Performance Estimation for Embedded Systems with Data and Control Dependencies

Paul Pop, Petru Eles, Zebo Peng
Dept. of Computer and Information Science, Linköping University, S-58183 Linköping, Sweden
{paupo, petel, zebpe}@ida.liu.se

ABSTRACT

In this paper we present an approach to performance estimation for hard real-time systems. We consider architectures consisting of multiple processors. The scheduling policy is based on a preemptive strategy with static priorities. Our model of the system captures both data and control dependencies, and the analysis is able to reduce the pessimism of the estimation by using the knowledge about these dependencies. Extensive experiments as well as a real life example demonstrate the efficiency of our approach.

1. INTRODUCTION

In this paper we present an approach to performance estimation for hard real-time systems that have both data and control dependencies. We consider applications that are implemented on distributed architectures and, in our approach, the system is modeled by a so called *conditional process graph* (CPG) [3]. Such a graph captures both the flow of data and that of control. Processes are scheduled using a priority based preemptive policy.

Process scheduling for performance estimation and synthesis of real-time systems has been intensively researched in the last years. Static non-preemptive scheduling of a set of processes on a multiprocessor system has been discussed in [3, 6, 9]. In [7] an *earlier deadline first* strategy is used for non-preemptive scheduling of processes with possible data dependencies. Preemptive scheduling of independent processes with static priorities running on single processor architectures has its roots in [8]. The approach has been later extended to accommodate more general system models and has been also applied to distributed systems [12]. An algorithm for optimal priority assignment to processes is proposed in [1]. In [11] and [13] *time offset* relationships and *phases*, respectively, are used in order to model data dependencies in the context of priority based preemptive scheduling.

When control dependencies exist, depending on conditions, only a subset of the set of processes is executed during an invocation of the system. *Modes* have been used to model a certain class of control dependencies [4]. Such a model basically assumes that at the starting of an execution cycle, a particular functionality is known in advance and is fixed for one or several cycles until another mode change is performed. However, modes cannot handle fine grained control dependencies or certain combinations of data and control dependencies. Careful modeling using the *periods* of processes (lower bound between subsequent re-arrivals of a process) is a possible solution for some cases of control dependencies [5]. If, for example, we know that a certain set of processes will only execute every second cycle of the system, we can set their periods to the double of the period of the rest of the processes in the system. However, using the worst case assumption on periods leads very often to unnecessarily pessimistic solutions. A more refined process model can produce much better results, as will be shown later.

CODES 2000 San Diego CAUSA

We propose in the next section a system model based on a conditional process graph that is able to capture both data and control dependencies. Then, we introduce a less pessimistic analysis technique in order to bound the response time of a hard real-time system modeled in such a way. In this paper we insist on various aspects concerning dependencies between processes in the context of priority based preemptive scheduling. Static cyclic scheduling of processes with both data and control dependencies has been addressed by us in [2, 3]. We have also discussed the particular aspects concerning scheduling and communication synthesis for distributed systems in [9, 10].

This paper is divided into 7 sections. The next section presents our graph-based system representation. Section 3 formulates the problem and sections 4 and 5 present the proposed performance estimation approaches. The techniques are evaluated in section 6, and section 7 presents our conclusions.

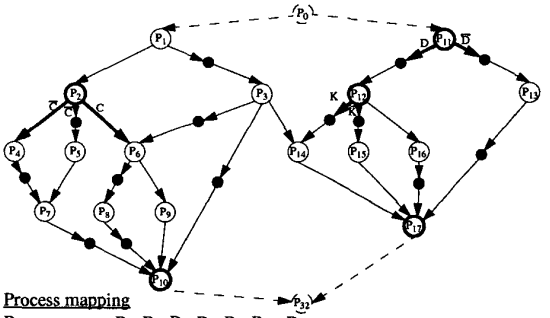
2. CONDITIONAL PROCESS GRAPH

As an abstract model for system representation we use a directed, acyclic, polar graph $\Gamma(V, E_S, E_C)$ [3]. Each node $P_i \in V$ represents one process. Such a process can be an ordinary process specified by the designer or a so called *communication process* which captures the message passing activity. E_S and E_C are the sets of simple and conditional edges respectively. $E_S \cap E_C = \emptyset$ and $E_S \cup E_C = E$, where E is the set of all edges. An edge $e_{ij} \in E$ from P_i to P_j indicates that the output of P_i is the input of P_j. The graph is polar, which means that there are two nodes, called *source* and *sink*, that conventionally represent the first and last process. These nodes are introduced as dummy processes so that all other nodes in the graph are successors of the source and predecessors of the sink respectively.

We consider a distributed architecture consisting of several *processors* connected through *buses*. These buses can be shared by several communication channels connecting processes assigned to different processors.

We assume that each process is assigned to a processor and each communication channel which connects processes assigned to different processors is assigned to a bus.

Each process P_i (ordinary or communication process), assigned to a processor or bus, is characterized by a worst case execution



Process mapping
Processor pe_1: P_1, P_2, P_4, P_6, P_9, P_{10}, P_{13}
Processor pe_2: P_3, P_5, P_7, P_{11}, P_{14}, P_{15}, P_{17}
Processor pe_3: P_8, P_{12}, P_{16}
Communications are mapped to a unique bus

Figure 1. Conditional Process Graph

time C_i. In the process graph depicted in Figure 1, P_0 and P_{32} are the source and sink nodes respectively. Nodes denoted P_1, P_2, .., P_{17}, are ordinary processes specified by the designer. Figure 1 also shows the mapping of processes to three different processors. The communication processes are represented in Figure 1 as solid circles and are introduced for each connection which links processes mapped to different processors. In this paper we do not consider the communication aspects which we have analyzed in [9, 10].

An edge $e_{ij} \in E_C$ is a *conditional edge* (thick lines in Figure 1) and it has an associated condition. Transmission on such an edge takes place only if the associated condition is satisfied. We call a node with conditional edges at its output a *disjunction process*. Alternative paths starting from a disjunction process, which correspond to complementary values of a certain condition, are disjoint and they meet in a so called *conjunction process. Conditions are dynamically computed by disjunction processes and their value is unpredictable at the start of an execution cycle of the conditional process graph.* In Figure 1 circles representing conjunction and disjunction processes are depicted with thick borders. We assume that conditions are independent.

A process, which is not a conjunction process, can be activated only after all its inputs have arrived. A conjunction process can be activated after messages coming on one of the alternative paths have arrived. All processes issue their outputs when they terminate. If we consider the activation time of the source process as a reference, the finishing time of the sink process is the delay of the system at a certain execution.

3. PROBLEM FORMULATION

An application is modeled as a set ψ of n conditional process graphs Γ_i, $i = 1..n$. Every process P_i in such a graph is mapped to a certain processor, has a known worst-case execution time C_i, a deadline D_i, and a uniquely assigned priority. All processes belonging to the same CPG Γ_i have the same period $T_{\Gamma i}$ which is the period of the respective conditional process graph. Typically, global deadlines $\delta_{\Gamma i}$ on the delay of each CPG are imposed rather than individual deadlines on processes.

We consider a priority based preemptive execution environment. We are interested to derive worst case delays for each CPG in a given system ψ. The approach can be easily extended if delays on individual processes are of interest.

To show the relevance of our problem, let us consider the example depicted in Figure 2, where we have a system modeled as two conditional process graphs Γ_1 and Γ_2 with a total of 9 processes (the four dummy processes are not counted), and one condition. The processes are mapped on different processors as indicated by the shading, and the worst case execution time, in milliseconds, for each process on its respective processor is depicted to the left of each node. Γ_1 has a period of 200 ms, Γ_2 has a period of 150 ms. The deadlines are 100 ms on Γ_1 and 90 ms on Γ_2.

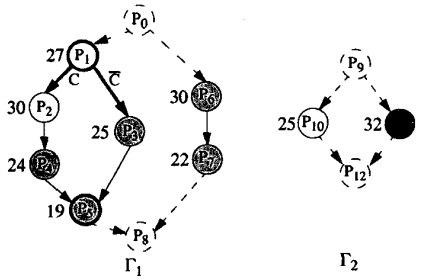


Figure 2. System with Control and Data Dependencies

When the analysis is applied to the set of processes, ignoring control dependencies, we get an estimated worst case delay of 120 ms for Γ_1 and 82 ms for Γ_2. This analysis assumes as a worst case scenario the possible activation of all nine processes for each execution of the system. This is the solution which will be obtained using a dataflow graph representation of the system. However, considering the CPG Γ_1 in Figure 2, it is easy to observe that process P_3 on the one hand and processes P_2 and P_4 on the other hand will not be activated during the same period of Γ_1. Making use of this information for the analysis we obtain a worst case delay of 100 ms for Γ_1, which indicates that the system is schedulable.

4. DELAY ESTIMATION FOR TASK GRAPHS WITH DATA DEPENDENCIES

Methods for schedulability analysis of data dependent processes with static priority preemptive scheduling have been proposed in [11] and [13]. They use the concept of *offset* or *phase*, respectively, in order to handle data dependencies. [13] provides a framework that iteratively finds the phases for all processes, and then feeds them back into the response time analysis which in turn is used again to derive better phases. Thus, the pessimism of the analysis is iteratively reduced.

We have used the framework provided by [13] as a starting point for our analysis. The response time of a process P_i is:

$$r_i = C_i + \sum_{\forall j \in hp(P_i)} C_j \left\lceil \frac{r_i - O_{ij}}{T_j} \right\rceil \quad (1)$$

where $hp(P_i)$ is the set of processes that have higher priority than P_i, and O_{ij} is the phase of P_j relative to P_i.

In [13] a system is modeled as a set S of n *task graphs* G_i, $i = 1..n$. The system model assumed and the definition of a task graph are similar to our CPG, but without considering any conditions. The aim of the analysis is to derive an as tight as possible worst case delay on the execution time of each of the task graphs in the system. This delay estimation is done using the algorithm DelayEstimate described in Figure 3. The function LatestTimes calculates worst case response times of processes and upper bounds for the offsets, while EarliestTimes derives the lower bounds of the offsets.

During a topological traversal of the graph G within LatestTimes, for each process P_i the worst case response time r_i is calculated according to equation (1). This value is based on the values of the offsets known so far. Once an r_i is calculated, it can be used to determine and update offsets for other successor processes. Accordingly, the EarliestTimes function determines the lower bounds on the offsets. The influence on graph G from other graphs in the system is considered in both of the functions mentioned earlier.

These calculations can be improved by realizing that for a process P_i, there might exist a process P_j mapped on the same processor, with priority(P_i) < priority(P_j), such that their execution windows never overlap. In this case, the term in the equation (1) that expresses the influence of P_j on the execution of P_i can be

```
DelayEstimate(task graph G, system S)
-- derives the worst case delay of a task graph G considering
-- the influence from all other task graphs in the system S
   for each pair (Pi, Pj) in G
      maxsep[Pi, Pj] = ∞
   end for
   step = 0
   repeat
      LatestTimes(G)
      EarliestTimes(G)
      for each Pi ∈ G
         MaxSeparations(Pi)
      end for
   until maxsep is not changed or step > limit
   return the worst case delay δG of the graph G
end DelayEstimate
```

Figure 3. Delay Estimation for Task Graphs

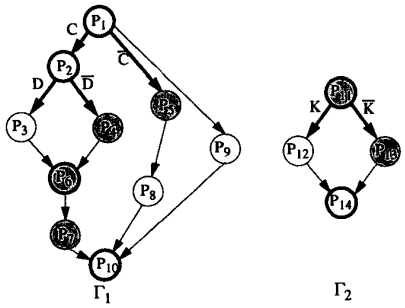


Figure 4. Example of two CPGs

dropped, resulting in a tighter worst case response time calculation. This situation is expressed through the so called *maxsep* table, computed by the MaxSeparations function, whose value *maxsep*$[P_i, P_j]$ is less than or equal to 0 if the two processes never overlap during their execution. The *maxsep* table is built using the worst case execution times and offsets determined in EarliestTimes and LatestTimes.

Having a better view on the maximum separation between each pair of processes, tighter worst case response times and offsets can be derived, which in turn contribute to the update of the *maxsep* table. This iterative tightening process is repeated until there is no modification to the *maxsep* table, or a certain imposed *limit* on the number of iterations is reached.

Finally, the DelayEstimate function returns the worst-case delay δ_G estimated for a task graph G, as the latest time when the sink node of G can finish its execution.

5. DELAY ESTIMATION FOR CPGs

Depending on the values calculated for the conditions, different alternative tracks through a conditional process graph are activated for a given activation of the system. To model this, a boolean expression X_{Pi}, called *guard*, can be associated to each node P_i in the graph. It represents the necessary condition for the respective process to be activated. In Figure 4, for example, $X_{P4}=C\wedge\overline{D}$, $X_{P5}=\overline{C}$, $X_{P9}=true$, $X_{P11}=true$, and $X_{P12}=K$.

We call an alternative track through a conditional process graph, resulting from a combination of conditions, an *unconditional subgraph*, denoted by g. For example, the CPG Γ_1 in Figure 4 has three unconditional subgraphs, corresponding to the following combinations of conditions: $C\wedge D$, $C\wedge\overline{D}$, and $\overline{C}$. The unconditional subgraph corresponding to the combination $C\wedge\overline{D}$ in the CPG Γ_1 consists of processes P_1, P_2, P_4, P_6, P_7, P_9 and P_{10}.

5.1 Ignoring Conditions (IC)

A straightforward approach to delay estimation for systems represented as CPGs is to ignore control dependencies and to apply the analysis as described in section 4.

This means that conditional edges in the CPGs are considered like simple edges and the conditions in the model are dropped. What results is a system S consisting of simple task graphs G_i, each one resulted from a CPG Γ_i of the given system ψ. The system S can then be analyzed using the algorithm in Figure 5.

```
DE/IC(system ψ)
-- derives worst case delays for each CPG in the system ψ
   transform each Γi ∈ ψ into the corresponding Gi ∈ S
   for each task graph Gi ∈ S
      DelayEstimate(Gi, S)
   end for
end DE/IC
```

Figure 5. Delay Estimation Ignoring Conditions

```
DE/CPG(CPG Γ, system S)
-- derives the worst case delay of a CPG Γ considering
-- the influence from all other task graphs in the system S
   extract all unconditional subgraphs gj from Γ
   for each gj
      DelayEstimate(gj, S)
   end for
   return the largest of the delays, which is
     the worst case delay δΓ of CPG Γ
end DE/CPG
```

a) DE/CPG -- Delay Estimate for Conditional Process Graphs

```
DE/BF(system ψ)
-- derives worst case delays for each CPG in the system ψ
   transform each Γi ∈ ψ into the corresponding Gi ∈ S
   for each Γi ∈ ψ
      DE/CPG(Γi, {G1, G2, ...Gi-1, Gi+1, Gn})
   end for
end DE/BF
```

b) DE/BF -- Delay Estimation: the Brute Force approach

Figure 6. Brute Force Analysis

This approach, which we call IC, is, of course, very pessimistic. However, this is the current practice when worst case arrival periods are considered and classical data flow graphs are used for modeling and scheduling.

5.2 Brute Force Solution (BF)

The pessimism of the previous approach can be reduced by using a conditional process graph model. A simple, brute force solution is to apply the analysis presented in section 4, after the CPGs have been decomposed into their constituent unconditional subgraphs.

Consider a system ψ which consists of n CPGs Γ_i, $i = 1..n$. Each CPG Γ_i can be decomposed into n_i unconditional subgraphs g_j^i, $j = 1..n_i$. In Figure 4, for example, we have 3 unconditional subgraphs g_1^1, g_2^1, g_3^1 derived from Γ_1 and two, g_1^2, g_2^2 derived from Γ_2.

At the same time, each CPG Γ_i can be transformed (as shown in subsection 5.1) into a simple task graph G_i, by transforming conditional edges into ordinary ones and dropping the conditions. When deriving the worst case delay on Γ_i we apply the analysis from section 4 (algorithm DelayEstimate in Figure 3) separately to each unconditional subgraph g_j^i in combination with the graphs (G_1, G_2, ...G_{i-1}, G_{i+1}, G_n). This means that we consider each alternative track from Γ_i in the context of the system, instead of the whole subgraph G_i as in the previous approach. This is described by the algorithm DE/CPG in Figure 6 a). Estimation for the whole system is performed as shown in the algorithm DE/BF in Figure 6 b).

Such an approach, we call it BF, while producing tight bounds on the delays, can be expensive from the runtime point of view, because it is applied for each unconditional subgraph. In general, the number of unconditional subgraphs can grow exponentially. However, for many of the practical systems this is not the case, and the brute force method can be used. Alternatively, less expensive methods, like those presented below, should be applied.

5.3 Condition Separation (CS)

In some situations, the explosion of unconditional subgraphs makes the brute force method inapplicable. Thus, we need to find an analysis that is situated somewhere between the two alternatives discussed in 5.1 and 5.2, which means it should not be too pessimistic and should run in acceptable time.

A first idea is to go back to the DelayEstimate algorithm in Figure 3, and use the knowledge about conditions in order to update the *maxsep* table. Thus, if two processes P_i and P_j never overlap their execution because they execute under alternative values of conditions, then we can update *maxsep*$[P_i, P_j]$ to 0, and thus improve the quality of the delay estimation. Two processes P_i and P_j never overlap their execution if there exists at least one condi-

```
DE/CS(system ψ)
-- derives worst case delays for each CPG in the system ψ
   transform each Γi ∈ ψ into the corresponding Gi ∈ S
     and keep guard XPi for each Pi
   for each Gi ∈ S
      -- derives the worst case delay of a task graph Gi considering
      -- the influence from all other task graphs in the system S
      for each pair (Pi , Pj) in Gi
         maxsep[Pi, Pj] = ∞
      end for
      step = 0
      repeat
         LatestTimes(Gi)
         EarliestTimes(Gi)
         for each Pi ∈ Gi
            MaxSeparations(Pi)
         end for
         for each pair (Pi , Pj) in Gi
            if ∃C, C ⊂ XPi ∧ C̄ ⊂ XPj then
               maxsep[Pi, Pj] = 0
            end if
         end for
      until maxsep is not changed or step > limit
   δΓi is the worst case delay for Γi
   end for
end DE/CS
```

Figure 7. Delay Estimation using Condition Separation

tion C, so that $C \subset X_{Pi}$ (X_{Pi} is the guard of process P_i) and $\overline{C} \subset X_{Pj}$.

In this approach, called CS, we practically use the same algorithm as for ordinary task graphs and try to exploit the information captured by conditional dependencies in order to exclude certain influences during the analysis. In Figure 7 we show the algorithm DE/CS which performs delay estimation based on this heuristic.

5.4 Relaxed Tightness Analysis (RT)

The two approaches discussed here are similar to the brute force algorithm (Figure 6) presented in subsection 5.2. However, they try to improve on the execution time of the analysis by reducing the complexity of the DelayEstimate algorithm (Figure 3) which is called from the DE/CPG function (Figure 6 a). This will reduce the execution time of the analysis, not by reducing the number of subgraphs which have to be visited (like in subsection 5.3), but by reducing the time needed to analyze each subgraph. As our experimental results show (section 6) this approach can be very effective in practice. Of course, by the simplifications applied to DelayEstimate the quality of the analysis is reduced in comparison to the brute force method.

We have considered two alternatives of which the first one is more drastic while the second one is trying a more refined trade-off between execution time and quality of the analysis.

With both these approaches, the idea is not to run the iterative tightening loop in DelayEstimate that repeats until no changes are made to *maxsep* or until the *limit* is reached. While this tightening loop iteratively reduces the pessimism when calculating the worst case response times, the actual calculation of the worst case response times is done in LatestTimes, and the rest of the algorithm in Figure 3 just tries to improve on these values. For the first approach, called RT1, the function DelayEstimate has been transformed like in Figure 8 a).

However, it might be worth using at least the MaxSeparations in order to obtain tighter values for the worst case response times. For the alternative RT2 in Figure 8 b), DelayEstimateRT2 first calls LatestTimes and EarliestTimes, then MaxSeparations in order to build the *maxsep* table, and again LatestTimes to tighten the worst case response times.

```
DelayEstimateRT1(task graph G, system S)
   LatestTimes(G)
end DelayEstimateRT1
```

a) Delay Estimation for RT1

```
DelayEstimateRT2(task graph G, system S)
   for each pair (Pi , Pj) in Gi
      maxsep[Pi, Pj] = ∞
   end for
   LatestTimes(G)
   EarliestTimes(G)
   for each Pi ∈ G
      MaxSeparations(Pi)
   end for
   LatestTimes(G)
end DelayEstimateRT2
```

b) Delay Estimation for RT2

Figure 8. Delay Estimation for the RT Approaches

6. EXPERIMENTAL RESULTS

We have performed several experiments in order to evaluate the different approaches proposed. The two main aspects we were interested in are the quality of the delay estimation and the scalability of the algorithms for large examples. A first set of massive experiments were performed on conditional process graphs generated for experimental purpose.

We considered architectures consisting of 2, 4, 6, 8 and 10 processors. 40 processes were assigned to each node, resulting in graphs of 80, 160, 240, 320 and 400 processes, having 2, 4, 6, 8 and 10 conditions, respectively. The number of unconditional subgraphs varied for each graph dimension depending on the number of conditions and the randomly generated structure of the CPGs. For example, for CPGs with 400 processes, the maximum number of unconditional subgraphs is 64. 30 graphs were generated for each graph dimension, thus a total of 150 graphs were used for experimental evaluation. Worst case execution times were assigned randomly using both uniform and exponential distribution. All experiments were run on a Sun Ultra 10 workstation.

In order to evaluate the quality of the results, we need a cost function that captures, for a certain system, the tightness of the delays produced by the proposed approaches. Our cost function is the difference between the deadline and the estimated worst case delay of a CPG, summed for all the CPGs in the system:

$$\text{cost} = \sum_{i=1}^{n} (D_{\Gamma_i} - \delta_{\Gamma_i})$$

where n is the number of CPGs in the system, $\delta_{\Gamma i}$ is the estimated worst case delay of the CPG Γ_i, and $D_{\Gamma i}$ is the deadline on Γ_i. A higher value for this cost function, for a given system, means that the corresponding approach produces better results (the estimation is less pessimistic).

For each of the 150 generated example systems and each of the five approaches to delay estimation we have calculated the cost function. Figure 9 presents the average percentage deviations of the cost function obtained in each of the five approaches, compared to the value of the cost function obtained with the BF approach. The BF is the least pessimistic approach and therefore has the largest value for the cost function. A smaller value for the percentage deviation means a larger cost function, thus a better result. The percentage deviation is calculated according to the formula:

$$\text{deviation} = \frac{\text{cost}_{BF} - \text{cost}_{approach}}{\text{cost}_{BF}} \cdot 100$$

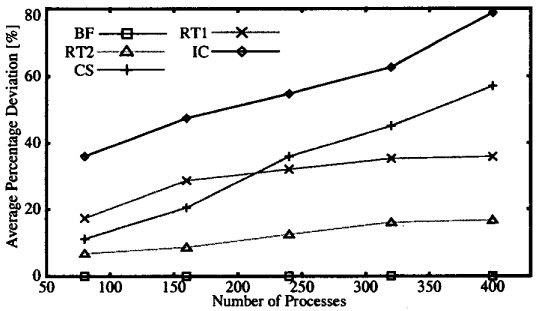


Figure 9. Quality of Estimation with Number of Processes

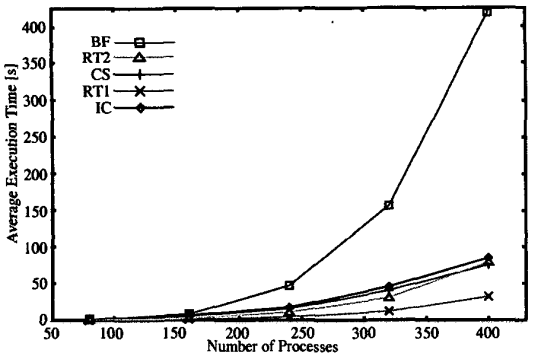


Figure 10. Average Execution Time

Figure 10 presents the average runtime of the algorithms, in seconds. The brute force approach, BF, performs best in terms of quality at the expense of a large execution time. At the other end, the straightforward approach IC, that ignores the conditions, performs worst and becomes more and more pessimistic as the system size increases. It is interesting to mention that the low quality IC approach has also an average execution time which is equal or comparable to the much better quality heuristics (except the BF, of course). This is because it tries to improve on the worst case delays through the iterative loop presented in DelayEstimate, Figure 3.

Let us turn our attention to the three approaches CS, RT1, and RT2 that, like the BF, consider conditions during the analysis but also try to perform a trade-off between quality and execution time. Figure 9 shows that the pessimism of the analysis is dramatically reduced by considering the conditions during the analysis. The RT1 and RT2 approaches, that visit each unconditional subgraph, perform in average better than the CS approach that considers condition separation for the whole graph. However, CS is comparable in quality with RT1, and even performs better for graphs of size smaller than 240 processes (4 conditions, maximum 16 subgraphs). The RT2 analysis, that tries to improve the worst case response times using the MaxSeparations, as opposed to RT1, performs best among the non-brute-force approaches. As can be seen from Figure 9, RT2 has less than 20% average deviation from the solutions obtained with the brute force approach. However, if faster runtimes are needed, RT1 can be used instead, as it is twice faster in execution time than RT2.

We were also interested to compare the five approaches with respect to the number of unconditional subgraphs in a system. For the results depicted in Figure 11 we have assumed CPGs consisting of 2, 4, 8, 16, and 32 unconditional subgraphs of maximum 50 processes each, allocated to 8 processors. Figure 11 shows that as the number of subgraphs increases, the differences between the approaches grow while the ranking among them remains the same, as resulted from Figure 9. The CS approach performs better than RT1 with a smaller number of subgraphs, but RT1 becomes better as the number of subgraphs in the CPGs increases.

Finally, we considered a real-life example implementing a vehicle cruise controller modeled using a conditional process graph. The graph has 32 processes, two conditions (4 subgraphs), and it was mapped on an architecture consisting of 4 nodes (processors), namely: Anti Blocking System, Transmission Control Module, Engine Control Module and Electronic Throttle Module. The period of the CPG was 200 ms, and the deadline was set to 110 ms. Without considering the conditions, IC obtained a worst case delay of 138 ms. The same result was obtained with the CS approach, and this is because the alternative tracks were mapped on different processors, thus not influencing each other. However, the brute force approach BF produced a worst case delay of 104 ms which proves that the system implementing the vehicle cruise controller is, in fact, schedulable. Both RT1 and RT2 produced the same worst case delay of 104 ms as the BF.

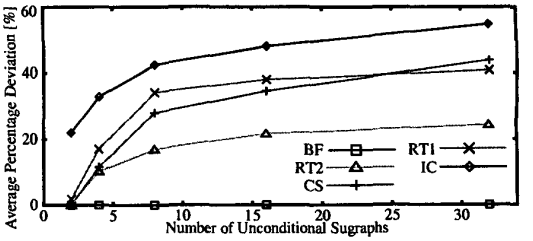


Figure 11. Quality of Estimation with Number of Subgraphs

7. CONCLUSIONS

In this paper we proposed solutions to performance estimation for hard real-time systems with control and data dependencies.

The systems are modeled through a set of conditional process graphs that are able to capture both the flow of data and that of control. We consider distributed architectures and a scheduling policy based on a static priority preemptive strategy.

Five approaches to delay estimation of such systems are proposed. Extensive experiments and a real-life example show that by considering the conditions during the analysis, the pessimism of the analysis can be drastically reduced.

While the brute force approach BF performed best, at the expense of execution time, the RT2 approach is able to obtain results with less than 20% average loss in quality, in a very short time.

REFERENCES

[1] N.C. Audsley, K.W. Tindell, A. Burns, "The End of the Road for Static Cyclic Scheduling", Proc. Euromicro Workshop on Real-Time Systems, 36-41, 1993.

[2] A. Doboli, P. Eles, "Scheduling under Control Dependencies for Heterogeneous Architectures", Proc. International Conference on Computer Design, 602-608, 1998.

[3] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, P. Pop, "Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems", Proc. DATE, 23-26, 1998.

[4] G. Fohler, "Realizing Changes of Operational Modes with Pre Run-time Scheduled Hard Real-Time Systems", *Responsive Computer Systems*, H. Kopetz-Y. Kakuda ed., 287-300, Springer, 1993.

[5] R. Gerber, D. Kang, S. Hong, M. Saksena, "End-to-End Design of Real-Time Systems", *Formal Methods in Real-Time Computing*, D. Mandrioli-C. Heitmeyer ed., John Wiley & Sons, 1996.

[6] P.B. Jorgensen, J. Madsen, "Critical Path Driven Cosynthesis for Heterogeneous Target Architectures", Proc. International Workshop on Hardware-Software Co-design, 15-19, 1997.

[7] C. Lee, M. Potkonjak, W. Wolf, "Synthesis of Hard Real-Time Application Specific Systems", *Design Automation for Embedded Systems*, 4, 215-241, 1999.

[8] C. L. Liu, J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *Journal of the ACM*, 20(1), 46-61, 1973.

[9] P. Pop, P., Eles, Z., Peng, "Scheduling with Optimized Communication for Time-Triggered Embedded Systems", Proc. International Workshop on Hardware-Software Co-design, 78-82, 1999.

[10] P. Pop, P., Eles, Z., Peng, "Bus Access Optimization for Distributed Embedded Systems based on Schedulability Analysis", Proc. DATE, 2000.

[11] K. Tindell, "Adding Time-Offsets to Schedulability Analysis", Department of Computer Science, University of York, Report Number YCS-94-221, 1994.

[12] K. Tindell, J. Clark, "Holistic Schedulability Analysis for Distributed Hard Real-Time Systems", *Microprocessing and Microprogramming*, 40, 117-134, 1994.

[13] T. Yen, W. Wolf, "Performance estimation for real-time distributed embedded systems", *IEEE Transactions on Parallel and Distributed Systems*, Volume: 9(11), 1125 -1136, Nov. 1998.