RATAN: A Tool for Rate Analysis and Rate Constraint Debugging for Embedded Systems

Ali Dasdan
Dept. of Comp. Sci.
Univ. of Illinois
Urbana, IL 61801
dasdan@cs.uiuc.edu

Anmol Mathur
Silicon Graphics Inc.
2011 N. Shoreline Blvd.
Mountain View, CA 94043
anmol@mti.sgi.com

Rajesh K. Gupta
Dept. of Info. & Comp. Sci.
Univ. of California
Irvine, CA 92697
rgupta@ics.uci.edu

Abstract

The increasingly complex design of embedded systems creates the problems of specifying consistent and satisfiable rate constraints on process execution rates, checking them for consistency and satisfiability, computing process execution rates, and debugging rate constraint violations. The high complexity of these problems requires a complete and automated framework to help the designer in producing correct systems in shorter design time. We present such a framework and its implementation in a tool called `Ratan`*. Experiments on large benchmarks show the suitability of the tool for an interactive debugging environment.*

1 Introduction

Embedded systems typically consist of several interacting processes that operate in parallel and execute at different rates. These processes are implemented in hardware as well as software. To ensure correct timing behavior and achieve performance goals, designers often impose constraints on the execution rate of each process. These constraints are usually placed under an overall view and in an *ad hoc* manner. As the design goes along, it becomes more and more difficult for the system to conform to these constraints. Due to the diverse interactions of the processes in the system with one another and the system's environment, synthesizing even one process can create constraint violations for all the other processes. Designers may need to make many refinements as violations occur. However, spotting these violations and debugging them are tedious and difficult tasks due to the complexity and size of current designs, the number of refinements needed, and so on. For a correct design in shorter time, these tasks should be placed on a firm basis and automated. This paper presents the basis in a complete, interactive framework and its automation in a tool called `Ratan`. Our framework includes modules to check the consistency of the imposed rate constraints, compute the execution rate of each process, and compare the imposed and computed rates to check satisfiability. In case of any violations, it helps debug these violations interactively. Experimental results on large benchmark graphs show that the tool is quite fast for interactive work.

Previous works mostly focus on rate analysis in the context of performance analysis of asynchronous, concurrent systems modeled using timed Petri nets [2, 6, 10, 12]. Their analyses are based on restrictive assumptions, and most importantly, they lack a complete framework for rate constraint debugging in the form we present. [5] examines the problem by allowing only non-pipelined implementation and a very limited interaction between component processes. [13] discusses algorithms only to derive bounds on the execution time of a set of processes. The rate analysis part of our work is in part based on all these works. A more detailed review of the previous works is given in [11].

The rest of the paper is organized as follows. § 2 presents our process-based system model. § 3 formally defines the process execution rate and the rate analysis problem. § 4 discusses the complete rate analysis and rate constraint debugging framework in detail. § 5 reports the experimental results. Finally, conclusions and future plans are given in § 6.

2 Process Graph Model

Our model for an embedded system is a hierarchical, two-level model. Further developments of this model are given in [3, 4, 11]. The high level uses a process graph to represent the processes and their interactions, and the low level uses a sequencing graph to model each process.

ED&TC '97 on CD-ROM

The process graph $G_P = (V_P, E_P)$ is a weighted, directed graph where each node represents a process, and each arc represents synchronization and communication in a process pair. An arc from process p_i to process p_j with integer weight (delay) d_{ij} indicates that p_i will enable p_j after d_{ij} time steps using an *enable signal*. All processes are concurrently active, either running or waiting for an enable signal. They can start running independently for the first time; later, a process will start running if it is enabled by *all* its predecessors. Our model allows self-loops. A self-loop is used to realize pipelined or non-pipelined execution of a process, e.g., for pipelining, we restart an instance of a process before its previous one terminates by making the self-loop delay less than the process' execution time.

Sequencing graphs [8] are directed, acyclic graphs that capture both data and control dependencies in a hierarchical manner. Nodes represent operations and arcs represent sequencing dependencies among operations. Operations can be simple like an arithmetic operation or complex like a loop. These graphs support multiple threads of concurrent execution flow, which all belong to a single process. Each arc from a process node in the process graph actually originates from a node in the sequencing graph of that process. Such a node is called an *enabling node*. The delay on an arc in the process graph is the time taken by the operations on the path from the source node to an enabling node in the corresponding sequencing graph.

To be able to compute arc delays in the process graph, we assume that all the delays in the sequencing graphs are bounded. This assumption enables us to use previous bound estimation techniques like [9]. As pointed out in [9], determining exact execution time of a program fragment is very hard. Thus, the execution times are usually given in terms of lower and upper bounds. For rate analysis, we only need such bounds, so we use a delay interval as arc weights.

3 Rate Analysis Problem

Let $x_i(k)$ be the time at which process p_i starts its $(k+1)$th execution. The start time of p_i is given by $x(0)$. Since the predecessors of p_i enable p_i, $x_i(k)$ can be defined as

$$x_i(k) = \max_{p_j \in pred(p_i)} \{x_j(k-1) + d_{ij}\} \qquad (1)$$

where $k > 0$ and $pred(p_i)$ is the set of predecessors of p_i in the process graph. The *execution rate* $r(i)$ of a process p_i is the number of executions per unit time and is defined as

$$\begin{aligned} r(i) &= \left(\lim_{t \to \infty} \frac{\sum_{k=0}^{t-1} x_i(k+1) - x_i(k)}{t} \right)^{-1} \\ &= \left(\lim_{t \to \infty} \frac{x_i(t)}{t} \right)^{-1} \end{aligned} \qquad (2)$$

if the above limit exists. This limit is well-defined for any process graph regardless of the initial start times of the processes [11]. The limit in the above equation serves the purpose of finding the periodic behavior of the process graph. The *rate analysis problem* is the problem of finding lower and upper bounds on the execution rate of each process in a process graph.

The execution rates are closely related to the cycle delays in a process graph. The *delay of a cycle* is the sum of the delays of the arcs on it. The *mean delay* of a cycle is defined as the delay of the cycle divided by the number of arcs on the cycle. The mean delay gives the average weight of each arc on the cycle. The *maximum cycle mean* of a digraph is the maximum of the mean delays over all the cycles in that graph. A cycle is said to be *critical* if its mean delay is equal to the maximum cycle mean. The maximum cycle mean of a digraph can be efficiently computed using Karp's algorithm [1, 7]. We also use this algorithm.

4 Constraint Debugging Framework

We now present our framework that uses rate analysis to interactively debug constraint violations in a system. A flow diagram of this framework is given in Figure 1. We implemented this framework in a C++ program called *Ratan*. Sections below explain the numbered steps in the flow diagram.

4.1 Step 1: Input Process Graph

The input process graph has weights on its arcs as well as on its nodes. Arc weights are delay intervals for enabling signals, and node weights are rate constraint intervals imposed by the system requirements. Thus, each process p_i has a rate constraint interval $I_i = [L_i, U_i]$ where U_i (L_i) is the upper (lower) bound constraint such that we want to have $L_i \leq r(i) \leq U_i$. If there is no upper (lower) bound constraint, then $U_i = \infty$ ($L_i = 0$). In this step, we read in the process graph and the given rate constraints. We also compute the strongly connected components (components for short) of the process graph and build the component graph corresponding to these components. This step has a running time linear in the size of the process graph.

4.2 Steps 2–3: Consistency Check

We say that a set of rate constraints is *consistent* if they can simultaneously be satisfied for the given process graph irrespective of the delay intervals on its

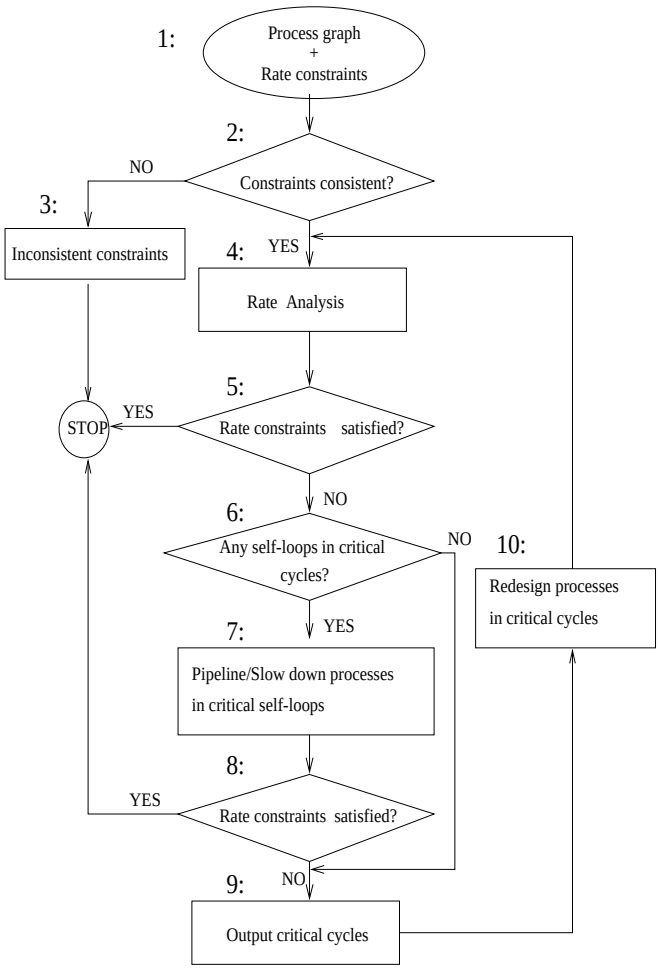

Figure 1: The framework for using rate analysis for debugging violations of rate constraints.

arcs. Thus, the consistency depends only on the topology of the process graph. If the rate constraints are inconsistent, the designer need to either modify the constraints or alter the design to change the process graph topology. We help the designer by reporting the cause of the inconsistency.

How can we check the consistency of the given set of rate intervals? Our algorithm runs in linear time and is based on the following result. Let C_j denote a component of the process graph. Define the intersection of C_j as $I_j^{\cap} = \bigcap_{p_i \in C_j} I_i$, and the minimum of C_j as $I_j^{\min} = \min(I_j^{\cap}, \min_{C_i \in pred(C_j)}\{I_i^{\min}\})$ if C_j has predecessors (denoted $pred(C_j)$) or $I_j^{\min} = I_j^{\cap}$ if it has no predecessors. (Note that the minimum of l intervals is $min(I_1, I_2, \cdots, I_l) = [\min_i\{L_i\}, \min_i\{U_i\}]$, the intersection of two intervals I_i and I_j is $I_i \bigcap I_j = [\max\{L_i, L_j\}, \min\{U_i, U_j\}]$, and $I_i < I_j$ if $U_i < L_j$.) Then, a set of given rate constraints is inconsistent if either there exists a component C_j of the process graph such that $I_j^{\cap} = \Phi$, or there exists a component C_j for which $I_j^{\min} < I_j^{\cap}$. Intuitively, the first condition holds since if the intersection of the rate constraints is empty, then it is impossible for a computed execution rate to fall in this intersection. The intuitive proof of the second condition is given in next section.

The algorithm first computes the $I_j^{\cap}$ for each component C_j. If this interval is empty for any of the components, the given rate constraints are inconsistent. If not, the algorithm topologically sorts the nodes of the component graph, and then traverses the nodes in topological order to compute $I_j^{\min}$ for each component. At the same time, the second condition is checked. If the check fails, the given constraints are again inconsistent. If they are found to be inconsistent in either case, we go to step 3 to exit by outputting the cause of the inconsistency and the component for which the inconsistency is found for the first time.

4.3 Step 4: Rate Analysis

Our algorithm used in this step computes the execution rate of every process, and is based on Karp's algorithm. We also keep enough information in the algorithm's data structures to determine critical cycles needed in later steps. This step runs in $O(|V_P||E_P|)$ for a process graph $G_P = (V_P, E_P)$, and is the most expensive step in terms of running time.

In case of only one component, we use Karp's algorithm to find the maximum cycle mean of the component. The execution rates of all the processes in a single component are the same, and they are all equal to the inverse of the maximum cycle mean of the component [10]. The process start times do not affect this mean [11]. One technicality here is how to handle the delay intervals, as Karp's algorithm assumes a single weight on each arc. Our strategy is to determine a rate interval in which the process execution rates are guaranteed to fall. By setting the delay of each arc to its upper bound, we compute the lower bound of the rate interval, as any delay value smaller than the upper delay bound can only decrease the maximum cycle mean or increase its inverse. Similarly, by setting the delay of each arc to its lower bound, we compute the upper bound of the rate interval.

In case of more than one component, we have to take care of the process interactions. Consider two components P and C (for Producer and Consumer) with rate intervals $[r_l(P), r_u(P)]$ and $[r'_l(C), r'_u(C)]$, respectively, where P is a predecessor of C in the component graph. Intuitively, if P is slower, C

has to wait for it because P enables it, or if C is slower, then C cannot execute faster just because P wants it to. Hence, the actual rate interval for C is $[r_l(C), r_u(C)]$ where $r_l(C) = \min\{r_l(P), r'_l(C)\}$ and $r_u(C) = \min\{r_u(P), r'_u(C)\}$. Similar reasoning also explains the second condition for the consistency check in the previous section. To compute the actual rate intervals for all the components, our algorithm traverses the components in topological order and updates their rate intervals using the result above. The rate intervals of components without any predecessors do not change.

4.4 Step 5: Satisfiability Check

Assuming that the given set of rate constraints is consistent, and that the process execution rates are computed, this step tries to verify if the given set of rate constraints can be satisfied. We say that a given rate constraint interval $I_i = [L_i, U_i]$ on a process p_i is *satisfied* if the computer rate interval $r(i)$ falls in I_i, or is *violated* if not. Note that the consistency of a given set of rate constraints does not imply their satisfiability but once they are inconsistent, they cannot be satisfied either.

Our algorithm in this step runs in linear time and works as follows. We compare the computed rate interval of each component C_j first with the intersection $I_j^{\cap}$ and then with the minimum $I_j^{\min}$. If the computed rate interval is not contained in $I_j^{\cap}$ or $I_j^{\min}$, then the given set is violated. In the former case, the violation is due to at least one process in C_j, and by going over each process in C_j, we output all the violations. In the latter case, we output the component causing the problem and expect the designer to correct the problem.

4.5 Steps 6–10: Processing Critical Cycles

As the critical cycles determine the execution rates, we should focus on the processes on critical cycles in case of a constraint violation. Our rate analysis algorithm can output a critical cycle for the designer to work on. (It is also possible to determine more than one critical cycle at the same time.) We can also determine if any of the self-loops is a critical cycle.

A redesign can involve changes in the process topology or processes in the output critical cycles. Assuming that the topology stays the same, the designer can do the following to eliminate some constraint violations. If there is an upper (lower) bound violation for a process, it indicates that the process is faster (slower) than required. The designer should slow down (speed up) the process. If the process is on a critical self-loop, we determine how much slow-down (speed-up) is required, and the designer can adjust the weight of the self-loop. If not, the designer should analyze the processes on critical cycles and redesign some or all of them accordingly. A slow-down can be achieved by introducing additional delay on some critical cycle arcs. Note that a single change in a process can affect many other processes, and it may be very difficult to predict these interactions precisely. That is why we need to go back and start the whole process again. Since our tool is very fast, many interactive reruns are possible in debugging a violation.

4.6 Extensions

The above analysis assumes that enable signals are issued unconditionally. However, if an enabling vertex is in a conditional branch, then it issues an enable signal conditionally, corresponding to a conditional arc. For such situations, we need to make two assumptions: (1) each branch of the conditional that contains the enabling node has a non-zero likelihood of being executed, i.e., the signal is really conditional, and (2) there is no arc, directly or indirectly, from the enabled process to the process issuing the signal, i.e., there is no deadlock due to this arc. Due to the second assumption, no critical cycle can contain a conditional arc; however, conditional arcs can affect the computed rate intervals of the processes. Since we are interested in guaranteed rate bounds, we can perform two rate analyses to find them, one with and another without the conditional arcs.

5 Experimental Results

Interactive rate constraint debugging requires a fast response time. To show the feasibility of our tool for use in an interactive debugging environment, we performed rate analysis and typical constraint violation scenarios on several large high-level synthesis benchmarks derived from several sources. We converted the benchmarks from `HardwareC` to sequencing graphs, and enclosed each sequencing graph operation in a separate process. This is done to simulate the worst-case HDL model, whereas a typical case would contain significantly less processes.

The benchmarks and the running time of the tool for each of them are given in Tab. 1. We did these experiments on a `Sun SPARCstation 20` with 64 MB real memory running under `SunOS R.5.4`. For each benchmark, the table lists its source, name, parts, size, and the running time. Small parts in a benchmark were selected for completeness, e.g., `noise` is a part of `ecc`. The running time is the sum of the total user and system CPU times and is in seconds. As can be seen from the table, the running times are very small even

for large instances, making the tool quite suitable for interactive debugging.

Table 1: Benchmarks and Running times (in seconds).

Source	Name	Part	#Nodes	#Arcs	Time
Olympus	cruiser	Calib	107	117	0.0
Olympus		Clock	32	38	0.0
Olympus		Display	60	66	0.0
Olympus		Getinfo	2082	2751	10.0
Olympus		State	301	353	1.0
Olympus		Temp	3	2	0.0
HLSynth92	diffeq		172	204	0.0
HLSynth91	ecc	decoder	145	162	0.0
HLSynth91		encoder	119	131	0.0
HLSynth91		noise	4	3	0.0
HLSynth92	gcd		49	55	0.0
HLSynth92	elliptic		307	344	0.0
HLSynth91	mc6502	group0	2219	2410	4.0
HLSynth91		group1	620	671	0.0
HLSynth91		group2	2928	3198	6.0
HLSynth91	parker86		172	177	0.0
HLSynth91	tseng		136	161	0.0

6 Conclusion

We have examined the problem of checking the consistency and satisfiability of rate constraints, the problem of computing process execution rates, and the problem of debugging constraint violations in an embedded system. Due to the high complexity of each problem, especially that of checking and debugging, and the need for correct, faster designs, a complete and automated framework is needed. We presented such a framework, and its implementation in a tool called `Ratan`. Our experiments on large high-level benchmarks show that the tool is considerably fast for an interactive debugging environment. Our future plans include refining the process model to include loops at the process graph level together with conditionals (§ 4.6), determining the "best" places for delays on critical cycles in case of upper bound violations (§ 4.5), and optimizing the tool even further.

Acknowledgments

This work is supported by the National Science Foundation under grants NSF Career Award MIP 95-01615 and ASC-96-34947.

References

[1] F. Bacelli, G. Cohen, G. J. Olsder, J.-P. Quadrat, *Synchronization and Linearity*, John Wiley and Sons, New York, 1992.

[2] S. M. Burns, A. J. Martin, *Performance Analysis and Optimization of Asynchronous Circuits,* Advanced Research in VLSI, Proc. Univ. of California/Santa Cruz Conf., pp. 71–86, 1991.

[3] D. Filo, D. C. Ku, C. Coelho and G. De Micheli, *Interface Optimization for Concurrent Systems under Timing Constraints,* IEEE Trans. VLSI Syst., vol. 1, no. 3, pp. 268–281, 1993.

[4] R. K. Gupta, *A Framework for Interactive Analysis of Timing Constraints in Embedded Systems,* Proc. Int. Workshop on Hardware/Software Codesign, 1996.

[5] R. K. Gupta, G. De Micheli, *Specification and Analysis of Timing Constraints for Embedded Systems*, Tech. Rep. UIUCDCS-R-96-1952, Univ. of Illinois, 1996.

[6] H. Hulgaard, S. M. Burns, T. Amon, G. Borriello, *An Algorithm for Exact Bounds on the Time Separation of Events in Concurrent Systems,* IEEE Trans. Comput., vol. 44, no. 11, pp. 1306–1317, 1995.

[7] R. M. Karp, *A characterization of the minimum cycle mean in a digraph,* Discrete Math., vol. 23, pp. 309–311, 1978.

[8] D. C. Ku, G. De Micheli, *High Level Synthesis of ASICs Under Timing and Synchronization Constraints,* Kluwer Academic Publishers, Boston, 1992.

[9] Y. -T. S. Li, S. Malik, *Performance Analysis of Embedded Software Using Implicit Path Enumeration,* Proc. Design Automation Conf., 1995.

[10] J. Magott, *Performance Evaluation of Concurrent Systems Using Petri Nets,* Information Processing Letters, vol. 18, pp. 7–13, 1984.

[11] A. Mathur, A. Dasdan, R. K. Gupta, *Rate Analysis for Embedded Systems,* Tech. Rep. UIUCDCS-R-96-1954, Univ. of Illinois, 1995.

[12] C. V. Ramamoorthy, G. S. Ho, *Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets,* IEEE Trans. Software Eng., vol. 6, no. 5, pp. 440–449, 1980.

[13] T.-Y. Yen, Wayne Wolf, *Performance Estimation for Real-Time Distributed Embedded Systems,* Proc. Int. Conf. Computer Design, pp. 64–69, 1995.