Proc. ISORC2006 (9th IEEE CS Int' Symp. on Object & Component Oriented Real-Time Distributed Computing), Gyeongju, Korea, April 24 - 26, 2006, pp.408-418.

A Hybrid Approach in TADE for Derivation of Execution Time Bounds of Program-Segments in Distributed Real-Time Embedded Computing

Chansik Im

Intel Corporation
Hillsboro, OR, USA
chansik.im@intel.com

K. H. (Kane) Kim

University of California
Irvine, CA, USA
khkim@uci.edu, http://dream.eng.uci.edu

Abstract: Guaranteeing response times of real-time (RT) distributed computing systems has been recognized as one of the biggest challenges by the RT software research community for three decades. The concept of a hybrid approach that combines analytical derivation approaches and testing-based statistical derivation approaches in a symbiotic form for meeting this challenge was presented in recent years. However, concrete practical hybrid approaches are still in early stages of development. One such approach pursued by the authors and their collaborators is presented here. This paper focuses on the cases of deriving tight execution time bounds of the segments of object methods which do not involve calls for services from the operating system kernel and middleware. A case-study that demonstrates how the adopted approach works in handling a simple practical application is also presented.

Keywords: real time, hybrid, worst-case execution time, worst-case execution path, program-segment, method-segment, acyclic path-segment, path enumeration, curve-fitting, execution time bound, TETB, analysis, measurement

1. Introduction

As the demands for large-scale mission-critical real-time (RT) distributed computing systems grow steadily, the urgency for developing a rigorous, broadly applicable method for determining the execution safety of such systems also grows. By *execution safety* we refer to the absence of the possibility of violating hard deadlines in an RT distributed computing system [Kim00a, Kim00b, Kim03]. The needs for new-generation RT safety-critical distributed embedded computing systems, such as tsunami alert systems and remote surgery systems, heighten the importance of the execution safety.

Determination of the execution safety involves analysis of the timing behaviors of RT distributed computing software components and their interactions including coordinated sharing of limited resources. A key requirement here is the ability to determine *acceptable and credible upper bounds on the service times* of RT distributed computing systems. Large-scale RT distributed computing systems are frequently constructed in a layered form containing layers such as hardware subsystems, operating system (OS) kernels and kernel-level communication protocols, middleware, and application software. Therefore, the problem of deriving tight service time bounds of large-scale RT distributed computing systems becomes extremely complex.

Determining tight service time bounds or equivalently, guaranteeing response times of RT distributed computing systems has been recognized as one of the biggest challenges by the RT software research community for three decades. However, the state-of-the-art in analyzing the worst-case service time of RT distributed computing systems remains substantially short of being applicable to practical RT distributed computing application systems.

Past research in analysis of worst-case execution times of RT programs has been mostly confined to the cases of single-node programs of simple types, e.g., program showing simple strictly periodic patterns and involving no OS service calls [Hec03, Li95, Par93, Pus97, Sha89].

Industry practice in deriving service time bounds has been to fully rely on limited numbers of test-runs. However, such approaches based on test-runs must contain techniques that address the following fundamental issue.

A limited number of test-runs with randomly and massively generated input data sets may not necessarily cover the worst-case execution scenario. Therefore, a margin value needs to be added to the maximum service time observed during the test-runs in order to derive a credible service time bound. Moreover, a reasonable estimation of the probability of the margin value turning out to be insufficient at run time should also be available.

Recently, researchers have proposed measurement-based approaches [Pus02, Wen05, Wil05] that focus on the issue of maximizing the probability of covering the worst-case execution scenario. A practical and systematic approach for handling the margin value has not been addressed.

We believe that the following factors are the biggest

obstacles to be removed on the way to meet the challenge of successfully establishing the technology for deriving tight service time bounds of sizable RT distributed computing systems [Kim00a, Kim00b]:

(1) Lack of a fully general and yet easily analyzable RT distributed program structure that provides rules to prevent major sources of temporal non-determinism from arising in both sharing of execution resources and accessing common data concurrently;

(2) Lack of useful and accurate timing models for the execution infrastructure, including the OS kernels, middleware, and the communication infrastructure, which enable systematic analysis of the worst-case timing behavior of the application systems.

Experiences of the research community have shown that it is very important to establish easily-analyzable RT distributed program structure before attempting to take the challenge of deriving tight service time bounds of RT distributed computing systems. Recently, research on establishing suitable program structures for RT distributed computing systems has produced some useful results [Bol00, Kim97, Kim00c, Kop97, OMG02, OMG05, Pus02, Sel02]. Time is ripe for developing techniques for analyzing temporal behaviors of RT distributed computing systems built with the adoption of modern program structuring approaches.

Researchers in the UCI DREAM Laboratory and their collaborators have been engaged in slow development of a tool-set since late 1990's [Kim00a, Kim00b, Kim03]. The tool-set was named the *Timeliness-Assured Design Environment* (*TADE*) recently. TADE is aimed for enabling major reduction in the system engineers' efforts in producing distributed RT embedded computing systems with service time guarantees. Among others, TADE adopted a hybrid approach, which combines analytical methods and measurement-based methods in a symbiotic form, for derivation of tight service time bounds of RT distributed computing systems [Kim03].

The high-level architecture and major approaches adopted in TADE for service time guarantees are discussed in Section 2. New concrete techniques for deriving *tight execution time bounds* (TETBs) of segments of object methods in RT component-based distributed computing systems are presented in Section 3. Only the method-segments which involve no service calls to the OS kernel and middleware are dealt with in detail here. These techniques are meant to be the first concrete step in realizing a hybrid approach in a form that can be of practical help to RT distributed computing software developers.

Several case-studies which have involved derivation of tight service time bounds of some real-world applications by use of the techniques presented in Section 3, have been conducted. The simplest case study that demonstrates how the adopted techniques work in handling a bubble sort program, an almost trivially simple application, is presented in Section 4.

Then in Section 5, possible application and extension of the hybrid approach presented in Section 3 for the cases of handling program-units larger than simple method-segments are discussed. The paper concludes in Section 6.

2. Major elements of TADE

2.1 Structuring of application software as TMO networks

Distributed application computations often exhibit complex forms of synchronizations and competitions for accessing shared data and obtaining OS services. Because of the logical complexity of sizable RT distributed applications and execution engines (aggregates of hardware, OS, and communication infrastructure), the analysis of execution safety becomes easily an unmanageable problem if the applications are not well structured [Kim00b]. Conventional process-network structuring of RT distributed applications leaves too wide a door open for incorporating various difficult-to-analyze competing modes for accessing shared data and obtaining OS services, compared to recently emerged RT object structuring. In other words, *guaranteeing timely services* (i.e., guaranteeing the abilities to meet hard deadlines) of RT distributed computing applications becomes much more feasible with proper RT object structuring than with the conventional use of processes or process-segments as modules.

One such RT object structuring approach is the *time-triggered message-triggered object* (TMO) structuring scheme [Kim97, Kim00c, http://dream.eng.uci.edu /TMO/TMO.htm]. The TMO incorporates several rules for execution of its components that make the analysis of the worst-case time behavior of TMOs to be systematic and relatively easy while not reducing the programming power in any way. The TMO scheme was formulated from the beginning with the objective of enabling *design-time guaranteeing of timely actions*.

Moreover, the expressive power of the TMO is so strong that all conceivable distributed computing applications, including both non-RT and RT applications, can be structured in the form of TMO networks. Therefore, *TMO network structuring of applications* has been adopted as an integral part of TADE.

2.2 Stepwise integration of service time bounds

TADE is basically aimed for enabling systematic derivation of *tight bounds on service times* of RT objects, i.e., TMOs. A service time of a TMO is the amount of time that the TMO takes in accepting a request, executing an appropriate service method, and returning a result. In deriving tight service time bounds, TADE uses the divide-and-conquer strategy by determining the following parts

separately and then integrating them into service time bounds:

(1) A tight bound on the execution time of every *program-segment* that involves no calls for services from the OS kernel and middleware, and

(2) A tight bound on the completion time of every call for an OS service including every call for a message communication service involving the communication infrastructure.

2.3 Analysis-testing hybrid approach for derivation of tight service time bounds

In deriving service time bounds, basically two types of approaches exist. One is the *testing-based statistical approach* which is essentially to measure service times during a limited number of test-runs of the target subsystem and adopt the maximum among the measured values plus some margin as a service time bound. This approach has some obvious limitations. That is, the worst-case service time observed during test-runs may or may not be the real worst-case service time. In this sense, the bound adopted through this statistical approach is a *soft bound* which is associated with a non-zero probability of being exceeded at run time. Therefore, such soft bounds should be used together with safety-backup measures which can come into play whenever the service time bounds adopted are violated.

The other approach is to first establish the service time bounds of the execution hardware or a virtual machine underlying the target software component through analysis or extensive testing with the possible cooperation of the manufacturers. Then the service time bounds of the target component are derived in conservative manners through *analyses* of the logical structure of the target component and application of the established service time bounds of the underlying physical or virtual machine. Due to the conservative estimates employed frequently in this *analytical determination* process, the resulting service time bounds tend to have large error margins, i.e., become much larger than actual worst-case service times. The analytically derived bounds are *hard bounds*.

TADE adopts a pragmatic hybrid of analysis and testing approaches. A concrete illustration will be given later in Section 3.

Figure 1 depicts TADE in an abstract form. TMOSM in the figure represents the TMO Support Middleware [Kim00c], a core part of the TMO execution engine. ROAFTS (*Real-time Object-oriented Adaptive Fault Tolerance Support*) [Kim01] is an extension of TMOSM which is capable of fault-tolerant execution of TMO-structured distributed applications. The extended ROAFTS handles TMOs and replicas containing specifications of soft bounds and hard bounds for service times. The API set in the figure represents an approximation of an idealistic language for programming of fault-tolerant TMOs distributed across the network.

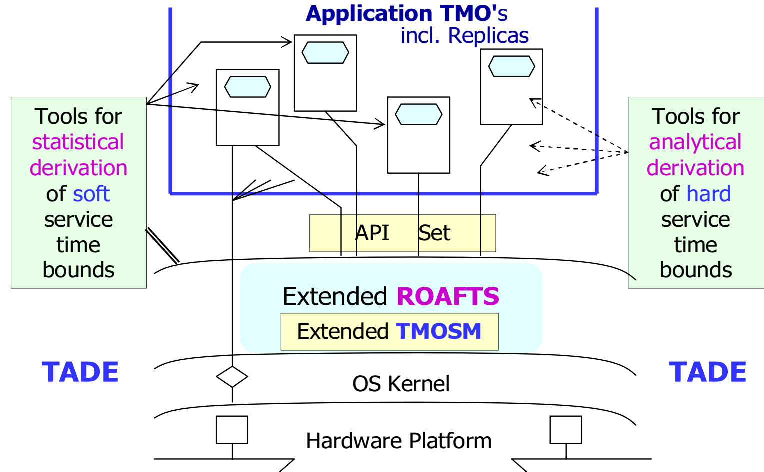


Figure 1. TADE (Timeliness-Assured Design Environment)

3. Techniques for deriving tight execution time bounds of program-segments

The *program-segment* is basically a sequence of code including neither OS calls nor TMO method calls [Kim00a, Kim00b]. Since the program-segment in this paper is a segment of a TMO method, the two terms, "program-segment" and "method-segment", are used interchangeably in the rest of this paper. Therefore, the effects of the activities of the TMOSM and an underlying OS kernel, including message communication, need not be considered during the derivation of a *tight execution time bound* (TETB) of a program-segment.

One approach that has been studied extensively and is applicable to the analysis of a TETB of a program-segment is the *implicit path enumeration* (IPE) technique [Li95, Pus97]. In this approach, a program-segment is represented as a directed graph where each node represents a *basic block* which is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end [Aho86].

The execution time of every sequential piece of code is assumed to be invariable over time. This is an assumption not matching well with the characteristics of modern microprocessors. Suppose ***xi*** is an execution count of a basic block ***Bi*** and ***ci*** is an execution time of ***Bi***. If there are ***N*** basic blocks in a given application program-segment, then the total execution time of the given program-segment is:

$$\text{Execution time } T(N) = \sum_{i}^{N} c_i x_i$$

Once the object function $T(N)$ is formulated, $T(N)$ is passed to a *integer linear programming* (ILP) solver to find out a combination of xi that maximizes $T(N)$.

3.1 Acyclic path-segment (APS) and APS-based path enumeration (APE)

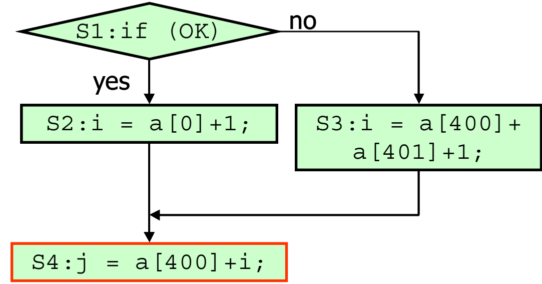
In order to reflect the characteristics of modern microprocessors better than in the case of the IPE approach, program-segments are further modularized in TADE into *acyclic path-segments* (APSs). An APS is *a possible execution sequence of instructions that does not contain any cycle.* Starting points of an APS can include an entry point of a given program-segment and heads of loops inside the given program-segment. Ending points of an APS can include loop-backward branches and an exit point of the encompassing program-segment.

Although there can be a number of different procedures for choosing APSs, a simple practical heuristic is to follow a depth-first search procedure where one APS is sealed whenever either the boundary of a program-segment or a loop-backward branch is encountered. For example, two APSs can be identified from the code-segment presented in Figure 2(a). The execution time of S4 may vary depending on which APS is in execution. Given an APS, the execution time of S4 can be easily estimated with a proper data cache analysis, which is usually not done with the IPE approach based on the assumption that the execution time of a basic block is invariable. The APSs in Figure 2 are of trivial complexity and the determination of a TETB of a more complex APS may involve a combination of an analysis and extensive testing. In any case, the worst-case execution time (WCET) of an APS can be estimated more accurately through such effort than through mere summation of the WCET estimates of constituent basic blocks.

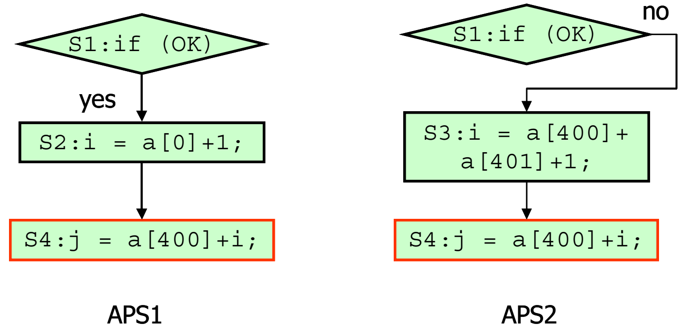
After all APSs in a given program-segment are identified, an *APS graph*, a graph showing the control-flow among the APSs, can be formed. Then the ILP technique can be applied to the APS graph to find the worst-case execution path / scenario of the program-segment. This technique is called the *APS-based path enumeration* (APE) here.

3.2 Hybrid approach involving the APS-based path enumeration (APE) and a statistical approach

It is possible that the path identified as the worst-case execution path with the APE technique is not the actual worst-case execution path. This is because the execution time of a path formed by a series of two APSs is not as accurately estimated as that of an individual APS. It is particularly so when such a path is a part of a loop. Therefore, it is safe to identify multiple candidates of the worst-case execution path and examine their execution times through various means including not just analysis but also testing-based approaches. This will result in a better-quality TETB of a given program-segment. It can be done by applying the APE technique iteratively.



(a) Original control flow graph



(b) APSs identified from (a)

Figure 2 Acyclic path segments

Let x_i be the execution count of an APS A_i , and c_i be the execution time of A_i. Given that there are N APSs in a given program-segment PS, the total execution time of the program-segment, $T(PS)$, is given by:

$$T(PS) = \sum_{i}^{N} c_i x_i$$

T_{worst1} is the value of $T(PS)$ when the worst-case execution path is found by the ILP solver. Then, with $T(PS) < T_{worst1}$ added as a new constraint to the original constraint set, the ILP solver will find the second worst-case execution path with its execution time, T_{worst2}. By repeating this procedure, application designers can find multiple candidates of the worst-case execution path.

A question then arises as to when to stop identifying candidates of the worst-case execution path. It is basically highly dependent on the application structure and context. Application designers may use one of the following ways or a combination of them to specify when to stop searching for candidates of the worst-case execution path of a given program-segment:

(1) Application designers can specify an absolute number of candidates to be used, e.g., 'Find ten candidates'.

(2) Application designers can stop searching for candidates when the execution time of the i-th candidate, T_{worsti}, is less than or equal to THRESHOLD (%) x T_{worst1}, where the value of THRESHOLD is given by application designers.

(3) Application designers may stop searching for candidates when the difference between T_{worsti} and $T_{worsti+1}$ becomes suddenly large.

During the phase of identifying path candidates, application designers can supply some additional information on the control-flow among the APSs whenever they have such knowledge. Insertion of such knowledge into the APE process can result in a better-quality TETB of the program-segment.

Since the APE process generates multiple candidate execution paths / scenarios and the associated TETBs could be overly pessimistic, a testing-based statistical approach is invoked in TADE as a supplement to APE. Therefore, execution times of the program-segment are measured during a limited number of test-runs with some input data sets on the target execution engine platform. The issues in obtaining appropriate input data sets are not discussed in this paper and instead are referred to [Im05]. Thereafter, the maximum among the measured values plus some margin will be considered as another TETB of the program-segment. The next questions are then:

(1) How to determine the margin value to be added to the maximum measured execution time of the given program-segment, and

(2) How to estimate the probability of violating the soft execution time bound obtained in (1).

A distribution of measured execution times of a given program-segment, ***PS***, can be expressed with *a probability mass function* (PMF), $p_{PS}(X)$, where X is a *discrete random variable* representing the measured execution time of ***PS*** [Ros02]. $p_{PS}(t)$ of X can be defined as:

$$p_{PS}(t) = P\{ X = t \}$$

where, $p_{PS}(t)$ is the number of occurrences of the measured execution time of the program-segment turning out to be equal to t divided by the total number of test-runs. The summation of $p_{PS}(t)$ over all possible t is equal to one.

In fact, a testing-based statistical approach for deriving a soft execution time bound can be applied at the APS level as well, especially when the APSs are quite complex.

Consider the statistical distribution of measured execution times of a given APS that resulted from 100 test runs in Table 1. The third column of the table shows the probability mass function of the measured execution time of the program-segment.

$T_{measured}$	Number of Occurrences	$p_{PS}(T_{measured})$
100	15	0.15
110	25	0.25
118	20	0.20
122	35	0.35
140	5	0.05

Table 1. Statistical distribution of measured execution time of a given program-segment

The *cumulative distribution function* (CDF) [Ros02], ***F***, of the measured execution times of a given program-segment can be expressed in terms of $p_{PS}(t)$ by

$$F(t) = \sum_{\text{all } t_i \le t} p_{ps}(t_i)$$

The graph representation of the cumulative distribution function of the measured execution times of the given APS presented in Table 1 is depicted in Figure 3.

The application of a *curve-fitting*, or *regression*, technique to the cumulative distribution function graph of the measured execution times of the given program-segment can provide a hint on a way to determine a margin value to be added to the maximum measured execution time. This will lead to a soft execution time bound of the given program-segment. Also, the probability of the soft execution time bound being exceeded at run time can be determined.

Figure 3 shows a smooth curve approximately fitting the cumulative distribution function, $F_{PS}(t)$, of the given APS. The *Richard model*, as described below, was used to produce the curve depicted in Figure 3.

$$y = \frac{a}{(1 + e^{b-cx})^{1/d}}$$

Application designers may also choose other sigmoidal models that may approximate the cumulative distribution function of the measured execution times obtained during test-runs. CurveExpert 1.3 [Hya05], a freeware package for simple curve-fitting, finds appropriate values for coefficients (a, b, c, and d) according to the points in the cumulative distribution function graph. The curve in Figure 3 becomes saturated when the execution time reaches around 213. This means that the probability of the execution time not exceeding 213 becomes practically one. Also, the tool can return the corresponding estimated execution time of a given APS when the probability of the execution time not exceeding the estimated execution time, α, is provided. In Figure 3, if α is equal to 0.99, then the corresponding estimated execution time of the APS is 147.14. If α is equal to

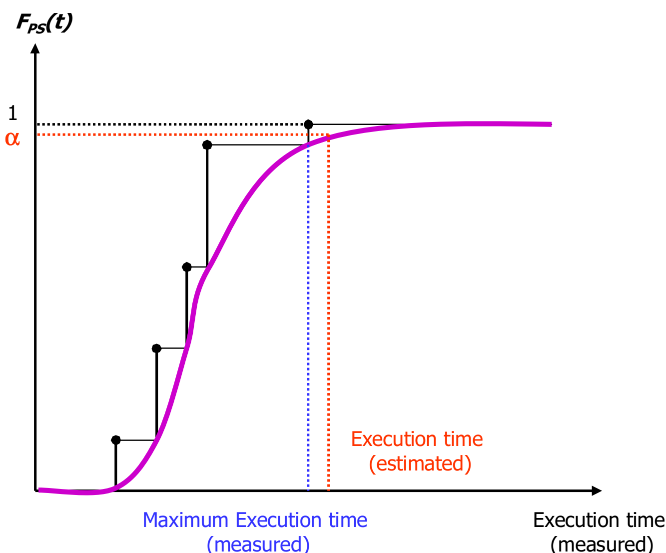


Figure 3. Finding a soft execution time bound of an APS by curve fitting

0.999, then the corresponding estimated execution time of the APS becomes 150.56.

In determining a soft execution time bound of the APS by use of curve fitting and finding the estimated execution time corresponding to a certain α, application designers must obviously select the value of α very carefully. If α is not sufficiently high, there is a non-negligible risk of the actual execution time during a certain run exceeding the derived execution time bound. If there is a hard TETB known for the APS, then it can be reflected in adjusting the soft execution time bound derived from the use of α.

Note that the maximum value of $F_{PS}(t)$ of a graph obtained through the curve-fitting technique may go beyond 1. This is because we set the value of $F_{PS}(Max_ET_Obs)$, where *Max_ET_Obs* represents the maximum execution time observed during test-runs, to one. In this situation $F_{PS}(t)$ going beyond 1 is natural since there is non-negligible probability of the actual worst-case execution time being larger than *Max_ET_Obs*. Therefore, the graph should preferably be normalized in such a way that it becomes saturated when $F_{PS}(t) = 1$. On the other hand, a graph in a certain case may start to become saturated when the value of $F_{PS}(t)$ is less than 1. In such a case, the normalization of the graph is not necessary.

In the above discussion on derivation of a soft execution time bound, the case of an APS was used for illustration purpose. The same approach can be used for derivation of a soft execution time bound of a program-segment. To be more specific, each candidate for the worst-case execution path in the program-segment can be put through test-runs and a soft execution time bound can be derived. Once a soft bound for every candidate path is obtained, those soft bounds and the hard TETB obtained through the APE process can be considered together to derive a final version of the TETB of the program-segment.

4. Case study

A bubble sort algorithm, whose worst-case execution input sets are known, is used here to demonstrate the techniques proposed in the paper. Because a bubble sort does not involve any OS service calls, it fits the definition of the program-segment. A more sophisticated case study with a sizable RT distributed computing application can be found in [Im05].

Figure 4 shows the code segment of the bubble sort. It is known that the execution time of this algorithm is bounded by $O(n^2)$. Also, if the input array is reversely sorted, the bubble sort algorithm is known to take the maximum execution time. If the size of the array is 1000, the innermost loop body, containing the swap operation, runs at most 499500 times.

A prototype of a tool for identification of APSs in a C++ program and derivation of TETBs of the APSs, has been implemented. The tool called the *APS Analyzer* (APSA) constructs a call hierarchy graph and control-flow graphs of procedures. Then, it identifies program-segments and APSs in the program-segments. Next, APSA instruments the APSs with the codes for measurement of their execution times during test-runs and derivation of TETBs. Thereafter, APSA automatically compiles and executes the instrumented code and from the measurement results, it produces TETBs of APSs.

APSA identified 11 APSs from the bubble sort program in Figure 4. The APS graph (APSG) generated by APSA is shown in Figure 5. The APSG shows that APS#10 consists of three basic blocks, 5, 6, and 10.

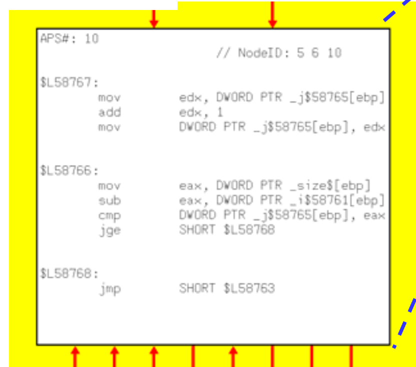
APSA instruments the APSs to enable test-run of the instrumented APSs without necessitating the generation of a customized input data set for each test-run. More details on the instrumentation approach can be found in [Im05]. Table 2 shows the 11 APSs with corresponding basic blocks. The third column shows the measured execution time of each instrumented APS. The instrumented version of the bubble sort program was executed on a Pentium 4 machine (2.8GHz, 1GB DDR RAM with 800 MHz system bus, Microsoft Windows XP, MS VC++ 7.1).

With the APSG shown in Figure 5 and the execution time information presented in Table 2, the APE technique was applied to identify candidates of the worst-case execution path of the bubble sort program. The TETB produced by the APE technique was 10242 μsec.

Then, with the worst-case input data set of the bubble sort, i.e., a reversely ordered integer array with 1000 elements, the execution times of the instrumented bubble

```
void BubbleSort(int * pArray, int size)
{
  for (int i = 0; i < size; i++)
    for (int j = 0; j < size - i; j++)
      if (pArray[j+1] < pArray[j])
      {
        int temp = pArray[j];
        pArray[j] = pArray[j+1];
        pArray[j+1] = temp;
      }
}
```

Figure 4. Bubble sort



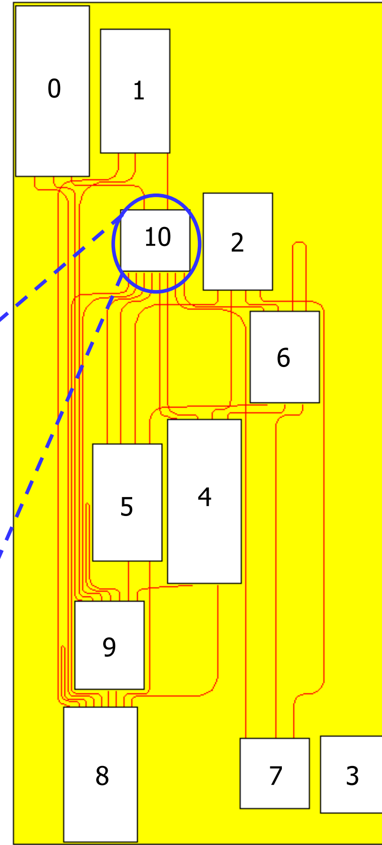


Figure 5. An APSG of bubble sort

sort program were measured 1000 times and the cumulative distribution function ***F(X)*** based on the measurement data is shown in Figure 6.

APS	Basic blocks belonging to APS	Measured execution time of instrumented APS (nsec)
0	1, 3, 4, 6, 7, 8, 9	32.05
1	1, 3, 4, 6, 7, 9	6.14
2	1, 3, 4, 6, 10	3.54
3	1, 3, 11	2.87
4	2, 3, 4, 6, 7, 8, 9	20.13
5	2, 3, 4, 6, 7, 9	4.50
6	2, 3, 4, 6, 10	2.87
7	2, 3, 11	2.16
8	5, 6, 7, 8, 9	20.50
9	5, 6, 7, 9	3.12
10	5, 6, 10	0.99

Table 2. APSs identified from bubble sort

The maximum execution time observed during the test-runs is 4748 µsec, which is almost a half of the TETB produced by the APE technique, 10242 µsec. The difference between the two time bounds is mainly due to the cache effect. During the program instrumentation, all memory access operations have been redirected randomly by the tool APSA, which will increase cache misses and, therefore, increase the execution time.

After adding the result obtained by the APE technique, 10242 µsec, as an analytically derived hard bound, the data has been passed to CurveExpert to obtain an appropriate smooth curve which approximates the measurement data as shown in Figure 6.

The tool produced the following curve:

$$y = \frac{a}{(1 + e^{b - cx})^{1/d}}$$

where $\boldsymbol{a} = 1.5717861$, $\boldsymbol{b} = 49.399415$, $\boldsymbol{c} = 0.011765744$, and $\boldsymbol{d} = 0.0054356987$.

However, the value of y of the curve goes beyond 1. After normalizing the curve, the value of the cumulative distribution function at the maximum observed execution time, ***F****(X*=4748 µsec*)* is 75.10%. This can be interpreted as follows:

'If 4748µsec is adopted as a soft execution time bound of the bubble sort, then the probability of the execution time bound being exceeded at least once during

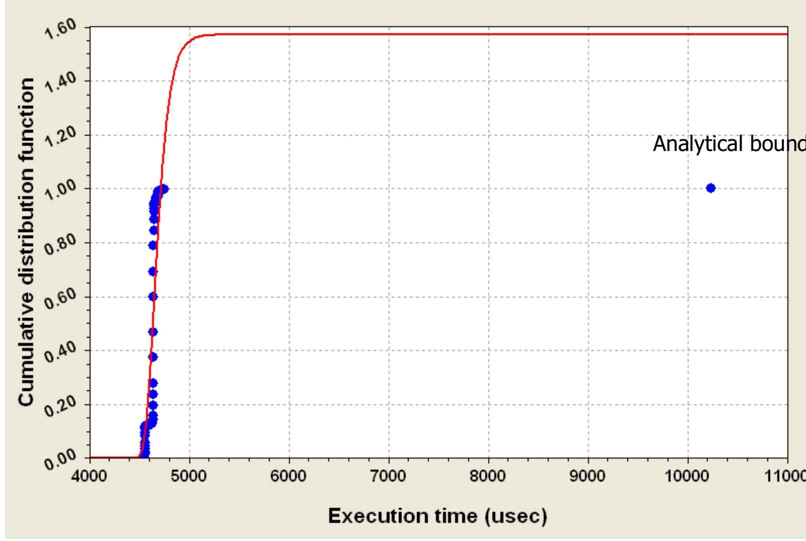


Figure 6. A CDF of the execution time of the bubble sort, a curve fitting, and an analytically derived bound

future executions is 24.90 %.'

The value of ***F(X)*** becomes 90% when the execution time bound is about 4833 μsec. The graph becomes saturated when the execution time is about 6463 μsec. For determining a TETB of the bubble sort, application designers can choose appropriate values of ***F(X)*** from the range reasonably close to 100%. Table 3 shows samples of soft execution time bounds for the bubble sort, and the probability of each of those execution time bounds being exceeded at run time according to the result from curve fitting and normalization.

Soft execution time bound (μsec)	Probability of the execution time bound being exceeded at run time(%)
4748	24.90
4833	10.00
5033	1.00
5092	0.5
5400	0.01
6463	Curve starts to become saturated

Table 3. Samples of soft execution time bounds of the bubble sort with probabilities of violation

5. Possible applications of the hybrid approach to program-units larger than method-segments

Service time bounds of TMOSM/OS services can also be obtained through the hybrid approach discussed in Section 3 if the source code is available. Otherwise, service time bounds of such services must be obtained through exhaustive test-runs with randomly and massively generated input data sets. Establishing an accurate and complete TMOSM/OS timing model and communication infrastructure timing model is beyond the scope of this paper, and it remains as future work.

Suppose service time bounds of TMOSM/OS services used by a given TMO method are available in the form of probability mass functions. Then, a distribution of the execution time of the TMO method, $p_{combined}(X)$, can be obtained by combining the TETBs of method-segments and the service time bounds of TMOSM/OS services. A TETB of the TMO method can then be obtained through the application of the curve-fitting technique to $p_{combined}(X)$.

On the other hand, if the TETBs of method-segments and TMOSM/OS services are available, candidates for the worst-case execution path in the TMO method can be identified by applying the variation of the APE technique in which nodes are method-segments and TMOSM/OS services rather than APSs. Then, the execution time of each of those candidate execution paths can be measured again during test-runs to obtain a distribution of the measured service time of the TMO method, $p_{measured}(X)$.

A soft time bound obtained from $p_{measured}(X)$ via the curve-fitting technique should be smaller than the TETB obtained from $p_{combined}(X)$ via the curve-fitting technique. This is because $p_{combined}(X)$ is obtained from the combination of TETBs of method-segments and TMOSM /OS services and the case of execution in which all those TETBs are exhibited, occurs rarely. Also, infeasible paths spanning method-segment boundaries may not be detected during the TETB analysis of method-segments.

Therefore, both the soft time bound obtained from $p_{measured}(X)$ and the TETB obtained from $p_{combined}(X)$ can be reflected in determining the final TETB of the TMO method. However, this plausibility of the hybrid approach must be validated through much further research work, including both analytical and experimental research, in the future.

6. Conclusion

The overall architecture of the tool-set for timeliness-assured design of RT distributed computing systems, TADE, has been introduced. This paper mainly focused on the derivation of TETBs of program-segments which are program units not containing any OS service calls including those for services of the communication

network infrastructure. Since analysis approaches and testing-based statistical approaches are complementary, TADE adopted hybrid approaches, combining analytical derivation approaches and statistical derivation approaches in a symbiotic form. Concrete instances of hybrid approaches were presented. A case study was presented as well.

The plausibility of applying essentially the same hybrid approaches to the cases of handling larger program-units, e.g., TMO methods, was briefly discussed. To enable such approaches, accurate and complete timing models for TMOSM/OS activities and communication network activities should be established. All such models and techniques must be developed and validated through much further research work, including both analytical and experimental research, in the future.

Acknowledgment: The research reported here is supported in part by the NSF under Grant Numbers 02-04050 (NGS), 03-26606 (ITR), and 05-24050 (CNS). No part of this paper represents the views and opinions of the sponsors mentioned above.

References

[Aho89] Aho, A.V., Sethi, R., and Ullman, J.D., '*Compilers, Principles, Techniques, and Tools*', Addison Wesley, 1986.

[Bol00] Bollella, G., and Gosling, James, “The Real-Time Specification for Java”, *IEEE Computer,* June, 2000, pp. 47-54.

[Hec03] R. Heckman, M. Langenbach, S. Thesing, and R. Wilhelm, “The influence of processor architecture on the design and the results of WCET tools”, *Proc. of the IEEE*, volume 91:7, pp. 1038–1054, July 2003.

[Hya05] Hyams, D.G., '*CurveExpert, A curve fitting system for Windows*', http://www.ebicom.net/~dhyams /cmain.htm, 2005.

[Im05] Im, C.S., '*A Hybrid Approach for Derivation of Tight Execution Time Bounds of Program-Segments and Service Time Bounds of Simple Object Methods in Real-Time Distributed Computing Systems*', Ph.D. Dissertation, EECS Dept., Univ. of Calif., Irvine, USA, Dec. 2005.

[Kim97] Kim, K.H., "Object Structures for Real-Time Systems and Simulators", *IEEE Computer*, August 1997, pp.62-70.

[Kim00a] Kim, K.H., "Analysis of Guaranteed Service Times of Distributed Real-Time Objects", *Proc. ISORC 2000 (IEEE CS 3rd Int'l Symp. on Object-oriented Real-time distributed Computing)*, Newport Beach, CA, pp.408-410, March 2000.

[Kim00b] Kim, K.H., Choi, L., and Kim, M.H., "Issues in Realization of an Execution Time Analyzer for Distributed Real-Time Objects", *Proc. 3rd IEEE Symp. on Application-Specific Systems and Software Engineering Technology (ASSET'00)*, March 2000. pp. 171-178.

[Kim00c] Kim, K.H., "APIs for Real-Time Distributed Object Programming", *IEEE Computer*, June 2000, pp.72-80.

[Kim01] Kim, K.H., “Middleware of Real-Time Object Based Fault-Tolerant Distributed Computing Systems: Issues and Some Approaches”, *Proc. IEEE 2001 Pacific Rim Int'l Symp. on Dependable Computing (PRDC 2001)*, Dec. 2001, Seoul, pp. 3-8 (keynote paper).

[Kim03] Kim, K. H., “Timeliness Assurance via Hybrid Approaches during Design of Distributed Embedded Computing Systems”, *Proc. WORDS ‘03F (IEEE CS 9h Workshop on Object-oriented Real-time Dependable Systems)*, Capri Island, Italy, Oct. 2003, pp.307-313.

[Kop97] Kopetz, H., '*Real-Time Systems: Design Principles for Distributed Embedded Applications*', Kluwer Academic Publishers, Boston, 1997.

[Li95] Li, Y.-T. S., and Malik, S., "Performance analysis of embedded software using implicit path enumeration", *Proc. 32th Design Automation Conference*, 1995, pp. 456–461.

[OMG02] Object Management Group, '*UML Profile for Schedulability, Performance, and Time Specification*', March 2002, http://cgi.omg.org/docs/ptc/02-03-02.pdf.

[OMG05] Object Management Group, "Real-time CORBA Specification, Version 1.2", http://www.omg.org /technology/documents/formal/real-time_CORBA.htm, January 2005.

[Par93] Park, C.Y., "Predicting program execution times by analyzing static and dynamic program paths", *Real-Time Systems, v.5 n.1*, March 1993, pp.31-62.

[Pus97] Puschner, P. and Schedl, A., "Computing Maximum Task Execution Times – A Graph-Based Approach", *Real-Time Systems*, Vol. 13, No. 1, pp. 67-91, July 1997.

[Pus02] Puschner, P., and Burns, A., “Writing Temporally Predictable Code”, *Proc. WORDS ’02 (IEEE CS 7th Workshop on Object-oriented Real-time Dependable Systems)*, 2002, pp.85-91.

[Ros02] Ross, S. M., '*Introduction to Probability Models', 8th edition*, Academic press, 2002.

[Sel02] Selic, B., "The real-time UML standard: definition and application", *Proc. 2002 Conference on Design, Automation and Test in Europe*, pp. 770 -772.

[Sha89] A.C. Shaw. "Reasoning About Time in Higher-Level Language Software," *IEEE Trans. on Software Engineering*, vol. 15, no. 7, July 1989, pp. 875-889.

[Wen05] Wenzel, I., Rieder, B., Kirner, R., and Puschner, P., "Automatic Timing Model Generation by CFG Partitioning and Model Checking", *Design, Automation and Test in Europe (DATE'05)*, Volume 1, 2005, pp. 606-611.

[Wil05] N. Williams, “WCET measurement using modified path testing”, 5th Int'l workshop on Worst-Case Execution Time analysis (WCET05), Spain, July 2005.