Measurement Techniques in a Hybrid Approach for Deriving Tight Execution-time Bounds of Program Segments in Fully-featured Processors

Juan A. Colmenares*
UC Irvine, USA
jcolmena@uci.edu

Chansik Im
Intel Corporation, USA
chansik.im@intel.com

K. H. (Kane) Kim
UC Irvine, USA
khkim@uci.edu

Raymond Klefstad
UC Irvine, USA
klefstad@uci.edu

Chae-Deok Lim
ETRI, Korea
cdlim@etri.re.kr

Abstract

A hybrid approach to deriving tight execution-time bounds of program segments was proposed very recently. This approach symbiotically combines analytical and measurement-based methods to find a tight execution-time bound falling between the maximum measured execution time and an analytically derived loose bound. It also enables the estimation of the probability of the derived tight bound not being exceeded at run time.

This paper provides a refined description of the hybrid approach and presents techniques for measuring the execution times of acyclic-path segments (APSs), which are possible execution sequences of instructions that contain no cycles and the basic units of analysis in the hybrid approach. In this paper, we also report the results of the hybrid approach in the derivation of tight execution-time bounds of three algorithms frequently used in the evaluation of WCET analysis techniques.

1 Introduction

Demands for large-scale real-time distributed computing (RTDC) systems are steadily growing, even in safety-critical applications. These demands are the major driver for the development of a rigorous, broadly applicable method for determining *execution safety* of RTDC systems (i.e., absence of possibility of violating hard deadlines) [16].

Determination of the execution safety of an RTDC system involves analysis and validation of the timing behavior of the system's software components and their interactions, including coordinated sharing of limited resources. A key requirement here is the ability to obtain acceptable and credible upper bounds on the system's service times. However, the problem of deriving tight *service-time bounds* (or equivalently, guaranteeing response times) of large-scale RTDC systems becomes highly complex because such systems are frequently built in multiple layers (e.g., hardware subsystems, operating system (OS) kernels, communication protocols, middleware, and application software) and each layer contains interplaying factors that influence overall timing behavior. Moreover, even the derivation of tight *execution-time bounds* (ETBs) of simple *program-segments* (i.e., non-preemptable code sequences not including OS and middleware calls) [18, 17, 13] is difficult because the acceleration features present in modern high-performance microprocessors (e.g., Intel Pentium and multi-core processors) make the execution time of an instruction strongly dependent on surrounding instructions and the microprocessor's state.

The research community has recognized the timeliness assurance problem as one of the biggest challenges in the area. The state-of-the-art related to it, however, remains substantially short of being applicable to realistic RTDC application systems.

1.1 Related Work

Industry practice in deriving service-time bounds of RTDC systems has been to fully rely on a limited number of test runs with randomly and massively generated input data sets that may not necessarily cover the worst-case execution scenario. Thus, a margin value needs to be added to the maximum service time observed during the test runs to derive a credible bound. These approaches need to be optimized much further. Moreover, a reasonable estimation of the probability of the margin being insufficient at run time should also be available.

On the other hand, static worst-case execution time (WCET) analysis approaches [26, 11, 8, 20, 21], which have been studied considerably for the basic research community, often turn out to be very pessimistic (i.e., they produce excessively loose bounds) when applied to modern fully-featured processors, even when subject programs are

*Also with the Applied Computing Institute, School of Engineering, University of Zulia.

1080-1812/08 $25.00 © 2008 IEEE
DOI 10.1109/RTAS.2008.35

of single-layer type (i.e., not involving OS kernel and middleware calls). This fact has served as the motivation for seeking ways to substantially improve measurement-based approaches practiced in industry for deriving tight ETBs.

Some measurement-based approaches [19, 27, 28] focus on maximizing the probability of covering the worst-case execution path, and others [23, 4] on conditioning the execution environments in order to obtain more narrowly varying execution-time measurements of program-segments than the measurements produced in normal environments. In a complementary direction, Edgar and Burns [1] estimate the confidence that can be given to derived time bounds of simple, monolithic programs by applying extreme value statistics on measured execution times of programs. Petters [23] extends that work to derive probabilistic time bounds of simple programs based on a convolution numerical solution for combining the extreme-value distributions of the execution times of program-segments. Bernat et al. [2] present a hybrid approach that uses a program-structure based schema for computing a distribution of estimated maximum execution times for a single-layer program from the collections of execution-time measurements of involved basic blocks (i.e., straight-line pieces of code with no jumps). The computation method involves convolution-based operations.

In this paper, we discuss a different hybrid approach [12, 13] that combines *measurements* with *loose but safe ETBs*, obtained from static WCET analysis, in order to produce tight ETBs of program-segments and determine the levels of execution safety of these tight ETBs.

1.2 Contribution

To facilitate the timing analysis of large-scale RTDC systems and cope with their inherent structural complexity, it is essential to apply *divide-and-conquer strategies* [16] that permit the *determination* and *stepwise integration into service time bounds* of: 1) *tight ETBs of program-segments*, and 2) *tight bounds on the completion times of OS kernel and middleware service calls*, including calls involving the communication infrastructure.

This paper focuses on the first point. Here we provide a refined description of a novel hybrid approach, originally proposed by Im and Kim [12, 13], for deriving non-preemptive tight ETBs of program-segments involving no OS kernel and middleware service calls. This hybrid approach has major distinguishing features summarized as follows [12]:

- Execution-time measurements and an analytically derived loose ETB of a given program-segment are used together for determining the safety margin to be added to the maximum observed execution time to produce a tighter ETB for the program-segment. The approach also enables estimation of the probability of the tighter ETB not being exceeded at run time. During the analysis, a *curve-fitting technique* is applied to relate the measured data to the analytic loose ETB.
- An analysis technique, called the *APS-based Path Enumeration Technique* (APET) [12], is applied to produce the analytical ETB of the program-segment. APET extends the Implicit Path Enumeration Technique (IPET) [20]. Unlike IPET that uses basic blocks as basic units in the WCET calculation, APET uses larger units called *acyclic-path segments* (APSs), which are possible execution sequences of instructions with no cycles. Since APSs contain partial execution history information, they allow for derivation of tighter ETBs of program-segments than the ETBs that can be obtained from the use of basic blocks. Due to its inherent pessimism, APET produces loose bounds that are considered *hard ETBs* (i.e., the probability of violating the bounds is near zero), but those loose bounds can be tightened by reflecting measurement results.
- In conducting measurements, the approach is aimed at exercising the most promising execution paths for exhibiting the worst execution time. In identifying such promising candidates for the worst-case execution path (WCEPath) of a program-segment, we apply APET multiple times. In each iterative step of APET, the previously identified WCEPath candidate is treated as an infeasible execution path.

APSs are the basic units of analysis in the hybrid approach. Thus, the quality of the analytical bound of a program-segment obtained with APET heavily depends on the ETBs of the APSs. Due to the absence of reliable up-to-date simulators of high-end microprocessors, the derivation of a tight ETB of an APS must only rely on extensive measurements of the execution time of the APS. Therefore, execution-time measurements of the APSs must be taken carefully and as precise as possible, considering the performance-enhancement features of the target processor.

In this paper, we discuss the key aspects that must be considered and present several techniques for measuring the execution times of APSs on fully-featured microprocessors. We also report the results of applying the hybrid approach in the derivation of tight ETBs of three algorithms frequently used in the evaluation of WCET analysis techniques. Additionally, we empirically demonstrate the advantage of using APSs instead of basic blocks in deriving analytical ETBs of program-segments executing on fully-featured microprocessors.

1.3 Paper Organization

This paper is structured as follows. Section 2 describes the hybrid approach. Section 3 presents the adopted measurement-based techniques for deriving the ETBs of APSs to be used for obtaining the analytical bound of a given program-segment. The APS Analyzer, our prototype tool for identifying APSs and deriving their ETBs through measurements, and other tools used for implementing the hybrid approach are presented in Section 4. The experimental results are discussed in Section 5. Finally, a conclusion is drawn in Section 6.

2 Description of the Hybrid Approach

In this paper, a *program-segment* is a sequence of code that contains no calls for operating system (OS) and network communication services [18, 17, 13]. Furthermore, we assume that program-segments execute in isolation, i.e., they are neither affected by background activities nor subject to preemption. Hence, the effects of the activities of the OS, including message communication, need not be considered in deriving tight *execution-time bounds* (ETBs) of program-segments.

ETB analysis approaches (e.g., [22, 20, 24, 21, 7, 26, 9]) involve: i) analysis of the logical structure of a given program, and ii) application of carefully established ETBs of instructions or execution primitives of the underlying physical or virtual machine. Due to conservative estimates frequently employed in analytical approaches, the resulting ETBs of program-segments tend to have large error margins. Consequently, analytically derived ETBs of program-segments are considered *hard bounds* with a *practically negligible probability of being violated at run time* [17]. In measurement-based approaches for deriving ETBs of program-segments (e.g., [1, 5, 23, 2]), determination of the safety margin to be added to the maximum among the measured execution times is a core area for improvement. If the adopted safety margin is not sufficiently large, then the resulting ETB becomes a *soft bound*, which is associated with a *non-negligible probability of being exceeded at run time* [17]. Therefore, the basic idea in the hybrid approach discussed in this paper is to *find an ETB falling between the maximum measured execution time and an analytically derived loose bound*. Thus, analytical methods and measurement-based methods are combined in a symbiotic form for deriving tight ETBs of program-segments.

The hybrid approach [12, 13] includes the following steps:

1. A given program-segment PS_i is divided into *acyclic-path segments* (APSs), where each APS is *a possible execution sequence of instructions that contains no cycles*. The assembly code of each APS is slightly manipulated to enable its execution as a stand-alone code unit while preserving its timing behavior as much as possible. Then, tight ETBs of the APSs are determined through extensive measurements.
2. A hard ETB of PS_i is obtained by solving an integer linear programming (ILP) problem formulated based on the tight ETBs of the APSs (derived in Step 1) and the control flow among the APSs. This technique, called the *APS-based Path Enumeration Technique* (APET) [12], is an extension of the Implicit Path Enumeration Technique (IPET) [20].
3. The optimal solution of the ILP problem of Step 2 (i.e., the execution counts of the APSs that produce a hard ETB of PS_i) may be considered as a candidate for the worst-case execution path (WCEPath) of PS_i. Then, a new variation of the ILP problem in which the previously identified WCEPath candidate is treated as an infeasible path is formulated and solved. As a result, another WCEPath candidate is obtained. This procedure is repeated a few times to produce multiple candidates for the WCEPath of PS_i.
4. After verifying the feasibility of the WCEPath candidates, the execution time of each feasible WCEPath candidate is measured during a number of test runs with generated input data sets. The candidate with the maximum observed execution time is selected for further analysis.
5. Finally, we identify a smooth model that approximates the cumulative distribution function (CDF) of the measured execution times of the selected WCEPath candidate (Step 4), augmented with the hard ETB of PS_i (Step 2). Using this model, we can determine soft, tighter ETBs of PS_i and estimate the probability of such ETBs not being exceeded in runtime.

The following sections discuss our hybrid approach in more detail.

2.1 Analytic Derivation of Loose Execution-time Bounds of Program Segments

Modern high-end microprocessors (e.g., Intel Pentium and multi-core processors) contain sophisticated acceleration features that enable better average performance (e.g., pipelining, out-of-order execution, instruction and data caches, and branch prediction). Due to these features, the execution time of an instruction depends on surrounding instructions as well as the microprocessor's state when the instruction is being executed.

ETB analysis approaches usually represent a program-segment as a *directed graph* (i.e., a control flow graph (CFG)), and each node of this graph is a *basic block* that consists of a sequence of consecutive statements in which the control flow enters at the beginning and leaves at the end without halt or possibility of branching before the end. Moreover, ETB analysis approaches typically consider *basic blocks as the basic units of timing analysis* (e.g., [20, 7, 9]). But basic blocks are usually small code units and thereby, their individual use in capturing the effects of the acceleration features of modern processors is very limited.

A number of ETB analysis approaches (e.g., [22, 21, 6, 7, 3, 10]) incorporate models of performance enhancing features of processors to reflect the effects of those features on the WCET estimates. These model-based approaches produce ETBs much tighter than those obtained under conservative assumptions on how the acceleration techniques affect the execution time of the program (e.g., memory access always causes cache misses, instructions are executed in order, and branch prediction always fails). Unfortunately, these approaches are applicable to rather simple microprocessors only. Also, they frequently assume very little or no interdependence among the acceleration features and

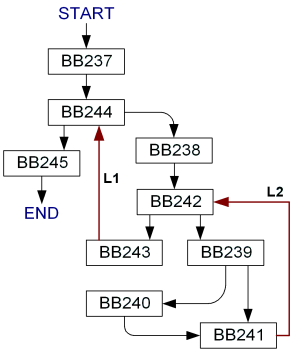


Figure 1. Control flow graph of bubble sort.

analyze them separately; assumption that is not valid in general for modern microprocessors [11].

To our knowledge, sufficiently validated models of high-end microprocessors are not publicly available. Hence, currently it is not possible for practitioners to apply model-based approaches to derive tight ETBs of program-segments executing on fully-featured microprocessors. Furthermore, construction and validation of models of this type of microprocessors are complex, time consuming, and costly tasks that require detailed knowledge on the implementations of the performance enhancing features. Thus, we do not expect that this situation will change in the near future.

Acyclic-path Segments (APSs)

To better reflect the characteristics of fully-featured microprocessors, we divide the program-segments into sequences of basic blocks, called *acyclic-path segments* (APSs). An APS is a *possible execution sequence of instructions that contains no cycles* [12, 13]. Since APSs contain partial execution history information, they allow for derivation of tighter analytical ETBs than those obtained using basic blocks as basic units of timing analysis. Thus, in this work APS is the *minimum unit of analysis and measurement*.

APSs are obtained by analyzing the assembly code of program-segments. The starting point of an APS is either the entry point of a program-segment or the head of a loop inside the program-segment, and its ending point is either a loop-backward branch or the exit point of the encompassing program-segment. Although there can be a number of different procedures for identifying the APSs of a program-segment, in this work we use a simple heuristic based on a depth-first search procedure that seals an APS whenever either the boundary of a program-segment or a loop-backward branch is encountered. For example, Table 1 presents the APSs identified from Figure 1, which represents the CFG of the algorithm bubble sort. The edges `L1` and `L2` represent the outer and inner loops of bubble sort.

Due to the absence of reliable up-to-date simulators of high-end microprocessors, the derivation of a tight ETB

Table 1. APSs identified from Figure 1.

APS	ID	Sequence of basic blocks	Connects to
1	#657	237-244-245	END
2	#658	237-244-238-242-239-241	242
3	#659	237-244-238-242-243	244
4	#660	237-244-238-242-239-240-241	242
5	#661	244-245	END
6	#662	244-238-242-239-241	242
7	#663	244-238-242-243	244
8	#664	244-238-242-239-240-241	242
9	#665	242-239-241	242
10	#666	242-243	244
11	#667	242-239-240-241	242

of an APS must rely on extensive measurements of the execution time of the APS. These measurements must be taken carefully and as precise as possible, considering the performance-enhancement features of the target processor. Thus, special techniques for measuring the execution times of ASPs (such as those described in Section 3) must be applied. In any case, the ETB of an APS can be obtained in a much tighter form through such effort rather than through mere summation of the execution-time estimates of the constituent basic blocks. Moreover, tighter analytical ETBs of program-segments are obtained when the analysis is based on APSs instead of basic blocks. This claim is empirically demonstrated in Section 5.

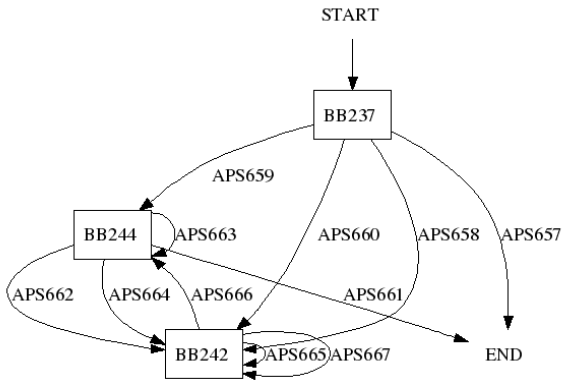


Figure 2. APS graph of bubble sort.

APS-based Path Enumeration Technique (APET)

Once the APSs of a program-segment are identified, an APS graph (APSG), which represents the control flow among APSs, is built. For example, Figure 2 shows the APSG of the bubble sort. Each edge of the APSG represents an APS, and each vertex represents the first basic block of one or more APSs. APSs that start with the same basic block are outgoing edges of the same vertex.

Then, an *integer linear programming* (ILP) problem is formulated based on the APSG, the CFG, and the obtained ETBs of the APSs of the program-segment. By solving

this optimization problem, we obtain the analytical (i.e., hard) ETB of the program-segment. This extension of IPET [20] is called the *APS-based path enumeration technique* (APET) [12].

More formally, let x_i be the number of times an acyclic-path segment APS_i is executed, and let e_i be the derived ETB of APS_i. Provided that there are N APSs in the program-segment, the general structure of the APET problem is as follows:

$$Maximize \sum_{i=1}^{N} e_i x_i$$
$$subject\ to\ constraints\ of\ the\ form:$$
$$\mathbf{G_j}: \sum_{i=1}^{N} p_i x_i = \{0 \mid 1\};\ where\ p_i = \{-1, 0, +1\}$$
$$\mathbf{H_k}: x_i \leq lu_i$$

Linear constraints of the first type (i.e., $\mathbf{G_j}$) reflect the structure of the program-segment. They can be automatically derived from the APSG following a basic principle: *given a vertex in the APSG, the sum of the execution counts of the incoming APSs is equal to the sum of the execution counts of the outgoing APSs*. On the other hand, a constraint of the second type (i.e., $\mathbf{H_k}$) is a basic functional constraint that usually specifies the *upper bound* lu_i of the times that APS_i executes. The functional constraints are obtained via flow analysis. This includes, for example, determining how many times loops iterate and identifying infeasible sequences of basic blocks and thereby, infeasible APSs. In particular, early detection of infeasible APSs typically results in important reduction of the analysis and measurement efforts. For example, we present below the general APET problem for bubble sort:

$$\textbf{Maximize}$$
$$e_{657}x_{657} + e_{658}x_{658} + e_{659}x_{659} +$$
$$e_{660}x_{660} + e_{661}x_{661} + e_{662}x_{662} + e_{663}x_{663} +$$
$$e_{664}x_{664} + e_{665}x_{665} + e_{666}x_{666} + e_{667}x_{667}$$
$$subject\ to\ the\ \textbf{constraints}:$$
$$x_{666} + x_{659} = x_{661} + x_{662} + x_{664}$$
$$x_{662} + x_{664} + x_{658} + x_{660} = x_{666}$$
$$x_{657} + x_{658} + x_{659} + x_{660} = 1$$
$$x_{661} + x_{657} = 1$$
$$x_{666} = 1000$$
$$x_{667} \leq 500500 - x_{660} - x_{664}$$
$$x_{657} = x_{658} = x_{659} = 0$$
$$x_{662} = x_{663} = x_{665} = 0$$

This problem was formulated assuming that the input is an array of 1000 elements reversely sorted (i.e., the worst case for bubble sort). The first four constraints are structural constraints and the rest are functional constraints. Note that 6 APSs out of 11 ($\approx 54\%$) are infeasible; they are: #657, #658, #659, #662, #663, and #665. The APS #657, for example, represents the case in which the algorithm ends without entering the loops even once. Clearly this cannot happen because the input array is not empty. In the APSs #659 and #663 the algorithm does not enter the inner loop (`L2`). This situation cannot happen either since all the elements in the array are reversely sorted. Similar percentage of infeasible APSs were found in other test functions (e.g., matrix multiplication and LU decomposition).

Once each term e_i is substituted by the derived ETB of APS_i, the APET problem can be easily solved using an ILP solver tool (e.g., `lp_solve`[1]). The ILP solver tool returns the hard ETB of the program-segment (i.e., the optimal value) and the execution counts of the APSs that yield such bound (i.e., the optimal solution).

APET can, however, produce solutions that correspond to infeasible execution paths because some functionality constraints may not have been included in the formulation of the ILP problem [20, 3]. Hence, system designers may decide to validate whether an APET solution is feasible or not. If not, they may try to determine additional functionality constraints to rule out such infeasible paths. Flow analysis is often considered the first choice for deriving such functionality constraints. In any case, both validating execution counts and correctly determining additional functionality constraints may require meticulous analysis of the program-segment and thus, considerable effort.

2.2 Identification of Promising Path Candidates

It is possible that the optimal solution for the APET problem (i.e., the execution counts of the APSs that maximize the objective function) does not correspond to the the actual *worst-case execution path* (WCEPath) of the program-segment. The reason is that the execution time of a path formed by a series of APSs is not as accurately estimated as that of an individual APS, in particular when the path is part of a loop. In that case, it is wise to identify the APS execution counts of multiple WCEPath candidates and then examine their execution times through various means including not only analysis but also measuring [12, 13]. This results in a better-quality tight ETB for a given program-segment.

Multiple candidates can be easily identified by iteratively applying APET as follows. After formulating the APET problem for a program-segment, the ILP solver tool finds the analytical bound b_1 and the optimal solution $\{S_1\}$; they both form the tuple $(b_1, \{S_1\})$ that represents the first WCEPath candidate. Then, the constraint $\sum_{i=1}^{N} e_i x_i \leq (b_1 - \delta)$, where δ is a small number provided by the system designers, is added to the APET problem and the ILP solver finds the second WCETPath candidate $(b_2, \{S_2\})$. This process continues until the specified stop criterion is met, for example: i) an absolute number of candidates are

[1] `http://sourceforge.net/projects/lpsolve/`

found; or ii) b_j, which is the latest WCET obtained by the IPL solver, goes below the threshold th given by the system designers; formally, $b_j \leq th \cdot b_1$ $(0 < th < 1)$. The definition of the stop criterion heavily depends on the application, and system designers may use one of the aforementioned ways or a combination of them to specify when to stop searching for WCEPath candidates of a given program-segment.

2.3 Determination of Safety Margins and Tighter Execution-time Bounds

Since APET may produce overly pessimistic ETBs, a testing-based statistical approach is introduced as a complement to APET. In this work, we take a simple approach in which the execution times of the identified WCEPath candidates are measured during a number of test runs with generated input data sets on the target execution platform.[2] Then, the candidate with the maximum measured execution time is selected for further analysis. Thereafter, this maximum value plus some margin will be considered as another ETB of the program-segment. The next questions are then [12]:

- How to determine the margin value to be added to the maximum measured execution time of the selected WCEPath candidate of the program-segment?
- How to estimate the probability of such a soft ETB being violated at runtime?

Assume the sequence $T = t_1, t_2, \cdots, t_i, \cdots, t_N$ contains the measured values of the execution time of the selected candidate, sorted in non-descending order. We can derive the cumulative distribution function (CDF) of the measured values using the expression:

$$F_c(t) = P(X \leq t_i)_{1<i<N} = n(t \leq t_i)/N$$

where $n(t \leq t_i)$ is the number of measurements that are less than or equal to t_i, and N is the number of measured values. Then, the analytical bound for the program-segment (obtained from APET) is appended to the CDF data and assigned a cumulative frequency of 1. Finally, a curve-fitting technique (based on nonlinear regression) is applied in order to obtain a smooth model that approximates *the augmented CDF* [12, 13].

Once identified, the approximation model can be used to answer the two previous questions. For example, consider the frequency distribution of 100 measurements of the execution time of a WCEPath candidate shown in Table 2, and assume that the analytical ETB is 250 ms. A model that approximately fits the augmented CDF is given by:

$$\alpha = \frac{1}{\left(1 + e^{(4.9196 - 0.0552 \cdot t)}\right)^{1/0.2626}}$$

This sigmoidal model is called Richards model [25] and its coefficients were obtained using a curve-fitting tool (e.g., `CurveExpert`[3]). Figure 3 shows the CDF data, the augmented CDF used for estimating the approximation model, and the resulting model. The model's saturation point is approximately 200 ms, which means that the probability of the actual execution time not exceeding 200 ms becomes practically one (i.e., α=0.9917). In addition, the model enlightens a way of estimating a soft ETB when the probability of the actual execution time not exceeding this soft bound, α, is provided. For instance, if α=0.95, the estimated ETB is 167.03 ms, and if α=0.98, the estimation becomes 183.98 ms.

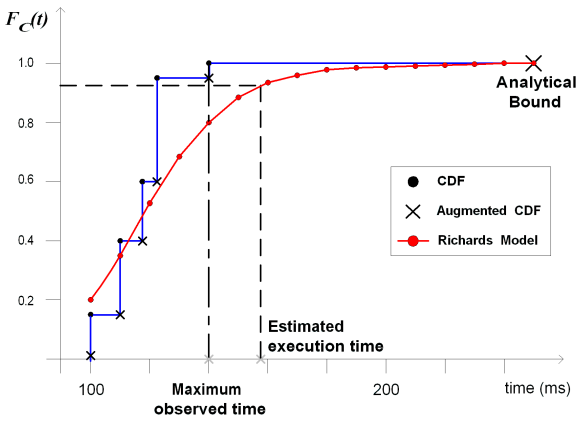


Figure 3. Applying the curve-fitting technique for estimating a soft execution-time bound of a program-segment.

Table 2. Relative and cumulative frequencies of the measured execution times of a WCEPath candidate

Execution time (ms)	Relative Frequency	Cumulative Frequency
100	0.15	0.15
110	0.25	0.40
118	0.20	0.60
122	0.35	0.95
140	0.05	1.00
250	Analytical execution-time bound	

In practice, application designers need to try with a number of models and then select the model that best fits the augmented CDF of measured execution times. Moreover, the approximation model identified using the curve-fitting technique may go beyond or stay below 1; for instance, Richards models often overstep 1. Hence, when the model does not converge to 1, it must be normalized so that the approximating curve is always below the CDF curve and saturated before the analytical bound derived from APET.

Application designers must select the value of α very carefully when determining a soft ETB of a program-segment. If α is not sufficiently high, there will be a non-negligible risk of the derived soft bound being exceeded by the actual execution time.

[2] Issues related to obtaining appropriate input data sets are not discussed here due to space limitations.

[3] `http://curveexpert.webhop.biz/`

3 Derivation of Execution-time Bounds of Acyclic-path Segments

The analytical ETB of a program-segment obtained with APET heavily depends on the ETBs of the acyclic-path segments (APSs). Therefore, it is very important to estimate the ETBs of the APSs from execution-time measurements that have been taken carefully and as precise as possible. Nevertheless, some overhead in the measurements is inevitable.

This section presents techniques for measuring the execution times of APSs. These techniques are specific for Intel processors; however, several principles discussed here can also be applied to measuring execution times in other processors. Moreover, the description of the techniques is in the context of using a Linux-based operating system.

3.1 Code Translation

In order to facilitate the execution of a specific APS, one needs to generate input data sets and identify at least one set that makes the control flow take the execution path of the APS [19, 27, 28]. The input data generation approach requires relatively simple instrumentation for both path detection and execution-time measurement and typically results in very little measurement overhead. However, very often finding the data sets for some particular APSs is difficult and many tests need to be performed. Instead, we use a different technique that eliminates the need for massive input data search efforts, though with a more complex implementation.

In our technique, called *code translation* [12], the basic blocks that compose an APS are extracted from the program-segment and inserted into the body of a procedure. Because the basic blocks of the APSs are taken out of their "original context" (i.e., the program-segment), the assembly code of the basic blocks is manipulated and a minimal number of instructions are added to facilitate the execution of those basic blocks in the order defined by the APS. For example:

- Each conditional branch instruction (i.e., `Jxx`) is replaced by the instruction `cmpl %eax,%eax` followed by either `je` or `jne`, depending if the branch must be taken or not.
- Since APSs are partial execution sequences, an APS may not include matching call-stack handling instructions; in this case, those instructions must be added.
- To prevent instructions from resulting in illegal memory access, memory references are redirected to predefined memory locations; for example, the instruction `movl (%eax),%edx` is replaced by the following line in GCC inline assembly:

```
asm("movl %0,%%edx\n\t" :: "m" (MEM_BLOCK
    [2492]));
```

In the present work *memory references are redirected randomly to reserved memory space, but in such manners that two instructions in an APS never access the same memory location*. By doing this, memory operations are likely to result in cache miss, which increases the time that APSs take to execute. Nevertheless, there is room for incorporating more sophisticated techniques that allow us to better capture the effects of data cache on the execution of the APSs, and thus, obtain tighter ETBs. For example, it is possible to make the addresses used by instructions fall into memory areas while preserving the arithmetic operations used to obtain the actual memory addresses.

3.2 Measuring the Execution Times of Acyclic-path Segments

Once an APS is manipulated and put into the body of a procedure, a test program executes the procedure many times in a loop and measures the machine cycles consumed by each execution of the APS. The number of machine cycles consumed during the execution of an APS can be determined by: i) reading the timestamp counter[4] (TSC) twice, right before the APS starts and right after it ends; and ii) subtracting the former TSC read from the latter [14]. Then this difference can be divided by the CPU's clock frequency to yield the elapsed time between the two measurement points. In addition, due to out-of-order execution capability of fully-featured microprocessors, we need to ensure that *all* and *only* the instructions of the APS under test are executed within the measurement window (i.e., between the TSC reads). Therefore, a serializing instruction (e.g., `cpuid`) must be executed immediately before each TSC read. Since `cpuid` can take longer the first couple of times it executes [14], `cpuid` must be called several times before taking the measurements. The basic structure of the code for measuring the execution times of APSs is shown in Listing 1.

Because program-segments are assumed to *execute in isolation*, we need to guarantee that the measurements of the APSs are taken under the same condition. To prevent execution of APSs from being preempted and affected by interrupt handling, we take advantage of several features present in Linux, namely:

- The test program is run by the superuser as a FIFO real-time process (i.e., using the `SCHED_FIFO` scheduling policy) with the highest priority. Thus, the test program is never preempted when executing an APS.
- Hardware interrupts (except the timer interrupt) are disabled (using the instruction `cli`) before each execution of an APS and re-enabled (using the instruction `sti`) once the execution of the APS has finished (see Listing 1). To be able to call `cli` and `sti`, the test program changes its I/O privilege level to 3 by calling the Linux-specific system call `iopl`.

Additional actions can be taken to put the processor's acceleration features in an unfavorable state before measur-

[4] The timestamp counter is a 64-bit register that counts the CPU clock cycles since power on.

```
void runAPS(void (*fn)(void), ...) {
  unsigned long long
    measurements[NUM_OF_MEASUREMENTS];
  . . .
  for (int i = 0; i < NUM_OF_MEASUREMENTS; i++) {
   unsigned long long t0 = 0, t1 = 0;
   . . .
   asm("cli \n\t");
   asm("cpuid \n\t");
   rdtscll(t0);
   (*fn)(); // Function containing the APS
   asm("cpuid \n\t");
   rdtscll(t1);
   asm("sti \n\t");
   . . .
   measurements[i] = t1 - t0;
   . . .
}}
```

Listing 1. Code structure for measuring the execution time of an acyclic-path segment.

ing the machine cycles consumed by the execution of an APS [23]. Those actions result in more conservative measurements.

It is well known that data and instruction caches have very strong influence over the execution time of programs on modern processors. We have several options for incorporating cache effects into measurements of APSs; before timing begins we can: i) *disable data and instruction caches* (after invalidating them), ii) *fill the caches with useless data*, iii) invalidate the caches, and iv) *do nothing*, of course.

To be able to invalidate, disable, and re-enable the caches from user space, we added several simple system calls to the Linux kernel. Caches are invalidated by calling the instruction `wbinv` (via the Linux internal function `wbinv()`). Caches are disabled in Intel processors by setting the 31th bit (CD, Cache Disable) of the control register `CR0` (i.e., `CR0[bit 30] = 1`). To re-enable the caches `CR0[bit 30]` is reset.

Intel processors are typically configured to use the write-back data caching strategy (i.e., `CR0[bit 29] = NW = 0`)[5]. In this case, the L1 data cache and L2 (unified) cache must be filled with useless modified data. Thus, when new data is brought into the cache, the old data in the cache is written to main memory, and not simply replaced in the cache. Filling the L1 data cache and the L2 unified cache with modified useless data can be done from user space. We allocate a memory space (i.e., an array) equal to the size of the L2 unified cache (i.e., 256 Kbytes in Pentium III (Coppermine) processors). Then, after invalidating the caches, memory locations from that space are read, and different values are written back to them.

The branch-prediction mechanism can also be preset in an unfavorable state prior to the measurements. We can run a piece of code specially devised to fill the Branch Target Buffer (BTB) with useless data before each execution of an APS. Thus, the mechanism is not able to dynamically predict the direction of the branches and must rely on the static strategy. The static branch-prediction mechanism in Pentium III processors always predicts forward branches as "*not taken*". Hence, during the manipulation of assembly code of the APSs conditional branches are converted in forward branches. In newer processors (i.e., Intel Core Duo), using forward branches, however, may have no effect because these processors alway predict conditional branches dynamically, even at first appearance [15].

These techniques should also be applied to measure the execution times of program-segments.

[5]Not Write-Through (NW)

3.3 Estimation of Execution-time Bounds of Acyclic-path Segments

After measuring extensively the execution times of the APSs, we need to decide which values from the measured data will be used as the execution-time bounds (ETBs) of the APSs in deriving the analytical ETB of the program-segment. To make that decision, we should have (definitive or preliminary) measurements of the execution time of the program-segment and compare the measured values with the analytical bound obtained with APET using some tentative ETBs of the APSs. If the analytical bound of the program-segment is too pessimistic, we may decide to use tighter estimates of the ETBs of the APSs. On the contrary, if the analytical bound is considered unsafe, then we may decide to use looser estimates of the ETBs of the APSs. In this way, the decision can be made systematically and supported by empirical data. Note that the measurements of the execution times of the APSs and the program-segment must be taken under the same conditions in order to have a meaningful comparison between the measured execution time of the program-segment and its analytical bound obtained with APET.

We have found from our experience that using the maximum among the measured execution times of each APS as the ETB of the APS may be sufficient to obtain a safe analytical ETB for the program-segment, given the inherent pessimism of this analytical technique. Moreover, the maximum measured execution time of an APS very often corresponds to the first observation and its value is significantly higher than the rest of the observations in the same measured data set. In this case, the maximum observed value should be dropped and the second maximum observed value should be used as the ETB of the APS. Even more, in our experience safe analytical ETBs of program-segments can be obtained when measurements of the execution times of the APSs are taken without controlling the caches. We will support these observations with experimental data in Section 5.

4 Prototype Implementation

We have developed a new version of the *APS Analyzer* for Linux. This C++ prototype tool takes as input the names of the C/C++ source code files of a program and the functions and class methods to be analyzed. It uses `gcc` to generate assembly code from the program source code, and constructs the call graph and the control flow graphs (CFGs) of the functions and methods specified in the input. Next, the APS Analyzer identifies the program-segments and their APSs, and builds the APS graphs (APSGs). Finally, it manipulates the APSs (as described in Section 3.1) and generates a test program for measuring the execution times of the APSs. The APS Analyzer also generates intermediate text files that represent the call graph, the CFGs, and the APSGs in *DOT* language.[6]

We use `lp_solve` for solving the APET optimization problem, and CurveExpert 1.3 as a curve fitting tool.

To facilitate the formulation of the APET problem, we have also developed a tool, called *APET-Formulator*. This tool reads the description of an APSG in *DOT* language and the files with the measured data of the execution times of the APSs, and then creates an `lp_solve` file with the objective function and the structural constraints of the APET problem; although the functional constraints must be added manually.

5 Experimental Results

Test Platform: The measurements were performed on an 866-MHz Pentium III (Coppermine) with 512-MB RAM, 256 KB L2 cache, and bus speed of 133 MHz running a slightly customized version of Linux 2.6.21.1. A few simple system calls were added to Linux to make it easy to invalidate, disable, and re-enable the caches from user space (Section 3.2). The CPU frequency reported by the system is 864.617 MHz; hence, this value is used for converting cycle counts into time units.

5.1 Experiment 1

This experiment was designed to empirically demonstrate the advantage of using APSs rather than basic blocks in deriving analytical ETBs of program-segments executing on fully-featured microprocessors.

We took the APS corresponding to the innermost loop of the bubble sort that contains the swap operation; that is, the APS #667 of Table 1. The APS #667 contains the following sequence of basic blocks: BB242, BB239, BB240, and BB241. When the input is an array of 1000 elements reversely sorted, the APS #667 runs 499500 times. That basic-block sequence, however, is executed 500500 times since it is also contained in the APSs #660 and #644.

[6]The *DOT* language is the text language used by the open-source graph visualization software *Graphviz* for representing graphs (`http://www.graphviz.org`).

We measured the execution time of each basic block and the entire APS #667 under 4 different scenarios:

1. with the cache mechanism disabled,
2. with the cache mechanism enabled but filling (i.e., polluting) the caches with useless data before each execution of the basic blocks and the APS,
3. with the cache mechanism enabled but invalidating the caches before each execution of the basic blocks and the APS, and
4. with the cache mechanism enabled and without taking any further action (i.e., non-controlled cache)

When enabled, the cache mechanism used the write-back strategy. The basic blocks and the APS were inserted directly between two `cpuid`/`rdtsc` instruction pairs in order to minimize measurement overhead. Additionally, another set of measurements were taken with the APS inserted in a procedure (APS in-proc), which is the way we normally measure the execution time of APSs.

In each case, we took 1000 execution-time measurements. In all the cases, the first measurement was dropped because it was considerably larger than the rest of the measurements. Thus, we use in our analysis the second maximum value of each measured data set.[7]

The results are presented in Table 3. We observe that *in all the cases the sum of the measured execution times of the basic blocks is larger than the measured execution times of both the bare APS and the APS inserted in a procedure*. Furthermore, that difference may be amplified many times during the derivation of the analytical bound. For example, in the case of the non-controlled cache scenario, the difference between the sum of the measured execution times of the individual basic blocks and the execution time of the APS #667 wrapped in a procedure is equal to 583 cycles (=1090-507). When multiplied by 499500 (the number of times APS #667 runs in the worst case) and divided by 864.617 MHz (the CPU frequency), it becomes 336.81 ms, which is 67% larger than the derived analytical bound of the bubble sort reported in Table 4 for the same scenario (i.e., non-controlled cache).

Therefore, we corroborated our conjecture that we can obtain tighter analytical ETBs of program-segments by using APSs, instead of basic blocks, as basic units of measurement and analysis.

5.2 Experiment 2

Here we present the results of applying the hybrid approach and the measurement techniques discussed in this paper in the derivation of ETBs for 3 test functions that implement the algorithms: bubble sort (`BSORT`), matrix multiplication (`MM`), and LU decomposition (`LUD`). These algorithms do not involve any OS service calls, thus they fit the definition of the program-segment given in Section 2.

In this experiment, the `BSORT`'s input consists of an array of 1000 integers reversely sorted, which corresponds to

[7]Nevertheless, we observed that the same conclusion reached here also holds for those very large measured values.

Cache	Disabled	Polluted	Invalidated	Non-controlled
BB242	2892	682	853	304
BB239	3802	776	887	303
BB240	5551	1221	1290	311
BB241	1690	585	796	172
$\sum BB_i$	**13935**	**3264**	**3826**	**1090**
APS (bare)	9321	1731	1602	422
APS (in-proc)	10341	1894	1771	507

All values are in machine cycles.

Table 3. Comparison between measured execution times of an acyclic-path segment and its constituent basic blocks.

its worst-case scenario. Therefore, the innermost loop of the algorithm, containing the swap operation, runs 500500 times. `MM` multiplies 2 20x20 matrices of integer values and `LUD` decomposes a 15x15 matrix of double values. The implementations of `MM` and `LUD` were obtained from the benchmark programs maintained by the Mälardalen WCET research group.[8] Only LUD was modified to make a function `inline`.

We first applied the *APS Analyzer* to `BSORT`, `MM`, and `LUD` and we obtained the test programs for measuring the execution times of the APSs. The execution times of the APSs were measured under 3 different scenarios[9]:

1. with caches disabled,
2. with caches enabled but filling (i.e., polluting) the caches with useless data before each execution of an APS, and
3. with caches enabled and without taking any further action (i.e., non-controlled cache)

When enabled, the cache mechanism used the write-back strategy.

The execution time of each APS was measured 1000 times and the maximum measured value was adopted as the ETB of the APS. In a number of cases, however, the maximum measured value of the execution time of an APS was the very first measurement and it was considerably larger than the rest of the measurements. We identified all these cases with our *APET-Formulator* tool and confirmed that in each case the first measurement was at least 4 times larger than the maximum of the rest of the measurements. In those cases we dropped the first measurement and adopted the second maximum measured value as the ETB of the APS.

Next, using APET we derived the analytical ETBs of `BSORT`, `MM`, and `LUD` for the 3 cache scenarios of above. In all the cases the optimal solutions obtained with APET were feasible execution counts of APSs. For `BSORT`, `MM`, and `LUD`, it is easy to determine input data sets that produce the execution counts of APSs corresponding to the analytical ETBs. Thus, there was no need to identify multiple candidates for the worst-case execution path. Instead, using

[8] Available at: `http://www.mrtc.mdh.se/projects/wcet/`

[9] We omitted the scenario where the cache mechanism is enabled and the caches are invalidated before each execution of an APS because the results are very similar to those of the polluted-cache scenario.

Bubble Sort (`BSORT`)			
Cache	*AB (ms)*	*MMVF (ms)*	*Ratio*
Disabled	6696.11	3763.47	1.78
Polluted	1451.62	17.20	84.41
Non-controlled	201.58	17.17	11.74
Matrix Multiplication (`MM`)			
Cache	*AB (ms)*	*MMVF (ms)*	*Ratio*
Disabled	105.01	56.11	1.87
Polluted	23.74	0.2529	93.85
Non-controlled	2.75	0.2348	11.72
LU Decomposition (`LUD`)			
Cache	*AB (ms)*	*MMVF (ms)*	*Ratio*
Disabled	19.55	9.13	2.14
Polluted	4.83	0.06537	73.87
Non-controlled	1.56	0.05416	28.9

AB: Analytical Bound
MMVF: Maximum Measured Value of the Function
Ratio = AB / MMVF

Table 4. Analytical execution-time bounds and maximum measured execution time of the test functions under different cache scenarios.

the chosen input data sets we directly measured the execution time of `BSORT`, `MM`, and `LUD`, 2000 times each, under the same 3 cache scenarios. The results are presented in Table 4.

The results show that all the derived analytical bounds are safe, although no safety margin was added to the ETBs of the APSs. Moreover, the analytical bounds in the non-controlled cache scenario are also safe when compared with the measured execution times in the polluted-cache scenario. Thus, the results suggest that the ETBs of APSs obtained in the non-controlled cache scenario may be used to derive safe analytical ETBs of program-segments with APET, given the inherent pessimism of this analytical technique. The pessimism introduced by APET is large in the polluted-cache scenario (with ratios larger than 70). In the non-controlled cache scenario, the analytical bounds are tighter but the pessimism of APET is still evident (in particular, for `LUD`). These results stress the need of a hybrid approach that allows us to derive tighter ETBs of program-segments in the context of fully-featured microprocessors.

Table 4 also shows that APET produced tight analytical bounds when caches were disabled. Unfortunately, the analytical bounds and the measured values obtained in this scenario are useless in practice because they are too large when compared with the measured execution times obtained in the polluted-cache and non-controlled cache scenarios. This confirms one more time the strong influence that cache mechanisms have on the execution time of computer programs in fully-featured microprocessors.

After obtaining the analytical ETBs of `BSORT`, `MM`, and `LUD` and collecting extensive measurements of their execution times, we applied the curve-fitting technique to derive tighter soft ETBs for these test functions. The rest of the analysis is based on the results obtained under the non-

controlled cache scenario since the analytical bounds in this scenario are the tightest, but still safe.

After augmenting the CDF of the measured execution times of each test function with the corresponding analytical bound, the augmented CDFs were passed to CurveExpert 3.1 to obtain appropriate smooth models. The identified models are shown in Figure 4.[10] With these models application designers may obtain soft ETBs much tighter than the analytical bounds by establishing the desired probability α of such ETBs not being exceeded during runtime. For example, if $\alpha = 0.9$, then the soft ETBs (and the ratios $SoftETB/MMVF$) for `BSORT`, `MM`, and `LUD` are 87.16 ms (4.91), 0.9348 ms (3.98), and 0.089 ms (1.64), respectively.

6 Conclusion

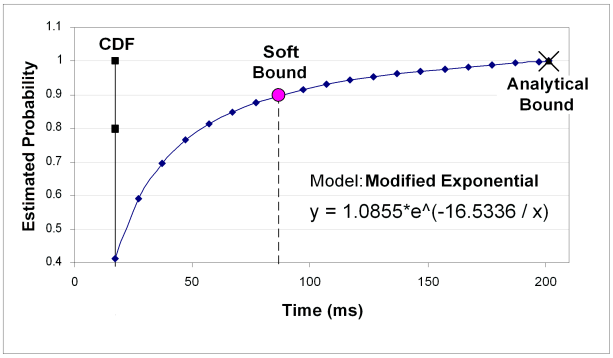
In this paper, we gave a refined description of a *hybrid approach* to deriving reasonably safe and tight execution-time bounds (ETBs) of program-segments. This hybrid approach symbiotically combines execution-time measurements and an analytically derived loose ETB of a given program-segment for: 1) determining the safety margin to be added to the maximum observed execution time to produce a tighter ETB for the program-segment, and 2) estimating the probability of the tighter ETB not being exceeded at run time.

We discussed several *techniques for measuring the execution times of acyclic-path segments* (APSs), which are the basic units of analysis in the hybrid approach. We presented a technique, called *code translation*, that manipulates the assembly codes of the APSs in order to enforce their execution during measurements. Then, we described the structure of the instrumentation program for measuring the execution times of APSs and program-segments and how to deal with the acceleration features of modern high-end processors to obtain representative execution-time measurements. Also, we discussed the estimation of ETBs for APSs from execution-time measurements.

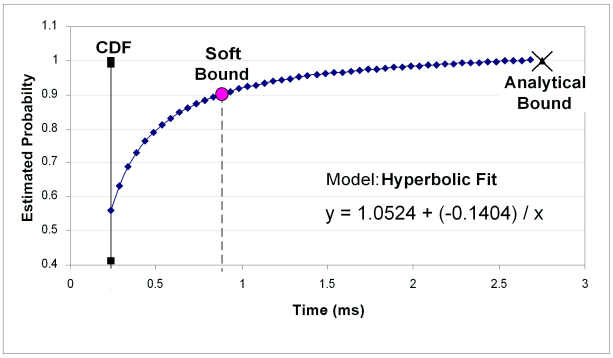
We applied the hybrid approach and the proposed measurement techniques in the derivation of tight ETBs of three algorithms frequently used in the evaluation of WCET analysis techniques; i.e., bubble sort, matrix multiplication, and LU decomposition. From the experimental work we confirmed that the analytical bounds produced under the *APS-based Path Enumeration Technique* (APET) tend still to be overly pessimistic. This fact only emphasizes the importance of hybrid approaches in deriving tighter ETBs of program-segments. We also found that using the maximum among the measured execution times of each APS as the ETB of the APS may be sufficient to obtain a safe analytical ETB with APET due to the inherent pessimism of this technique. Additionally, we empirically demonstrated that tighter analytical ETBs of program-segments are obtained indeed when the analysis is based on APSs instead of basic blocks.

In the future, we plan to extend the hybrid approach in order to handle larger and more complex program units that interact with the real-time distributed computing (RTDC) infrastructure (i.e., operating system, middleware, and communication network). This task requires, among other things, accurate and complete timing models for the RTDC software infrastructure that must be developed and validated through much further work, including both analytical and

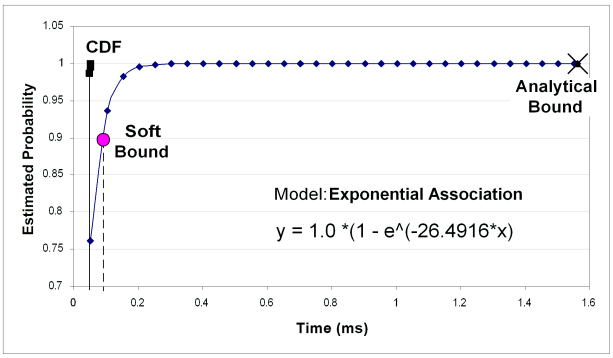
[10] Note that CDF looks like a line due to the scale.



(a) Bubble sort.



(b) Matrix multiplication.



(c) LU decomposition.

Figure 4. Derivation of tight execution-time bounds for the test functions.

experimental research.

Acknowledgments

The research reported here is supported in part by the NSF under Grant Numbers 03-26606 (ITR) and 05-24050 (CNS). Juan A. Colmenares also thanks the University of Zulia (LUZ) for supporting his participation in this research. Chae-Deok Lim wishes to acknowledge the support of ETRI for the sabbatical at UCI. No part of this paper represents the views and opinions of the sponsors mentioned above.

References

[1] A. A. Burns and S. Edgar. Predicting computation time for advanced processors architectures. In *Proc. 12th Euromicro Conf. on Real-Time Systems (ECRTS 2000)*, page 89, June 2000.

[2] G. Bernat, M. J. Newby, and A. Burns. Probabilistic timing analysis: an approach using copulas. *Journal of Embedded Computing*, 1(2):179–194, 2005.

[3] C. Burguière and C. Rochange. History-based schemes and implicit path enumeration. In *Proceedings of the 6th Workshop on Worst-Case Execution Time Analysis (WCET'06)*, pages 17–22, July 2006.

[4] J. F. Deverge and I. Puaut. Safe measurement-based WCET estimation. In *Proceedings of the 5th Workshop on Worst-Case Execution Time Analysis (WCET'05)*, pages 13–16, July 2005.

[5] S. Edgar and A. Burns. Statistical analysis of WCET for scheduling. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, pages 215–224, December 2001.

[6] J. Engblom and A. Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *Proceedings of the 6th IEEE Int'l Conference on Real-Time Computing Systems and Applications (RTCSA99)*, December 1999.

[7] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS'00)*, pages 163–174, November 2000.

[8] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Worst-case execution-time analysis for embedded real-time systems. *Int. J. on Software Tools for Technology Transfer*, 4(4):437–455, August 2003.

[9] A. Ermedahl, F. Stappert, and J. Engblom. Clustered worst-case execution-time calculation. *IEEE Transactions on Computers*, 54(9):1104–1122, September 2005.

[10] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proceedings of the 1st Int'l Workshop on Embedded Software (EMSOFT'01), LNCS 2211*, pages 469–485, London, UK, 2001. Springer-Verlag.

[11] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(2):1038–1054, July 2003.

[12] C. Im. *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*. PhD thesis, University of California, Irvine, December 2005.

[13] C. Im and K. H. Kim. A hybrid approach in TADE for derivation of execution time bounds of program-segments in distributed real-time embedded computing. In *Proceedings of the 9th IEEE Int'l Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, pages 408–418, 2006.

[14] Intel Corporation. Using the RTDSC instruction for performance monitoring. Pentium II processor application notes, 1997.

[15] Intel Corporation. IA-32 intel architecture software developer's manual, volume 1: Basic architecture, January 2006.

[16] K. H. Kim. QoS certification of real-time distributed computing systems: issues and promising approaches. *IEICE Transactions in Information and Systems (IEICE/IEEE joint special issue on assurance systems and networks)*, E86-D(10):2077–2086, October 2003.

[17] K. H. Kim. Timeliness assurance via hybrid approaches during design of distributed embedded computing systems. In *Proceedings of the IEEE 9th Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03F)*, pages 307–313, October 2003.

[18] K. H. Kim, L. Choi, and M. H. Kim. Issues in realization of an execution time analyzer for distributed real-time objects. In *Proceedings of the 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'00)*, pages 171–178, 2000.

[19] R. Kirner, P. Puschner, and I. Wenzel. Measurement-based worst-case execution time analysis using automatic test-data generation. In *Proc. 4th Workshop on Worst-Case Execution Time Analysis (WCET'04)*, pages 67–70, June 2004.

[20] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 16(12):1477–1487, December 1997.

[21] Y.-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems*, 4(3):257–279, July 1999.

[22] S.-S. Lim, Y. Bae, C. Jang, B.-D. Rhee, S. Min, C. Park, H. Shin, K. Park, and C. Ki. An accurate worst-case timing analysis for risc processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[23] S. M. Petters. *Worst Case Execution Time Estimation for Advanced Processor Architectures*. PhD thesis, Technische Universität München, Germany, March 2002.

[24] P. Puschner and A. Schedl. Computing maximum task execution times - a graph-based approach. *Real-Time Systems*, 13(1):67–91, July 1997.

[25] F. J. Richards. A flexible growth function for empirical use. *Journal of Experimental Botany*, 10(2):290–301, June 1959.

[26] F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *J. of Systems Architecture*, 46:339–355, 2000.

[27] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner. Automatic timing model generation by CFG partitioning and model checking. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'05), Volume 1*, pages 606–611, March 2005.

[28] N. Williams, B. Marre, P. Mouy, and M. Rogers. PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In *Proceedings of the 5th European Dependable Computing Conference (EDCC 2005), LNCS 3463*, pages 281–292, April 2005.