Analysis of Operation Delay and Execution Rate Constraints for Embedded Systems

Rajesh K. Gupta

Department of Computer Science
University of Illinois, Urbana-Champaign
1304 W. Springfield Avenue, Urbana, Illinois 61801.

Abstract

Constraints on the delay and execution rate of operations in an embedded application are needed to ensure its timely interaction with a reactive environment. In this paper, we present a static analysis of the timing constraints satisfiability by a given system design consisting of interacting hardware and software components. We use this analysis to evaluate the effect of individual timing constraints on system design issues, such as the choice of the software runtime system, bounds on loop invocations, and the hardware-software synchronization operations. We show, by example, the use of static analysis techniques in the design of embedded systems.

1 Introduction

Estimation and analysis of timing performance of individual hardware and software components is crucial to efficient design of embedded systems [1, 2]. *Constraints* on system performance are often used to determine the division of system functionality into hardware and software components [3, 4]. In particular, we consider timing constraints on language-level operations, such as read/write operations. These constraints may be on the relative timing of two operations, or on the rate of execution of an operation. We present a case analysis of timing constraint satisfiability, and identify cases where deterministic answers to constraint satisfiability are possible even in the presence of uncertainty in the system model.

We begin with a description of the desired application in a hardware description language (HDL). Since most HDLs use static data types and only un-aliased data references, it is possible to do a static determination of operation dependencies and memory references. Both of these features are essential for analysis of constraints on timing and size of implementation, and hence our choice of HDL for input specification. In this specification, an application is described as a collection of processes that execute concurrently. All communication between operations in a process body is based on shared storage, declared as a part of the process body. Inter-process communication is specified by message-passing operations that use a blocking protocol for synchronization purposes.

The input description is compiled into a graph-based model consisting of *flow graphs*. A flow graph is a polar acyclic graph where the vertex set represent language-level operations and the edge set represents dependencies between operation vertices. A vertex in the flow graph represents one of the following operations: *nop, conditional, logic, arithmetic, io, wait* and *link*. The *wait* operation represents synchronization events at system inputs and outputs.

The *link* operation captures hierarchy of models by representing a call or a loop operation. The called flow graph corresponding to a link vertex may be invoked one or many times. A loop link operation consists of a loop condition operation that performs testing of the loop exit condition and a loop body. The number of invocations of a loop body are controlled by a *loop index* variable associated with each loop operation.

The successors to a conditional operations are enabled depending upon the outcome of the condition evaluation. Thus the flow graph is a bilogic control graph [5], where a fork represents either a concurrent or disjoint set of operations.

An *implementation* of a graph G refers to assignment of delay and size properties to operations in G, and a choice of *runtime scheduler* that repeatedly enables execution of the source operation in G. This assignment of values is related to the hardware or software implementation of operations in G. We assume that the expressed concurrency in flow graph models can be supported by available hardware resources. That is, any serialization required to meet hardware resource constraints has already been performed. This assumption is made in view of the primarily control-dominated target applications for which our hardware synthesis methodology completes resource binding before any operation scheduling is done.

The *delay*, δ, of an operation refers to the execution delay of the operation. We assume that for a graph model, the delay of all operations is expressed in cycles for a given cycle time associated with the graph model. The *latency*, $\lambda(G)$, refers to the execution delay of the graph model G. The latency of a flow graph may be variable due to the presence of conditional paths and synchronization operations. An execution delay is associated with link vertices as the latency of the corresponding graph model times the number of times the called graph is invoked. Since the latency can be variable, therefore, the delay of a link vertex can be variable. In (bi-

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA

logic) flow graphs, link and wait operation vertices introduce uncertainty over the precise delay and order of operations in the system model. Due to this uncertainty, these operations are called *non-deterministic delay* or ND operations.

We define a lower bound on latency as the **length** of the longest path between the source and sink vertices assuming the loop index to be one for the loop operations. In presence of conditional paths, the length is a vector, $\underline{\ell} = (\ell[i])$ where each element $\ell[i]$ indicates the execution delay of a path in G. The elements of $\underline{\ell}$ are the lengths of the longest paths that are mutually-exclusive. Notationally, ℓ_m and ℓ_M refer to the minimum and maximum element in $\underline{\ell}$ respectively. Note that static characterization of conditional paths by $\underline{\ell}$ may also include some infeasible execution paths. This estimation can be improved by additional user input as in [6, 1]. We define the rate of execution, $\rho_i(k)$, at invocation k of an operation v_i as the inverse of the time interval between its current and previous execution. By convention, the instantaneous rate of execution is 0 at the first execution of an operation.

2 Timing Constraints

Maximum and minimum operation delay constraints bound the time interval between initiation of two given operations. A minimum or maximum execution rate constraint bounds the rate of execution of an operation similarly. A rate constraint on an operation, v_i, may also be *relative* to execution of a graph model, G. This bounds the rate of execution of v_i over time periods when G is continuously enabled. The relative rate of execution expresses rate constraints that are applicable to a specific *context* of execution as captured by the control flow in G.

We specify timing constraints by tagging statements in the HDL description. Relative rate constraints on an operation are applied only relative to the flow graphs in the hierarchy in which the operation resides. The operation hierarchy is indexed with the flow-graph in the inner-most hierarchy as index 0. In Example 2.1 below, there are two relative rate constraints on the *read* operation relative to the two *while* statements.

Example 2.1. Specification of rate constraints.

```
process example (frameEN, bitEN, bit, word)
   in port frameEN, bitEN, bit;
   out port word[8];
{
   boolean store[8], temp;
   tag A;

   while (frameEN)
   {
      while (bitEN)
      {
A:       temp = read(bit);
         store[7:0] = store[6:0] @ temp;
      }
      write word = store;
   }
   attribute "constraint minrate of A = 100 cps";
   attribute "constraint minrate 0 of A = 1 cps";
   attribute "constraint minrate 1 of A = 10 cps";
}
```

In this example, a minimum rate constraint of 100 cycles per sample execution, or 0.01 executions per cycle, is specified on the read operation. In addition, two *relative* minimum rate constraints of 1 and 0.1 per cycle are specified for the read operation relative to the loops *while(bitEN)* and *while(frameEN)* respectively.

Operation delay constraints are specified similarly using tags using the syntax below:

```
   .constraint mintime from <tag1> to <tag2> =
                  <num> cycles;
   .constraint maxtime from <tag1> to <tag2> =
                  <num> cycles;
 .constraint finish|before|during <tag1> <tag2>;
```

□

2.1 Constraint Satisfiability

Given a schedule of operation execution times, a timing constraint is considered *satisfied* if the operation initiation times (determined by the scheduling function) satisfy the corresponding bounds on the time intervals. Clearly, the satisfaction of timing constraints is related to the choice of the scheduling function. In general, the choice of a particular operation scheduler depends upon the types of operations supported and the resulting control hardware or software required to implement the scheduler. For constraint analysis purposes, it is not necessary to determine a specific schedule of operations, but only to verify the existence of a schedule. We present constraint satisfiability tests in the context of a multi-threaded software implementation [7] and a bilogic relative schedule for hardware portions [8].

The satisfiability tests are based on a timing constraint graph, $G_T = (V, E, \Delta)$ where the set of edges consists of *forward* and *backward* edges, $E = E_f \cup E_b$ and $\delta_{ij} \in \Delta$ defines the weights on edges such that $t_k(v_i) + \delta_{ij} \leq t_k(v_j)$ for all $k > 0$. The following results form the basis of operation-level timing constraint analysis.

• A constraint graph is considered *feasible* if it contains no positive cycle when the delay of ND operations is assigned to zero. Minimum and maximum operation delay constraints are satisfiable if and only if the constraint graph is feasible and there exist no cycles with ND operations [9].

• A maximum rate constraints is always satisfiable since it leads to a lower bound on the static path lengths. This is checked by comparing the bound against ℓ_m.

ℓ_m also defines the fastest rate at which an operation in the graph model can be executed by a non-pipelined implementation. This points to the necessary condition for meeting a minimum execution rate constraint. A sufficient condition for meeting the minimum rate constraints is obtained by placing a bound on the interval from the completion of operations in G to the start of the source operation in G. We call this interval, $\gamma(G)$, as the *overhead* on the repeated executions of G. A minimum rate constraint, r_i, on operation, $v_i \in V(G)$ is satisfiable if $\overline{\gamma}(G) + \ell_M(G) \leq \frac{\tau}{r_i}$.

A bound $\overline{\gamma}$ on the overhead delay $\gamma(G)$ implies a bound on the invocation interval of G_+, the calling graph (also called the parent graph) of G. Therefore, this bound on the overhead term can be expressed recursively over the flow graph hierarchy as follows:

$$\overline{\gamma}(G) = [\ell_M(G_+) + \overline{\gamma}(G_+)] - \ell_m(G) \qquad (1)$$

In particular, this bounds the invocation interval of the parent process graph G_0. The invocation interval of a process graph refers to the delay due to the runtime scheduler. Thus a overhead bound restricts the choice of the runtime scheduler. *Note that a bound on $\gamma(G)$ does not imply a bound on the* latency λ *of G which may, in fact, be unbounded.*

However, in the presence of ND operations in G, deterministic satisfiability of (an absolute) minimum rate constraint can only be guaranteed by transforming the ND operations into bounded-delay operations. Justification of a bound on the ND operation comes from the observation that constraint satisfiability is a property of an implementation (and not of a specification), so while it is always possible for a reactive environment to overrun any specified time bound between its actions, however, it does not affect the satisfiability of an implementation.

In practical terms, this means constraint satisfiability is answered assuming conditionals paths as taken, and by considering bounds on the loop indices. The delay of *wait* operation depends upon its implementation. For instance, in a *busy-wait* implementation (i.e., *while(!signal);*), the wait operation is implemented as a loop operation that iterates until the concerned input (*signal*) is received. This implementation is commonly used for hardware synthesis [9]. In another implementation, the execution of wait operation causes a *context-switch*. This implementation is particularly applicable for software implementations [7]. For this implementation, the delay of the wait operation is characterized by a fixed overhead as the delay due to the runtime system, and hence treated as a non-ND operation.

3 Implementation

Operation-level constraint analysis is implemented as a part of the co-synthesis framework, VULCAN [10], to allow the system designer to explore hardware versus software implementations of a given system model. The operation delays corresponding to hardware implementation are obtained using the high-level synthesis tools [9], whereas software implementation is considered in the context of a specific processor cost model specified by the designer. To evaluate the effect of the runtime system, we have explored the following three ways to implement the software routines: (a) subroutine-based, (b) coroutine-based and (c) description-by-cases. Very briefly, a subroutine implementation refers to translation of program threads into program subroutines that operate under a global task scheduler. In contrast, a coroutine implementation reduces the overhead by placing routines in a co-operative, rather than hierarchical, relationship to each other. The coroutines maintain a local state and willingly relinquish control of the processor at exception conditions which may be caused by unavailability of data (for example, a data dependency on another thread) or an interrupt. In case of such exceptions the coroutine switch picks up the processes according to a predefined priority list. Upon resumption a coroutine execution starts execution from the position where its was detached last. A restricted coroutine implementation reduces the overhead further by suitably partitioning the on-chip register storage between program routines such that program counter is the only register that is saved/restored during an inter-routine transfer. Finally, in the description-by-cases, we merge different routines and describe all operations in a single routine. This scheme is simpler than the coroutine scheme. Here we construct a single program which has a unique state assignment for each synchronization operation. A global state register stores the state of execution of a thread. Transitions between states are determined by the runtime scheduling of different ND operations based on the data received. This method is restrictive since it precludes use of nested routines and requires description as a single switch statement, which in cases of particularly large software descriptions, may be too cumbersome.

Implementation	*Processor*	*Overhead cycles*
Subroutine	'86	728
Coroutine	'86	364
Restricted Coroutine	'86	103
Description by cases	'86	85
Restricted Coroutine	DLX	19
Description by cases	DLX	35

Table 1: Runtime overhead in cycles.

Table 1 summarizes program overhead for different implementation schemes. Results are reported for two processors, DLX [11] and the Intel 8086. Overhead cycles refers to the overhead (in cycles) incurred due each transfer operation from one program thread to another. It is clear that a restricted coroutine implementation gives the best result, or the least runtime overhead. In case of the x86 processor, the case description scheme reduces the overhead by reducing amount of ALU operations in favor of a slight increase in memory input-output operations. This scheme entails smaller overheads when compared to the general coroutine scheme.

The results of constraint satisfiability tests are put together in the procedure *check_satisfiability* shown below. The input to the procedure is a set of graph models with delay and rate constraints along with a choice of the runtime system. Its output is null if the constraints are satisfiable, else either G is unsatisfiable or it returns bounds on the delay of ND operations that would make constraints satisfiable. These bounds can then be verified by the system designer as being applicable, or requiring system redesign.

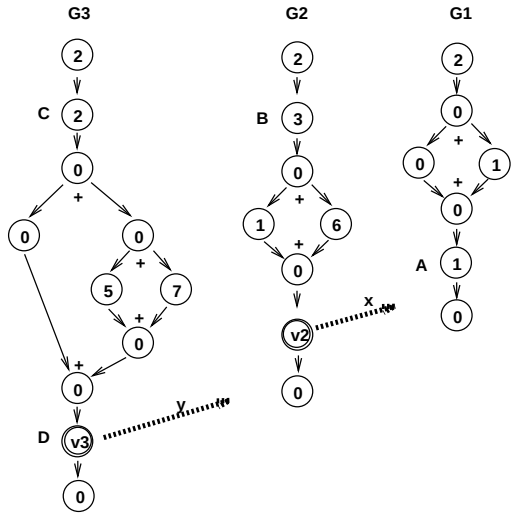
---

check_satisfiability(G)
  for $v \in V(G)$
    if $v =$ loop
      check_satisfiability(G_v);
  construct G_T
  for each backward edge u in G_T
    if (cycle-set = find-cycles(G_T))
      for $\Gamma \in$ cycle-set
        if $(\ell_M(\Gamma) > 0)$
          return (G is unsatisfiable);
        for $v \in \Gamma$ and $v \in ND$
          print $\delta_v = u - \ell_M(\Gamma)$;
          bound delay of $v = \delta_v$;
          mark v as non-ND;
  $s = \lceil \ell_m(G) - \tau \cdot \max_i R_i^{-1} \rceil$
  if $s \geq 0$
    return (G is satisfied);
  else
    add NOP with $\delta = s$;
    update $\underline{\ell}(G_o)$;
    check_satisfiability(G);
  if G_+ exists
    add constraint $[\frac{\tau}{r_i} - \ell_M(G) + \ell_m(G)]^{-1}$
      on link operation in G_+;

The following illustrates an example where the satisfiability tests successfully returns with bounds on ND operation

delays.

Example 3.1. Consider the hierarchy of flow graphs below with the following constraints: $r_A = 1/100, r_A^{G_1} = 1/6, r_A^{G_2} = 1/40, r_B = 1/50, r_B^{G_2} = 1/30, r_C = 1/200, u_{CD} = 12$ and a bound on the runtime overhead $\overline{\gamma}_0 = 20$. Here $r_A^{G_1}$ refers to a minimum rate constraint on operation A relative to G_1.



The procedure first considers G_1. The corresponding constraint graph G_{T1} has three backward edges with following weights:

$$
\begin{aligned}
r_A^{G_1} = 1/6 \quad &\Rightarrow \quad -6 \\
r_A^{G_2} = 1/40 \quad &\Rightarrow \quad -[40 - \gamma(G_1)]_{\gamma(G_2)=0} \\
&= \quad -[40 - \{\gamma(G_2) + \ell_M(G_2) - \ell_m(G_1)\}]_{\gamma(G_2)=0} \\
&= \quad -[40 - 0 - 15 + 3] = -28 \\
r_A = 1/100 \quad &\Rightarrow \quad -[100 - \gamma(G_1)] = -(100 - [\gamma(G_2)] - 15 + 3) \\
&= \quad -(88 - [\ell_M(G_3) + \overline{\gamma}_0 - \ell_m(G_2)]) \\
&= \quad -(88 - [26 + 20 - 9]) = -51
\end{aligned}
$$

The maximum forward path length is 4 (< 6), G_{T1} contains no ND-cycles.

These constraints are propagated to G_2 as follows: $r_A^{G_1}$ is relative to G_1 and, therefore, it is not propagated to G_2. $r_A^{G_2}$ is propagated as a constraint $r_{v_2}^{G_2} = 1/(28-1) = 1/27$ on link operation v_2. Finally, r_A is propagated as $r_{v_2} = 1/(51-1) = 1/50$.

In the next iteration the analysis is done on G_2. G_{T2} has four backward edges with weights $(-50 - \overline{\gamma}(G_2)) = -13$, -27, -50 and -30. Constraint $r_B^{G_2}$ bounds the delay due to the ND operation to 16 cycles.

Finally for G_3, there are three backward edges in the constraint graph with weights -12, $-(200 - \overline{\gamma}(G_3)) = -180$ and -44. Constraint r_C bounds the delay of v_3 to 165 cycles. □

4 Conclusions

We have presented delay and rate constraints on operations in a system model. A notion of constraint satisfiability is developed based on the ability to determine the existence of a schedule of operations that meets the constraints. A run-time scheduler models the uncertainty caused by ND operations in the invocation of graph models. The satisfaction of the bounds on delay of ND operations requires additional information from their implementations (such as context switch delay, possible loop index values) against which the questions about satisfiability of minimum rate constraint are answered. An important implication of having bounds derived from timing constraints is that it makes it possible to seek transformations to the system model which tradeoff these measures of constraint satisfiability against implementation costs. Under certain conditions, these bounds can be extended by modifying the structure of the flow graphs with ND cycles [10].

The operation-level analysis forms a part of the system performance analysis. Our future plans are to develop a comprehensive framework for system analysis by incorporating both runtime schedulability and process-level rate analysis for pipelined implementations.

5 Acknowledgments

This research was supported by NSF CAREER Award MIP 95-01615, and a grant from NSF Engineering Research Center 89-43166.

References

[1] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proceedings of the Design Automation Conference*, June 1995.

[2] W. Ye, R. Ernst, T. Benner, and J. Henkel, "Fast timing analysis for hardware-software co-synthesis," in *Proceedings of the International Conference on Computer Design*, 1993.

[3] R. K. Gupta and G. D. Micheli, "Hardware-Software Cosynthesis for Digital Systems," *IEEE Design & Test of Computers*, pp. 29–41, Sept. 1993.

[4] R. Ernst, J. Henkel, and T. Benner, "Hardware-Software Cosynthesis for Microcontrollers," *IEEE Design & Test of Computers*, pp. 64–75, Dec. 1993.

[5] V. Cerf, *Multiprocessors, Semaphores and a Graph Model of Computation.* PhD thesis, UCLA, Apr. 1972.

[6] C. Y. Park, "Predicting program execution times by analyzing static and dynamic program paths," *Real-Time Systems*, vol. 5, no. 1, pp. 31–62, Mar. 1993.

[7] R. K. Gupta, C. Coelho, and G. D. Micheli, "Program Implementation Schemes for Hardware-Software Systems," *IEEE Computer*, Jan. 1994.

[8] R. K. Gupta and G. D. Micheli, "Specification and Analysis of Timing Constraints for Embedded Systems," Tech. Report DCS-UIUC-1995, University of Illinois, 1995.

[9] D. Ku and G. D. Micheli, *High-level Synthesis of ASICs under Timing and and Synchronization Constraints.* Kluwer Academic Publishers, 1992.

[10] R. K. Gupta and G. D. Micheli, "A Co-Synthesis Approach to Embedded System Design Automation," *Design Automation for Embedded Systems*, vol. 1, no. 1-2, Jan. 1996.

[11] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan-Kaufman, 1990.