work, we use a scenario-based performance analysis approach. A *scenario* is a set of application execution behaviors with similar resource usage [7]. Goal of the scenario-based execution time prediction is to estimate the frame execution time by a linear equation of the form: $\Delta_N \leq \alpha(0) + \sum_i \alpha(i)F(i)$. The right-hand part of this inequality is the *parameter function*. Note that the '$\leq$' sign indicates it is a conservative estimate, in line with our requirements. The $\alpha(i)$ are constant scenario *coefficients*, i.e., the constant contributions of a parameter to the execution time, and the $F(i)$ are *parameters*, typically variables counting the number of invocations of the scenario. The parameters are chosen to be implementation-independent meta-characteristics of the workload that are assumed to be given. For example, the I and P blocks in video coding algorithms can act as scenarios, the total counts of I and P blocks in a video frame can act as parameters and the conservative processor cycle counts to process I and P blocks can be used to derive coefficients.

The main purpose of a performance analysis method is to calculate the optimal coefficients such that the estimation is conservative and the error (i.e. the difference between the parameter function and the execution time) is minimized. This is the central problem of this paper. Throughout this paper, we use small Greek letters for the values that act as scenario coefficients.

In the remainder of this section, we show a state-of the-art ([6, 13]) derivation of SDF frame execution time in terms of a parameter function. We use it as a basis for our contributions presented in Sections IV and V.

B. Analyzing a Single SDF Scenario

In the context of SDF graphs, one defines a scenario as a mode of graph execution where the same set of firing times of all actors is constantly repeated at every graph iteration. This definition is convenient, because a graph iteration often corresponds to the realization of different processing stages for the same data sample. If one can distinguish a finite set of possible data sample *types* (e.g. I-block and P-block in an earlier example) this immediately corresponds to a set of scenarios, because the processing times for the same type can be approximated by constant processing times [8]. If the types cannot be distinguished manually, [8] proposes a general approach to distinguish them automatically.

One can apply well-known analytical tools to characterize the graph's timing behavior as long as a graph stays in the same scenario. In the rest of this subsection, we briefly summarize the tools that are relevant for our purposes.

To express the mathematical relationship between the token release times in different iterations, the so-called *max-plus matrix algebra* [2] is traditionally applied. The major difference from the 'usual' algebra is that for matrix

products, addition is replaced by the max operation and multiplication is replaced by addition. For example,

$$\begin{bmatrix} 0.1 & 5.0 \\ 5.4 & 0.2 \end{bmatrix} \cdot \begin{bmatrix} 4.0 \\ 0.3 \end{bmatrix} = \begin{bmatrix} \max(0.1+4.0,\ 5.0+0.3) \\ \max(5.4+4.0,\ 0.2+0.3) \end{bmatrix} = \begin{bmatrix} 5.3 \\ 9.4 \end{bmatrix}$$

Adding (or subtracting) a constant to a vector or matrix is short-hand notation for increasing (or decreasing) every element, e.g. if $\bar{\mathbf{a}} = [5.0; 1.5; 7.0]$ then $2.1 + \bar{\mathbf{a}} = [7.1; 3.6; 9.1]$. The *norm* $||\cdot||$ is the maximal element, e.g. $||\bar{\mathbf{a}}|| = 7.0$. The *normalization* operator subtracts the norm from the vector: $\bar{\mathbf{a}}^{norm} = \bar{\mathbf{a}} - ||\bar{\mathbf{a}}||$.

The state of the graph is represented by a *state vector* $\bar{\mathbf{x}}_n$, where n is the iteration index. It is a column-vector with R elements, where R is the number of initial tokens in the graph. The i-th element $\{x_n\}_i$ is the release time of token i in iteration n, so $\{x_n\}_i = x_{n,i}$.

The state vector in iteration $n+1$ can be obtained from the state vector in iteration n by $\bar{\mathbf{x}}_{n+1} = \mathbf{G} \cdot \bar{\mathbf{x}}_n$, where $\mathbf{G}$ is an $R \times R$ matrix that characterizes the graph in the given scenario and can be calculated by an algorithm given in [6]. For HSDF graphs, the matrix element at row i column j gets the value of the longest (in terms of the total of firing times) token-free graph path from initial token j to initial token i. If there are no such paths, value $-\infty$ is assumed. For example, in Figure 1, the longest path from token 2 to token 1 is 0.0, so $\mathbf{G}_{12} = 0.0$. There are no token-free paths from token 3 to token 1, so $\mathbf{G}_{13} = -\infty$.

An important property of a max-plus matrix is the solution of the eigenvalue equation: $\mathbf{G} \cdot \bar{\mathbf{x}} = \bar{\mathbf{x}} + \lambda$, where $\bar{\mathbf{x}}$ is a *max-plus eigenvector* and λ is the *max-plus eigenvalue* of matrix $\mathbf{G}$. The eigenvalue represents the average interval between iterations in steady state. The meaning of an eigenvector is a periodic schedule. Indeed, if the state vector is equal to an eigenvector, then after one iteration the state vector is the same except for an addition of λ, and after two iterations it is the same plus twice the λ, and so on.

Not only an eigenvector leads to a periodic execution of the SDF graph. According to a well-known theorem ([2 §3.7]), for any initial vector $\bar{\mathbf{r}}_{start}$, there exist T and W such that for any $n > 0$ we have $\mathbf{G}^{T+n} \cdot \bar{\mathbf{r}}_{start} = \mathbf{G}^{T+n-W} \cdot \bar{\mathbf{r}}_{start} + \lambda W$, which means that the graph executes in a W-periodic regime, the λ being the *average iteration interval* over the W iterations in the period. We refer to the smallest such T as the transient iteration count, because it reflects the number of 'transient' iterations of the graph before it enters the periodic regime, i.e. the 'steady state'.

The eigenvalue and an eigenvector for an SDF graph can be calculated using an efficient iterative search algorithm of [6]. If multiple eigenvectors exist then the eigenvector found by this search algorithm depends on the initial search point. To calculate the set of *basis* eigenvectors defining the space of all eigenvectors one can apply the algorithm of [2 §3.7.2].

C. Analyzing Multiple Scenarios

In general, an SDF scenario model for a given application consists of a finite set of scenarios indexed by $s = 1 \ldots S$, corresponding to different data sample types processed by the application. Because the scenarios have different actor firing times, every scenario is characterized by a distinct matrix $\mathbf{G}(s)$ which has a distinct eigenvalue $\lambda(s)$.

It is convenient to split the processing of a frame into intervals $p = 1 \ldots P$, where every interval is a maximal range of subsequent graph iterations with the same scenario s_p. Every iteration n belongs to a certain interval $p(n)$. For example, suppose that the number of iterations in a frame is 10, and the scenario index s progresses as $\{1,3,3,3,3,1,1,2,2,2\}$. Then there are four intervals, and $s_1 = 1$, $s_2 = 3$, $s_3 = 1$, $s_4 = 2$.

The evolution of the graph state vector in a frame is expressed by:

$$\bar{\mathbf{x}}_{n+1} = \mathbf{G}(s_{p(n)})\bar{\mathbf{x}}_n, \quad n = 0 \ldots N-1 \tag{1}$$

where without loss of generality we assume $\bar{\mathbf{x}}_0 = [0; \ldots; 0]^T$. The frame execution time can be written as:

$$\Delta_N = ||\bar{\mathbf{x}}_N|| \tag{2}$$

[6] introduces a so-called *reference schedule method*, which estimates the frame execution time as the sum of contributions of all intervals, whereby the contribution of arbitrary interval p is expressed in the form:

$$\widetilde{\Delta}(I) = \tau + \lambda \cdot I \tag{3}$$

where $\lambda = \lambda(s_p)$, I is the iteration count of interval p and τ is called the *transition delay*, because it reflects the transient effect of the transition from previous scenario s_{p-1} (or from the initial state) to the steady state of the current scenario, s_p. The term $\lambda \cdot I$ reflects the throughput of the SDF graph in the steady state.

[6] defines the transition delay τ such that starting from a certain state $\bar{\mathbf{r}}_{start}$ after any number of iterations in scenario s_p the final state vector $\bar{\mathbf{x}}$ is separated from a certain target state $\bar{\mathbf{r}}_{end}$ by at most time $\widetilde{\Delta}(I)$, i.e., $||\bar{\mathbf{x}} - \bar{\mathbf{r}}_{end}|| \leq \widetilde{\Delta}(I) = \tau + \lambda \cdot I$. Let us look for minimal such τ, to make bound $\widetilde{\Delta}$ as tight as possible. Observing that $\tau \geq ||\bar{\mathbf{x}} - \bar{\mathbf{r}}_{end} - \lambda I||$ and $\bar{\mathbf{x}} = \mathbf{G}^I \cdot \bar{\mathbf{r}}_{start}$, where $\mathbf{G} = \mathbf{G}(s_p)$, we see that the minimal τ is a function τ_{gen} defined as:

$$\tau_{gen}(\bar{\mathbf{r}}_{start}, \mathbf{G}, \bar{\mathbf{r}}_{end}) = \max_{n=1 \ldots T} ||\mathbf{G}^n \cdot \bar{\mathbf{r}}_{start} - \bar{\mathbf{r}}_{end} - \lambda n|| \tag{4}$$

where T is the transient iteration count. In this expression we have made use of the W-periodic regime theorem.

Vectors $\bar{\mathbf{r}}_{start}$ and $\bar{\mathbf{r}}_{end}$ are called the *start schedule* and the *end schedule*. According to [6], both vectors are normalized, $\bar{\mathbf{r}}_{start}$ estimates the normalized state vector $\bar{\mathbf{x}}_n^{norm}$ before the start of the interval and $\bar{\mathbf{r}}_{end}$ estimates this vector after the completion of the interval. Due to max-plus normalization, any schedule should satisfy:

$$||\bar{\mathbf{r}}_{end}|| = 0 \tag{5}$$

The start schedules are implied from the end schedules. $\bar{\mathbf{r}}_{start}$ is equal to $\bar{\mathbf{r}}_{end}$ of the previous interval except for the first interval, where $\bar{\mathbf{r}}_{start} = [0.0;\ 0.0;\ \ldots;\ 0.0]^T$.

In the reference schedule method, one can choose arbitrary $\bar{\mathbf{r}}_{end}$, and the $\bar{\mathbf{r}}_{start}$ are implied from the $\bar{\mathbf{r}}_{end}$. However, the

accuracy of the reference schedule method is sensitive to the correct choice of the $\bar{\mathbf{r}}_{end}$. Although [6] suggests the possibility of different $\bar{\mathbf{r}}_{end}$ for different intervals, the method assumes that the $\bar{\mathbf{r}}_{end}$ are the same and referred to as $\bar{\mathbf{r}}_{ind}$. Notation '*ind*' refers to a schedule that is independent of the scenario it is applied to. $\bar{\mathbf{r}}_{ind}$ is calculated as an eigenvector of matrix $\mathbf{G}_{all}$, where $\mathbf{G}_{all} = \max_{s=1...S}(\mathbf{G}(s) - \lambda(s))$. We call this method the *scenario-independent reference schedule* method. Summing up Equality (3) for all intervals, we get the parameter function that estimates the frame execution time in this method [6]:

$$\Delta_N \leq \tau_{ind-ini} + \sum_s (\lambda(s)J(s) + \tau_{ind}(s)L(s)) \qquad (6)$$

where $\tau_{ind-ini} = \tau_{gen}([0;\ldots 0]^T, \mathbf{G}(s_1), \bar{\mathbf{r}}_{ind})$; $J(s)$ is the total count of iterations in scenario s; $L(s)$ is the total number of intervals of scenario s except for the first interval. $\tau_{ind}(s)$ and $\lambda(s)$ are scenario coefficients, and $J(s)$ and $L(s)$ are scenario parameters.

The method that we propose in Section V uses the reference schedule methodology too, but we calculate the schedules differently.

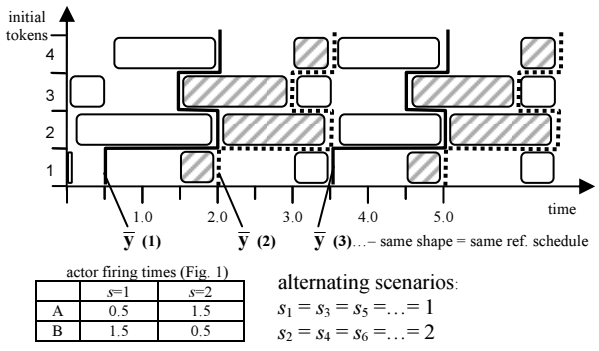
D. Reference Schedule: a Discussion

In this section, we illustrate the reference schedule method using the Gantt charts to see what is different in our method introduced later. First we need to add some useful notations. Let $\bar{\mathbf{r}}_{end}(p)$ be the end schedule of interval p. Observe that the reference schedule methodology estimates the state vector at the completion of interval p as the sum of $\widetilde{\Delta}(I)$ for all the intervals up to that point plus vector $\bar{\mathbf{r}}_{end}(p)$. We use notation $\bar{\mathbf{y}}(p)$ for that estimate.

Using vectors $\bar{\mathbf{y}}(p)$, we can imagine the working of the method as follows. Let us add to the SDF graph a virtual 'scheduler' engine that can interfere with the SDF graph execution between the graph iterations. After a token has been released, the scheduler can hold it, delaying its capture time until a certain scheduled time. Suppose that the scheduler only interferes at the end of the scenario intervals, and holds the tokens until the times specified in $\bar{\mathbf{y}}(p)$. Such a scheduler models the operation of a reference schedule method. Note that in reality such a scheduler is not used and actors fire as soon as they are enabled. Due to monotonicity of the behavior of an SDF graph, the behavior of the model with this hypothetical scheduler is a conservative upper bound of the real behavior.

For example, Figures 2(a) and (b) show Gantt charts for the graph in Figure 1, as already explained before in Section II. Two scenarios are assumed, and their firing times are given in the table in the figure. It is assumed that the graph alternatingly switches between the two scenarios. The diagrams for vectors $\bar{\mathbf{y}}(p)$ are plotted with bold lines, solid for the odd p and dotted for the even p.

In Figure 2(a), all the $\bar{\mathbf{y}}(p)$ diagrams have the same shape, which corresponds to the independent reference schedule $\bar{\mathbf{r}}_{ind} = [-1.5;\ 0.0;\ -0.5;\ 0.0]^T$. This turns out to be an inefficient solution, because the virtual scheduler delays

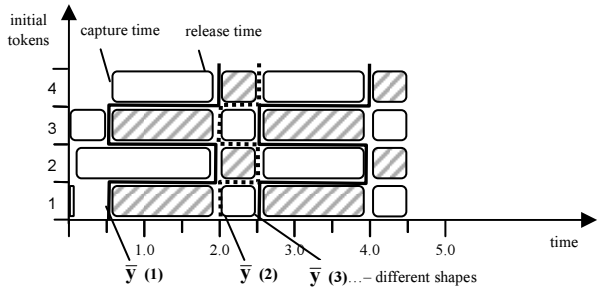


actor firing times (Fig. 1)

	s=1	s=2
A	0.5	1.5
B	1.5	0.5

alternating scenarios:
$s_1 = s_3 = s_5 = \ldots = 1$
$s_2 = s_4 = s_6 = \ldots = 2$

(a) Independent reference schedule leads to poor results due to the same shape of $\bar{\mathbf{y}}$ for all scenario transitions



(b) Specific reference schedule

Fig. 2. An SDF simulation demonstrating the superiority of a specific reference schedules over independent schedules

token 3 by 1.0 at every transition from scenario 1 to scenario 2. In Figure 2(b) we see the graph execution with two specific schedules: $\bar{\mathbf{r}}_{end}(1) = [-1.5;\ 0.0;\ -1.5;\ 0.0]^T$ for the odd intervals and $\bar{\mathbf{r}}_{end}(2) = [-0.5;\ 0.0;\ -0.5;\ 0.0]^T$ for the even ones. The execution coincides with the self-timed execution, leading to a zero estimation error. This is due to the fact that, in this example, the shapes of the specific schedules ideally match the shapes of the token release at the end of the scenario intervals. Because these shapes are essentially different for the two scenarios, no scenario-independent schedule would match both of them well.

Our method introduced in Section V exploits different schedules for different intervals to overcome this problem.

IV. Transition Delay Calculation

A. Improved Calculation of Transition Delay

Equality (4) is used to calculate the transition delays in the two previous throughput prediction methods of [13, 6]. We need to calculate transition delays in the new method as well. In the previous work, this equality was applied *directly* and thus algorithmic complexity depends on transient iteration count T. This creates a potential threat for the performance of the throughput prediction, because T can become uncontrollably large. In this subsection, we remove this potential problem based on the following proposition.

Proposition 1 Transient iteration count T in the definition of τ_{gen} (Equality (4)) can be replaced by $\min(R,T)$, where R is the number of rows/columns of matrix $\mathbf{G}$.

Moreover:

$$\tau_{gen}(\bar{\mathbf{r}}_{start}, \mathbf{G}, \bar{\mathbf{r}}_{end}) = ||\widetilde{\mathbf{G}}^{+} \cdot \bar{\mathbf{r}}_{start} - \bar{\mathbf{r}}_{end}|| \qquad (7.1)$$

where:

$$\widetilde{\mathbf{G}}^{+} = \max_{n=1\ldots R} \widetilde{\mathbf{G}}^{n} \tag{7.2}$$

$$\widetilde{\mathbf{G}} = \mathbf{G} - \lambda \tag{7.3}$$

Proof See the extended version of this paper in [14].

Example (adapted from [2 §3.7]). Let

$$\mathbf{G} = \begin{bmatrix} 99.0 & 0.0 \\ 100.0 & 100.0 \end{bmatrix}.$$

Suppose $\bar{\mathbf{r}}_{start} = \bar{\mathbf{r}}_{end} = [0;\ \ldots;\ 0]^T$. The argument of the max operator in Eq. (4) evolves for $n = 1 \ldots 100$ as $[1.0;\ 0.0]^T$; $[2.0;\ 0.0]^T; \ldots [100.0;\ 0.0]^T$ and for $n \geq 100$ it stays constant, so $T = 100$. From this, we may conclude that we have $\tau_{gen} = 0.0$. Proposition 1 gives us the possibility to reach this conclusion after 2 iterations instead of 100.

The '+' matrix operator defined in (7.2) is commonly referred to as a transitive closure or longest-path matrix [2 §1.2]. It can be calculated by an $O(R^3)$ all-pair longest path algorithm. Note that [1] also employs a transitive closure to calculate a bound on a time difference between events. However, [1] applies it to the 'steady-state' part of the model exploration, not to the 'transient' part.

B. Reference Schedule with Minimal Delay

Recall that Equality (3) gives an upper-bound on the execution time of a given interval I_p. Suppose that we fix $\bar{\mathbf{r}}_{start}$ and would like to find such an $\bar{\mathbf{r}}_{end}$ that this upper bound is minimized. This would certainly serve our intention to have an execution time estimate that is as tight as possible, if we were focusing on the execution time of one interval separately from the other intervals.

Observe that at the right-hand part of Equality (3), the only part that depends on $\bar{\mathbf{r}}_{end}$ is $\tau_{gen}(\bar{\mathbf{r}}_{start}, \mathbf{G}(s_p), \bar{\mathbf{r}}_{end})$. Then our problem of minimizing the estimate for a single scenario interval is solved by the following proposition.

Proposition 2 For the given matrix $\mathbf{G}$ and start schedule $\bar{\mathbf{r}}_{start}$ and given the constraint $||\bar{\mathbf{r}}_{end}|| = 0$ (as in Equality (5)), the minimum transition delay is reached only for an end schedule that satisfies the following criterion: $\bar{\mathbf{r}}_{end-min} \leq \bar{\mathbf{r}}_{end} \leq [0.0;\ 0.0;\ \ldots;\ 0.0]^T$, where:

$$\bar{\mathbf{r}}_{end-min} = (\widetilde{\mathbf{G}}^{+} \cdot \bar{\mathbf{r}}_{start})^{norm} \tag{8}$$

Proof See the extended version of this paper in [14].

We use this proposition to derive a new method.

V. The Supermatrix Method

A. Scenario-specific Reference Schedule

We propose a method with an improved accuracy w.r.t. the scenario-independent reference schedule method at the expense of an increased analysis cost; the method uses *scenario-specific* reference schedules as explained below.

A scenario-specific schedule is an end schedule that depends on the interval's scenario; we use notation $\bar{\mathbf{r}}_{spec}(s)$ for it. The schedule can be potentially adjusted to its scenario in such a way that it would yield a better accuracy than the scenario-independent method - recall the example of Figure 2.

In this method, $\bar{\mathbf{r}}_{start} = \bar{\mathbf{x}}_0$ for $p = 1$ and $\bar{\mathbf{r}}_{start} = \bar{\mathbf{r}}_{spec}(s_{p-1})$ for $p > 1$. The end schedule $\bar{\mathbf{r}}_{end}$ is $\bar{\mathbf{r}}_{spec}(s_p)$ for every p. Using these schedules and summing up the execution time estimates of all intervals, we have:

$$\Delta_N \leq \tau_{spec-ini} + \sum_{s} \lambda(s) J(s) + \sum_{s,t,s\neq t} \tau_{spec}(s,t) K(s,t) \tag{9}$$

where: $\tau_{spec-ini} = \tau_{gen}(\bar{\mathbf{0}}, \mathbf{G}(s_1), \bar{\mathbf{r}}_{spec}(s_1))$ is the initial delay. Scenario coefficient $\tau_{spec}(s,t) = \tau_{gen}(\bar{\mathbf{r}}_{spec}(s), \mathbf{G}(t), \bar{\mathbf{r}}_{spec}(t))$ is the delay of the transition from scenario s to scenario t; scenario parameter $K(s,t)$ is the total number of transitions from s to t. Parameter $J(s)$ and coefficient $\lambda(s)$ have the same meaning as in Eq. (6). Note that Eq. (9) has more scenario parameters than Eq. (6). This is necessary to make use of the scenario-specific schedules to achieve better accuracy.

B. The Minimal-error Coefficient Optimization Problem

Note that the first term in Eq. (9) is insignificant if N is large enough. The second term cannot be influenced. Therefore, to minimize the prediction error we are focusing on the third term.

The optimization problem we are considering now is as follows. The problem instance consists of the parameter values $K(s,t)$ and the set of the scenario matrices $\mathbf{G}(s)$. We have to fill the set of scenario-specific reference schedules with vector values $\bar{\mathbf{r}}_{spec}(s)$ such that the scenario coefficients $\tau_{spec}(s,t)$ induced by these schedules yield the minimal contribution in the third term of Eq. (9). Note that this is a particular case of the minimal-error coefficient optimization problem mentioned in Section III.A.

Similar to [6], in the solution method proposed in the next section we only use the scenario matrices $\mathbf{G}(s)$ and not the frame specific parameter values, which will only become available at run-time. This approach enables the reuse of the calculated coefficients $\tau_{spec}(s,t)$ for multiple frames, independently of the $K(s,t)$. For many applications, $\mathbf{G}(s)$ are known at design time [6], which means that using our method one can calculate the reference schedules at design time as well.

C. A Method to Calculate the Reference Schedules

Our method introduced here is a heuristic solution for the problem introduced above. For the reason that becomes apparent later, we call this method the *supermatrix method*.

Consider an arbitrary interval. Suppose that that interval is in scenario t. Similar to Section IV.B, consider the problem of minimizing the transition delay in that interval. The difference is, however, that instead of one start schedule we have a set of possible start schedules: $\bar{\mathbf{r}}_{start} \in \{\bar{\mathbf{r}}_{spec}(s) \mid 1 \leq s \leq S, s \neq t\}$.

In this heuristic approach, we define the end schedule $\bar{\mathbf{r}}_{spec}(t)$ as the optimal schedule for an aggregate start schedule $\bar{\mathbf{r}}_{start-aggr}(t)$, representing a certain weighed combination of the possible start schedules:

$$\bar{\mathbf{r}}_{start-aggr}(t) = \max_{s,s\neq t}(\bar{\mathbf{r}}_{spec}(s) + w(s)) \tag{10}$$

$$\begin{bmatrix} \mathbf{E} & \widetilde{\mathbf{G}}^+(1) & \cdots & \widetilde{\mathbf{G}}^+(1) & \widetilde{\mathbf{G}}^+(1) \\ \widetilde{\mathbf{G}}^+(2) & \mathbf{E} & \cdots & \widetilde{\mathbf{G}}^+(2) & \widetilde{\mathbf{G}}^+(2) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \widetilde{\mathbf{G}}^+(S-1) & \widetilde{\mathbf{G}}^+(S-1) & \cdots & \mathbf{E} & \widetilde{\mathbf{G}}^+(S-1) \\ \widetilde{\mathbf{G}}^+(S) & \widetilde{\mathbf{G}}^+(S) & \cdots & \widetilde{\mathbf{G}}^+(S) & \mathbf{E} \end{bmatrix}$$

Fig. 3. Matrix $\mathbf{G}_{SUP}$ – the 'supermatrix'

where $w(s)$ is the weight determining the degree of influence of scenario s in the aggregate schedule. Substituting the start schedule from Eq. (10) to Eq. (8), we have:

$$\bar{\mathbf{r}}_{spec}(t) = \bar{\mathbf{z}}(t)^{norm} \tag{11}$$

where:

$$\bar{\mathbf{z}}(t) = \widetilde{\mathbf{G}}^+ \cdot \bar{\mathbf{r}}_{start-aggr}(t) + c \tag{12}$$

c can be selected arbitrarily, but below we will choose the only possible value leading to feasible solutions.

In Eq. (10), we choose to use the weights $w(s) = ||\bar{\mathbf{z}}(s)||$. We do this because this allows us to solve the resulting set of equations analytically, by a known method. With these weights we transform Equalities (10-12) to a system of equations equivalent to the eigenvector equation where constant c is the eigenvalue:

$$t = 1 \ldots S: \quad \bar{\mathbf{z}}(t) = \widetilde{\mathbf{G}}^+(t) \cdot \max_{s, s \neq t}(\bar{\mathbf{z}}(s)) + c \tag{13}$$

To make it more obvious that eigenvector methodology can be re-applied here, we rewrite Eq. (13) in matrix form:

$$\bar{\mathbf{z}}_{SUP} = \mathbf{G}_{SUP} \cdot \bar{\mathbf{z}}_{SUP} + c \tag{14}$$

where $\bar{\mathbf{z}}_{SUP}$ is a concatenated vector of size SR: $\bar{\mathbf{z}}_{SUP} = [\bar{\mathbf{z}}^T(1)\ \bar{\mathbf{z}}^T(2)\ \ldots \bar{\mathbf{z}}^T(S)]^T$; and $\mathbf{G}_{SUP}$ is a concatenated $SR \times SR$ matrix composed of $R \times R$ block submatrices, shown in Figure 3. This matrix consists of 'super-rows' filled with matrices $\widetilde{\mathbf{G}}^+(t)$ everywhere except at the 'super-diagonal', where matrix $\mathbf{E}$ is filled. The latter is an $R \times R$ matrix whose elements are all $-\infty$. We refer to $\mathbf{G}_{SUP}$ as the *supermatrix*.

Extracting $\bar{\mathbf{z}}_{SUP}$ as an eigenvector of $\mathbf{G}_{SUP}$ and applying Equality (11), decomposing $\bar{\mathbf{z}}_{SUP}$ into vectors $\bar{\mathbf{z}}(1)$, $\bar{\mathbf{z}}(2)$, ..., we obtain all the reference-specific schedules.

Note that in the case of two scenarios, Eqs (10) transform into two equalities in the form of Eq. (8), which means that the two reference schedules are optimal end schedules with respect to each other. The two schedules in Fig. 2(b) are, in fact, obtained from the supermatrix method.

VI. Experimental Evaluation

In this section, we compare the accuracy of the supermatrix method experimentally with the independent reference schedule method of [6] and the minimum overlap method of [13]. In both cases for the eigenvector calculation we used the iterative search algorithm of [6] with the initial search point $[0; \ldots; 0]^T$. We ran experiments for a set of random benchmarks as well as a real application.

To generate the SDF graphs randomly and produce the input for the experiments we used the random SDF graph generator of the open-source SDF3 tool [16]. In all experiments, the generated graphs had 10 actors and 15 edges on average. In addition, we implemented a random generator of SDF scenarios and frames. In the generated frames, all the actors in the generated graph had different firing times in different scenarios. The number of scenarios was set to S=8, the ratio between max and min actor firing time was in most cases 5 and below. The frame iteration count was set to 30. Note that neither the firing time ratio nor the frame iteration count were found to have a significant impact on the prediction quality and overhead. To make the prediction problem complex enough, we set the frequency of scenario transitions to at least 70% of iterations.

TABLE I

HSDF run: average relative error (%) in different graphs

ovp	49	18	0	10	41	0	13	9	41	11	19
ind	6	1	1	1	10	0	2	7	6	5	0
sup	1	0	1	0	3	0	2	2	4	1	0

TABLE II

SDF run: average relative error (%) in different graphs

ind	4	18	9	5	7	5	2	14	1	3
sup	1	12	2	3	2	0	0	2	0	0

TABLE III

JPEG run: average relative error (%) for different images

ovp	55	72	51	36	72	50	50	56	40	52
ind	21	20	14	17	27	18	19	23	17	18
sup	14	16	11	15	16	16	15	15	15	13

For every generated graph, the generator produced multiple frames. In order to verify that the methods are not too sensitive to the changes in the input data, at every frame, a set of scenarios with slightly different actor firing times was offered to them. Therefore, every prediction method had to recalculate the scenario coefficients for every frame (although in practice this can be done once, at design time).

We have run experiments on two sets of graphs: for HSDF and for general SDF graphs. In the HSDF graphs, the total initial token count R was in the range 4-11. In the SDF graphs, the generator had to select larger values of R: 18-25 to ensure absence of deadlock. Every HSDF graph was evaluated with 50 frames, and every SDF graph was evaluated for 10 frames. In all cases, the average total running times of different methods were comparable (mostly 10-50 seconds on a 1.2 GHz CPU). However in the current implementation a fair overhead comparison was not possible, it is postponed for future work.

To evaluate the results, we calculate the frame execution times from simulation and use the result as the reference for relative execution time prediction error. Tables I and II show the results of the accuracy evaluation, where the columns correspond to different graphs. Rows *ovp*, *ind* and *sup* correspond to the minimal overlap, independent schedule, and

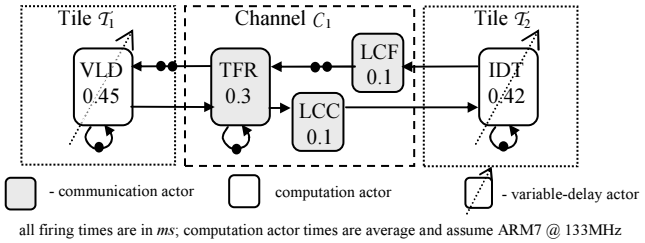


Fig. 4. HSDF model: JPEG decoder mapped to two processing tiles

supermatrix methods. Table II misses the minimal overlap results, as it supports only HSDF graphs.

From the tables, we see that in almost all the cases, the supermatrix method produced the best results, improving the accuracy by up to 12%. It also demonstrates more reliable accuracy, as the error variation among different graphs is smaller. The minimal overlap method shows in almost all the cases worse results, although it uses the same meta-characteristics as the supermatrix method [13].

Figure 4 shows the HSDF graph of a JPEG decoder mapped to two processor tiles (i.e. multiprocessor segments with local memory systems), communicating via a network channel. This example is adapted from a case study in [12], but with some differences. The variable-length decoder (VLD) is mapped to a separate tile from the other actors. The inverse transform and scaling operations for all 6 subblocks are now modeled by a single actor (IDT). The IDT actor now has variable firing time, which, in line with [3], depends on an extra parameter - the number of subblocks (0-6) whose count of non-zero DCT coefficients is large enough. We assume that every subblock with more than 5 coefficients contributes 0.12 ms to the IDT firing time and the other subblocks contribute 0.02 ms. The TFR, LCC and LCF actor models the network channel (see [12] for channel modeling).

For JPEG, we introduce scenarios as follows. The firing times of the VLD and IDT actors depend on the decoded bit count and the DCT coefficient count. We split the dynamic range of the bit count into sub-ranges of 100 bits and of the coefficient count into subranges of 10. A combination of the two types of subranges is a scenario. This yields around 400 scenarios, but every image involves only a small subset (typically 7-12). We have measured the execution time prediction error for 10 arbitrary images. We used graph simulation with real VLD and IDT firing times as the reference. The results are presented in Table III. They confirm the best quality of the supermatrix method when compared to the two other methods for a realistic benchmark.

VII. Conclusions and Future Work

In this paper, we have presented an analytical throughput prediction method for variable workload in multiprocessors and potentially other systems whose concurrency can be modeled by SDF graphs, such as asynchronous circuits. This method can be used in the design-time resource allocation for a given workload profile or as a preparatory phase of run-time resource management to estimate the timing costs in different possible run-time application scenarios. We also removed an important potential problem for the overall methodology by giving an algorithm with better and more robust complexity for calculating a common metric, the transition delay.

The proposed method, called the supermatrix method, follows an approach that is able to analyze arbitrarily long application runs with a constant overhead. The experiments demonstrate that the method outperforms the other comparable methods in terms of accuracy, but has a potentially higher overhead.

In future work, we will refine and evaluate the new method for the extended model of computation that allows a different SDF structure and rates in different scenarios [6]. We will also evaluate the new method in other application case studies in terms of its overhead and the energy savings that can be obtained when it is applied in the context of dynamic voltage and frequency scaling.

References

[1] T. Amon, H. Hulgaard S. M. Burns and G. Borriello, "Algorithm for Exact Bounds on the Time Separation of Events in Concurrent Systems", in *proc ICCD-1993*, pp. 166-173, IEEE.

[2] F. Baccelli, G. Cohen, G. J. Olsder, and J. P. Quadrat. *Synchronization and Linearity*. New York: Wiley, 1992.

[3] M. Bereković, H. J. Stolberg, and P. Pirsch, "Multicore System-On-Chip Architecture for MPEG-4 Streaming Video", in *IEEE Trans. Circuits and Systems for Video Technology*, vol. 12, no. 8, pp. 688-699, 2002.

[4] N. Bambha, V. Kianzad, M. Khandelia, and S. S. Bhattacharrya, "Intermediate Representations for Design Automation of Multiprocessor DSP Systems", in *Design Automation for Embedded Systems*, vol. 7, pp. 307-323, Kluwer Academic Publishers, 2002.

[5] M. Bekooij *et al*, "Chapter 15. Dataflow Analysis for Real-time Embedded Multiprocessor System Design," in Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices, *Philips Research Book Series*, vol. 3, Springer, pp. 81-108, 2005.

[6] M.C.W. Geilen, "Synchronous Dataflow Scenarios", in *ACM Trans. Embedded Computing Systems*, 2010.

[7] S. V. Gheorghita *et al*, "A System Scenario based Approach to Dynamic Embedded Systems", in *ACM Transactions on Design Automation of Electronic Systems*, vol. 14, no. 1, 45 pages, Jan. 2009.

[8] S. V. Gheorghita, T. Basten, and H. Corporaal, "Scenario Selection and Prediction for DVS-Aware Scheduling of Multimedia Applications", in *Journal of Signal Processing Systems*, vol. 50, no. 2, pp. 137-161, Springer, 2008.

[9] Y. Huang, A. V. Tran, and Y. Wang. "A Workload Prediction Model for Decoding MPEG Video and its Application to Workload-scalable Transcoding", in *proc. ACMM-2007*, pp. 952-961, ACM.

[10] 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.

[11] T. M. Parks, "Bounded Scheduling of Process Networks". PhD Dissertation, EECS Department, University of California, 1995.

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

[13] P. Poplavko, T. Basten, and J. van Meerbergen, "Execution-time Prediction for Dynamic Streaming Applications with Task-level Parallelism", in *proc. DSD-2007*, pp. 228-235, IEEE.

[14] P. Poplavko, M. Geilen, and T. Basten, *Predicting the Throughput of Multiprocessor Applications under Dynamic Workload*, Technical Report ESR-2010-02, Eindhoven University of Technology, 2010.

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

[16] S. Stuijk, M.C.W. Geilen and T. Basten. "SDF3: SDF For Free", in *proc ACSD-2006*, pp. 276-278, IEEE.

[17] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit, "Modeling Run-time Arbitration by Latency-rate Servers in Dataflow Graphs", in *proc. SCOPES-2007*, pp. 11-22, ACM.