Analysis and visualization of execution traces of dataflow applications

Analysis and Visualization of Execution Traces of Dataflow Applications

Hadi Alizadeh Ara*, Amir Behrouzian*, Marc Geilen*, Martijn Hendriks†*, Dip Goswami*, Twan Basten*†,
*Eindhoven University of Technology, Eindhoven, The Netherlands
†TNO ESI, Eindhoven, The Netherlands

I. Objectives

Embedded applications are an integral part of modern embedded systems. These applications typically have real-time constraints on throughput or latency. Temporal analysis techniques are required at the design stage to ensure that the applications meet their constraints and determine the required amount of resources (processors or memories for the application).

The Synchronous Dataflow Graph (SDFG) [4] model of computation is a popular method for temporal analysis of applications. SDFG represents the application by a graph consisting of *actors* and *channels*. Actors represent the individual computations within the application. Each actor has an execution time which usually indicates the upper bound on the time it requires to complete its *firing* (execution), once it is started. Channels model the dependencies of individual computational tasks on the resources, on input data, and on the data produced by other tasks. When an actor starts and completes firing, it consumes and produces a fixed amount of *tokens* on the FIFO channels respectively. An actor is able to fire if its consumption rates are not greater than the number of tokens on the channels from which it consumes. Figure 1 shows two SDFGs. Each SDFG models a working mode of a single application. Each mode has three actors. The execution time of actors x, y and z are assumed to be $101, 73$ and 125 milliseconds in both modes respectively. The numbers near the channel endings indicate the consumption and production rates. The actors are bound to different processors in different modes. This property is modeled by having a self edge on each actor which contains a token labelled with the name of the processor to which it is bound (e.g. x is bound to processors $P1$ and $P2$ in modes A and B respectively).

The temporal analysis techniques developed for SDFGs can analyse throughput and latency for data-driven execution of the application or when the application is mapped to a predictable platform with limited resources. These techniques output the latency, throughput and possibly the critical path within a graph that models the task and resource dependencies. However, they do not give much tangible information about the system such as when and where the tasks are being processed or in which time intervals which one of the resources are busy or idle. We therefore develop a technique to visualize execution traces of dataflow applications. Visualization gives developers a more detailed understanding of the temporal behaviour of the application. For example visualization of the critical data and resource dependencies, makes it easier to recognize the system's bottlenecks. TRACE [1] is a powerful Gantt chart visualization tool capable of presenting tasks on resources and dependencies between them as a function of time. Moreover, it can run performance analysis such as critical path analysis, latency and throughput analysis based on execution trace information and it can compare traces and visualize their differences for model validation and design-space exploration.

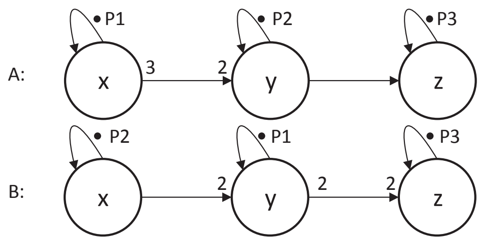

Fig. 1: An example application having two different modes

We have integrated TRACE visualization into the SDF^3 tool. SDF^3 is a tool for temporal analysis of SDFG and various types of dynamic dataflow models such as Cyclo Static Dataflow (CSDF) [2], Senario Aware Dataflow (SADF) [6] and Finite State Machine-SADF (FSM-SADF) [5]. SDF^3 also supports automated mapping of applications onto predictable multi-processor platforms. This integration enables us to visualize execution traces of application tasks and track the resource usage associated with these executions. Moreover, we can utilize the various analysis and visualization options in TRACE such as critical path analysis to visualize the critical dependencies.

II. Methods

In dataflow theory, the set of actor firings that brings the tokens back to the initial token distribution, is called an *iteration*. According to [3], the temporal behaviour of an SDFG can be captured by finding the *time differences* between the production times of tokens at the end and start of an iteration. In this method the production times of each token and consequently the start and the completion times of each actor firing can be represented by a *symbolic time stamp* of the form $t = \max_i(t_i + g_i)$ where t_i are the symbolic availability times of initial tokens ($P1, P2, P3$ for the example). g_i are suitable constants that indicate the time difference between t and t_i. It is $-\infty$ if there is no dependency between t and t_i. In $(max, +)$ algebra we can represent a symbolic time stamp with the vector dot-product $t = \bar{t}.\bar{g}$ where $\bar{t} = [t_{P1}; t_{P1}; t_{P2}]$ for the example. Assuming $\bar{t} = [t_{P1}; t_{P1}; t_{P2}] = [0; 0; 0]$, the start and completion time of the first firing of actor x in mode A of the example can be represented as $[0; -\infty; -\infty]$ and $[101; -\infty; -\infty]$ respectively. This vector representation of the start (completion) time of firings is called the *symbolic start (completion)* time. The time differences g_i are determined by *symbolically simulating* the application graph for one iteration. Symbolic simulation characterizes the time differences of the application by a matrix in $(max, +)$ algebra. This matrix allows to compute the completion times of any iteration using the following equation in $(max, +)$ algebra

$$\gamma_{k+1} = G_m \times \gamma_k \qquad (1)$$

where G_m is called the *characterization matrix* of the application when it is in mode m and γ_k is a vector that determines the start of iteration k. Using Equation 1 we can capture the evolution of the application in time. However, this equation cannot be directly used for tracing purposes, because, it only records the completion time of firings that produce tokens at the end of the iteration. Individual firings inside the iteration have been abstracted into a single matrix multiplication. To overcome this problem, we construct another matrix that allows us to reconstruct detailed information about the individual actor firings from the starting point of an iteration. During the symbolic simulation we store the symbolic start time and completion times. We do this for all firings involved in one iteration of the graph in the order they get enabled and completed. All stored vectors are vertically concatenated to construct a matrix. With this matrix, all firings within the iteration can be calculated from the initial token time stamps γ_k as follows.

$$\tau_k = H_m \times \gamma_k \qquad (2)$$

where H_m is the *tracing matrix* in mode m and τ_k is a vector that contains the start and completion times of all firings within iteration k in the order they are stored during the symbolic simulation. It is important to realize that tracing only requires extra calculations for the iterations for which we want to observe the traces. In the worst-case, the number of firings in a single iteration is exponential in the size of the graph. Nevertheless, the tracing function is computationally efficient in many practical cases.

III. Results

We implemented this tracing algorithm for FSM-SADF in the SDF^3 tool. We generate two files as input to the TRACE tool, a configuration file and the actual trace file. The configuration file defines specific properties of the FSM-SADF executions as actor names, scenario names and iteration indexes. These properties are called *attributes* in the TRACE tool and used to colour and group the traces for different visualization purposes. TRACE considers an individual task execution as a single *claim* which is made on a resource from the start time of the execution to its completion time. The trace file then lists all claims and their attributes for the requested tracing interval. TRACE outputs Gantt charts of *activities*, i.e. task executions, and resource claims for a given trace file. These Gantt charts then can be grouped together and coloured, based on the selected attributes.

Consider the application in Figure 1. Assume it executes iterations in the given mode order: A, B, B, A, A, B. TRACE outputs, by default, a Gantt chart in which every task execution has a different colour. For a better visualization, we group the executions by actor names and use different colours for different iterations with the same colour for task executions within an iteration. Figure 2 shows the Gantt chart of activities and the corresponding processor claims for the given order respectively. By applying the critical path analysis to the traces, the dependencies between the different executions are analysed and critical dependencies are found. Then the colouring of the Gantt charts are changed for better visualization of the critical path, as shown in Figure 3 (with the critical path coloured in red). By visualizing the critical path, we can easily verify that

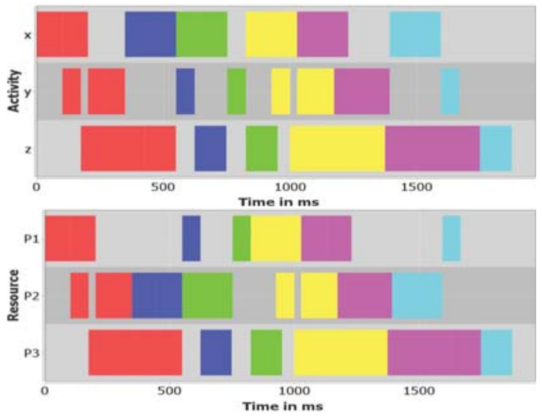

Fig. 2: Gantt charts of actor firings and processor claims

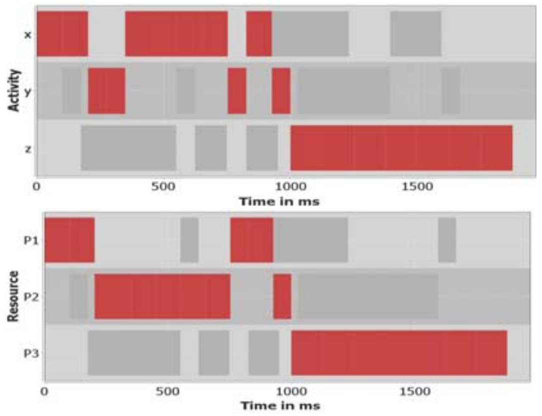

Fig. 3: Gantt charts of actor firings and processor claims, visualizing the results of critical path analysis

for example increasing the frequency of processors $P1$ and $P2$ during the first second and $P3$ during the next second will reduce the system latency; also the throughput will improve if the sequence is being repeated.

IV. Conclusion

We provided an efficient method to symbolically compute the start and end time of all actor firings of a dataflow graph. This information may be used to visualize dataflow execution traces to obtain a better understanding of the temporal behaviour of the system.

Acknowledgment

This research is supported by the ARTEMIS joint undertaking through the ALMARVI project (621439).

References

[1] Embedded Systems Innovation by TNO. website http://trace.esi.nl.

[2] G. Bilsen, M. Engels, et al. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44, 1996.

[3] M. Geilen and S. Stuijk. Worst-case performance analysis of synchronous dataflow scenarios. In *Proc. ISSS+CODES*, 2010.

[4] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *In Proc. IEEE*, 75, 1987.

[5] S. Stuijk, M. Geilen, et al. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Proc. SAMOS*, 2011.

[6] B. Theelen, M. Geilen, et al. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Proc. Memocode*, 2006.