Reinforcing Total Bandwidth Server with Multivalued WCET

Amr Ashmawy[1,a) Kiyofumi Tanaka[1,b)

Abstract: As real-time embedded systems get more diverse and complicated, systems with different types of tasks (e.g., periodic and aperiodic tasks) are getting prevalent. In such systems, guaranteeing schedulability is important for hard periodic tasks while keeping response times of soft aperiodic requests short enough. Total Bandwidth Server (TBS) is one of the promising scheduling algorithms for hybrid task sets which include both periodic and aperiodic tasks. Basically, TBS follows the Earliest Deadline First algorithm, where tasks with earlier deadlines are prioritized for scheduling. This implies a promising opportunity for shortening response times of chosen tasks by assigning earlier deadlines. This paper describes a technique, stepwise deadline update, that moves deadlines earlier in the context of TBS. In this technique, a job execution is divided into two or more sub instances and each is given an individual deadline. The deadlines are calculated based on estimated execution times instead of a simple worst-case execution time (WCET). Considering that task's actual execution time is in most cases shorter than its WCET, that potentially improves such tasks responsiveness. For the estimated execution times of each task, this paper introduces multivalued WCET, a collection of possible and representative execution times of that task. The estimated execution times are obtained by static code analysis and abstract symbolic execution. The simulation-based evaluation shows that stepwise deadline update technique with multivalued WCET reduces average response times of aperiodic tasks. When the processor utilization is high, the reduction rate of aperiodic response times reaches 51.1% compared to traditional TBS, with negligible scheduling overhead.

Keywords: real-time scheduling, Total Bandwidth Server, response time, worst-case execution time (WCET), multivalued WCET

1. Introduction

With growing demands for highly functional electronic devices and automation systems, it is getting common in real-time embedded systems that different types or criticalities of tasks compose a system. Thus real-time scheduling gains growing importance in order to meet the different real-time requirements [1], [2], [3]. To achieve the required real-time processing in such a system, sophisticated scheduling algorithms must be used to target both hard and soft (or non real-time) tasks while guaranteeing the schedulability of the hard tasks and at the same time exhibit short response times for the soft (or non real-time) tasks.

The schedulability is satisfied when all hard tasks meet their deadlines. Therefore, hard tasks have to be periodically (or at least sporadically) invoked and executed, and assumed to spend their worst-case execution times (WCETs) in schedulability analyses for safety concerns. On the other hand, soft (or non real-time) tasks with less tight timing requirements can run on aperiodic invocations as long as they do not influence the schedulability of hard tasks. Total Bandwidth Server (TBS) [4] is one of the scheduling algorithms for both hard and soft (or non real-time) tasks, which provides reasonable response times for aperiodic tasks. Hard (periodic) tasks' schedulability is preserved unless soft (aperiodic) executions were found to exceed the corresponding WCETs. TBS is based on the earliest deadline first (EDF) algorithm [5] and therefore has the characteristic that a processor can be utilized up to 100% while maintaining schedulability. This study is based strongly on TBS and explores enhanced techniques to improve TBS.

Considering the complexity of current processors and application programs it is difficult to exactly estimate WCETs [6]. Moreover, the worst-case execution path in a program is in many cases impossible to find along the possibly infinite branches and loop structures. Additionally, enormous number of input patterns makes the search space virtually unbounded [7]. Consequently, for safety, WCETs must be pessimistically over-estimated and probably higher than the actual possible execution times. This gap decreases the effectiveness of scheduling algorithms when making scheduling decisions based on tasks' execution times; Shortest Job First (SJF) – Shortest Remaining Time First (SRTF) [8]. Similarly, unnecessarily long WCETs worsen responsiveness to aperiodic requests in TBS, since the long WCETs leads to long deadlines. Therefore, considering the difficulty in obtaining exact WCETs, a safe scheduling technique is desired whose performance dosesn't depend on WCETs.

The aim of this work is to improve TBS in terms of re-

[1] School of Information Science, Japan Advanced Institute of Science and technology, Nomi, Ishikawa 923–1211, Japan
a) a.ashmawy@jaist.ac.jp
b) kiyofumi@jaist.ac.jp

sponse times of aperiodic tasks. The improvements are achieved by a technique, stepwise deadline update which we proposed in the previous work [9], [10], [11]. This technique tries to give a job a deadline which is not based on WCET but on two or more estimated execution times shorter than WCET. The shorter estimated execution time leads to earlier deadline and earlier scheduling of the job. However, how to estimate the execution times was out of the scope of the previous work. In this paper, we propose a series of techniques for estimating two or more execution times, called multivalued WCET. The execution times (multivalued WCET) are obtained by static code analysis and abstract symbolic execution then applied to the stepwise deadline update technique.

This paper consists of six sections. Section 2 describes related work for scheduling algorithms for task sets with hard/periodic and soft(or non real-time)/aperiodic tasks as well as WCET analysis. Section 3 describes an extended TBS with stepwise deadline update and virtual release advancing, for reducing response times of aperiodic tasks. Then, in Section 4, a method of obtaining multivalued WCET is proposed. The proposed technique is evaluated in Section 5. Finally, Section 6 concludes the paper with summary and future work.

2. Related Work

2.1 Scheduling Algorithms for Both Periodic and Aperiodic Tasks

There are various scheduling algorithms for task sets consisting of periodic and aperiodic tasks. They are categorized to fixed-priority servers and dynamic-priority servers. Fixed-priority servers are based on rate monotonic (RM) scheduling [5], which has the merit that higher-priority periodic tasks with shorter-periods tend to have lower jitters and shorter response times. Representative examples are Deferrable Server [12], Priority Exchange [12], Sporadic Server [13], and Slack Stealing [14]. On the other hand, dynamic-priority servers are based on the EDF algorithm. EDF possesses a the capability to make a processor utilization reaches almost 100% while maintaining schedulability. Dynamic Priority Exchange [4], Dynamic Sporadic Server [4], TBS, Earliest Deadline Late Server [4], and Constant Bandwidth Server (CBS) [15] are examples of dynamic-priority servers. Scheduling algorithms handling such mixed tasks' sets try to make the response times for aperiodic requests shorter while guaranteeing schedulability by adding complexity. Considering the significance of the higher utilization bound, this paper targets EDF-based server algorithms.

There is another strategy, slack reclaiming, for improving responsiveness of jobs. Based on [16], there are two types of slack reclaiming; reclaiming within a server (as Constant Bandwidth Server) and reclaiming between servers [14], [17], [18][*1]. In both cases, when some jobs finish earlier than their WCETs, the slack time (or unused bandwidth) is utilized by other soft or non real-time task executions. This means slack time can be utilized only after the (early) completion of the prior jobs. In contrast, in the stepwise deadline update, it is detected/predicted that the execution of a target task would be completed earlier. Then a corresponding earlier deadline is assigned so that the target task can use its own future. This technique is regarded as one kind of reclaiming within a server and therefore discussion with comparison to reclaiming between servers is left beyond the scope of this paper.

2.2 Total Bandwidth Server (TBS)

Resource reservation is a general technique for real-time systems where aperiodic tasks are executed only in some reserved bandwidth and avoid influencing hard tasks' schedulability [3], [19]. TBS is one of the resource reservation methods, a scheduling algorithm for a mixture of hard periodic and non real-time aperiodic tasks[*2]. TBS provides fair response times for aperiodic tasks while keeping its implementation complexity moderate [4], [20]. U_p usually refers to the processor utilization factor by all hard periodic tasks and U_s the bandwidth or the processor utilization factor of the server. It has already been proven that a task set is schedulable if and only if $U_p + U_s \leq 1$ [4].

There are other algorithms derived from TBS. [21] proposed a method for firm (not hard) periodic tasks and soft (or non real-time) aperiodic tasks. This method aims to achieve short response times by sacrificing completeness of periodic task executions. On the other hand, this paper targets techniques of shortening response times of aperiodic jobs and at the same time ensuring the integrity of hard periodic tasks' executions.

2.3 Improved TBS

In another TBS enhancement, after a deadline is given by TBS, the finishing time of the target task is pre-computed by summing up its WCET with the WCETs of other higher-priority tasks and then the deadline is fixed in advance of the execution [21], [22]. This way the response time for aperiodic tasks can potentially be improved in mixed tasks' sets without jeopardizing the safety of the periodic tasks thus schedulability. The estimation of the finishing time and the update of the deadline are repeated until the deadline converges (TB*). This is based on EDF optimality that a task set is scheduled feasibly by EDF if any other algorithm can make the task set feasible [23]. This technique forces considerable complexity for the iterative calculation of the finishing times, thus TB(n) limits the number of iterations to some n. Such added complexity posses considerable overhead that our proposed solution

[*1] Strictly, Slack Stealing [14] is a method of utilizing slack times obtained by postponing hard tasks' execution, not by earlier finishing.

[*2] A server in TBS basically targets non real-time tasks. However, it can handle soft tasks as well as non real-time tasks. Therefore, this paper does not distinguish between soft and non real-time tasks.

manages to reduce.

2.4 Constant Bandwidth Server (CBS)

CBS is another scheduling algorithm intended for mixed tasks' sets with comparable complexity to TBS. Unlike TBS, CBS does not expect fixed execution times which makes it suitable for multimedia applications with a budget for aperiodic tasks. CBS has a server period phase varying depending on requests' arrival timing. A common deadline is repeatedly updated together with the server period and budget for all aperiodic jobs regardless their execution times. According to the literature [15], CBS is summarized as follows. A CBS has the server period (T_s), the maximum budget (Q_s), the server bandwidth ($U_s = Q_s/T_s$), the current budget (c_s), and, at each instant, the common deadline ($d_{s,k}$).

CBS has the merit that slack reclaiming within the server is naturally performed since the budget is decreased according to the actually spent execution times of jobs. If a subsequent job's execution fits the remaining budget, it benefits from the current common deadline. In addition, when the budget is exhausted but the execution is not completed, the budget is replenished by borrowing from the future job[*3]. However, the deadline depends on the server period, not on individual jobs' execution time, which might not provide an optimal deadline for each job. On the other hand, the technique in this paper can give each job an optimal deadline according to the actual execution time.

2.5 WCET Analysis and Abstract Symbolic Execution

Estimation of the upper bound WCET is essential to guarantee schedulability of real-time tasks. The nature of real-time applications must be highly predictable and responsive using simpler code structures. While obtaining such WCET is impossible in general tasks, it is feasible for real-time systems due to the peculiar nature of real-time tasks. Still WCET estimation is quite a complicated task and highly dependent on the target hardware [24], [25].

An estimated single-valued WCET is not as useful as a customizable or multivalued WCET. Running WCET analysis several times to get more than one value is extremely prohibitive. Other alternatives fall under the general idea of a (customizable) parametric WCET function built at compile (analysis) time to provide different WCET estimations at run-time given input (control) values [26]. Such parametric WCET functions can only be used for research and impractically small code sizes.

Unlike the related work mentioned above, the WCET analysis we propose provides a new alternative to obtain multivalued WCET. The values are obtained without the need for such parametric function or running WCET analysis several times. Concolic execution technique is here adapted from software validation and error detection research [27]. Concolic execution (explained below) is an extension of symbolic abstract execution techniques, which is also known as symbolic execution for short [28]. Simple symbolic execution was used in previous WCET analysis, although it was restricted to small problems and code size. The complexity of the constraints generated during analyzing and tracing the code is infeasible with most constraint solvers [29].

Both symbolic and concolic execution have been long used in software testing [27] The former is preferred to maintain test completeness, but restricted to small simple code. On the other hand, concolic execution is preferred to tackle much bigger program sizes, obfuscated code structures and data ambiguity, maintaining only soundness and sacrificing completeness. Concolic execution is the combination of both concrete execution and abstract symbolic execution [30].Concrete execution can be considered as profiling of instrumented code to trace and guide and execution. The tracing and guiding is used to force the execution into the different possible paths and test those paths for possible occurrences of errors.

3. Stepwise Deadline Update

In our previous ATBS research, a technique to provide some thing like multivalued WCET was hinted without details. This section and the next one describe details of TBS modification, stepwise deadline update and multivalued WCET.

3.1 Definition of stepwise deadline update

In the stepwise deadline update, a job is divided into two or more sub instances. Let J_k be the k-th aperiodic job. J_k is divided into sub instances, $J_k^1, J_k^2, J_k^3, \ldots$. The first sub instance, J_k^1, corresponds to the execution from the beginning of J_k to the time when C_k^1 has been elapsed. Then J_k^i is the execution from $\sum_{j=1}^{i-1} C_k^j$ to $\sum_{j=1}^{i} C_k^j$. Here, C_k^i is the estimated execution time for J_k^i. If J_k finishes in $\sum_{j=1}^{i} C_k^j$, J_k^{i+1} and the following sub instances do not exist.

When the arrival time of the k-th aperiodic request is r_k, each sub instance, J_k^i, is given the deadline, d_k^i, as:

$$d_k^1 = max(r_k, d_{k-1}) + \frac{C_k^1}{U_s} \quad (1)$$

$$d_k^i = d_k^{i-1} + \frac{C_k^i}{U_s} \qquad (i > 1) \quad (2)$$

3.2 Example of stepwise deadline update

Figure 1 (1) and (2) depict examples of the original TBS and TBS with the stepwise deadline update, respectively. There is a periodic task, τ_1, which has a period of $T_1 = 6$, execution time of $C_1 = 4$, and utilization of $U_1 = C_1/T_1 = 2/3 = U_p$. The TBS bandwidth is $U_s = 1 - U_p = 1/3$. The aperiodic request released at $t = 2$ has $C_1^{WCET} = 6$. The actual execution time of this aperiodic job can be from 1 to 6 units of time. As in Figure 1 (1), the original TBS produces response times of 3, 4, 9, 10, 15, or 16 units of time for the corresponding execution

*3 This is called "borrowing from the future" in [18].

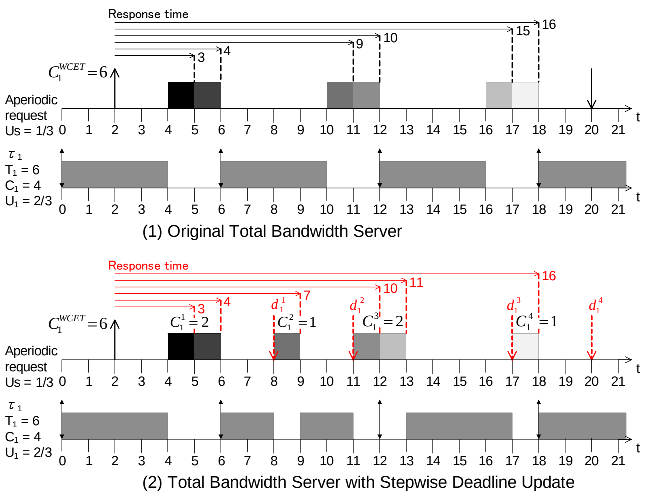


Fig. 1: Example of stepwise deadline update.

time. On the other hand, in Figure 1 (2), the stepwise deadline update technique divides the aperiodic job into four sub instances, J_1^1, J_1^2, J_1^3, and J_1^4, and estimates the execution times of $C_1^1 = 2$, $C_1^2 = 1$, $C_1^3 = 2$, and $C_1^4 = 1$ for them. The corresponding deadlines are calculated as $d_1^1 = 8$, $d_1^2 = 11$, $d_1^3 = 17$, and $d_1^4 = 20$. Depending on the actual execution time, the response time becomes 3, 4, 7, 10, 11, or 16 units of time. In this example, the response time is shortened when the actual execution time is 3 or 5 units of time,compared to the original TBS.

The effectiveness of the stepwise deadline update depends on the estimated execution times. In the above example, if C_1^1 had been estimated as one unit of time, d_1^1 would have become $t = 5$ and the job execution would have started immediately at the arrival time and the response time would have been one unit of time.

3.3 Estimation of execution times of sub instances

To apply the stepwise deadline update technique to TBS, estimated values of execution times, C_k^i, for sub instances are prepared. Since how to estimate execution times can be considered separate from the definition of the stepwise deadline update technique, various kinds of estimation methods can be applied. In this paper, we apply multivalued WCET proposed in Section 4. The multivalued WCET can be expected to be close to actual execution times since it is obtained by analyzing and concolically executing a target application code and extracting representative execution paths in the code.

After an aperiodic job is divided into two or more sub instances, this enhanced technique leads to the same behavior as the original TBS. That is, the two or more sub instances are regarded as instances that occur simultaneously and independently (but are clearly ordered), each with its own WCETs C_k^i. Then the original TBS would give each sub instance the same deadline value as the one calculated by equation 1 and equation 2. Therefore, schedulability of the stepwise deadline update is the same as the original TBS, which is that a task set is schedulable if and only if $U_p + U_s \leq 1$.

3.4 Implementation complexity and run-time overhead

In TBS with the stepwise deadline update algorithm, aperiodic job's execution is divided into more than one sub instances. However, operating systems should manage a task with a single information set, task control block (TCB). This is realized by resetting up the deadline and reinserting the TCB in the ready queue every time the estimated execution time for the current sub instance has elapsed but the job did not finished, which is the only difference from the original TBS. To find that the execution has reached the estimated execution time, the scheduler should be executed every tick timing. This is achieved by calling the scheduler when timer interrupts occur, which is a natural procedure that operating systems commonly follow.

The deadline calculations defined in equation 1 and equation 2 include division. The overhead can be alleviated by using constants or statically prepared values for C_k^i and performing only an addition. Since U_s is a constant, the second term in the right side of the equations becomes constant.

3.5 Qualitative comparison with CBS

The aim of the stepwise deadline update is to provide appropriate deadlines to tasks with varying execution times while keeping the schedulability, which is the same as the aim of CBS. The difference is mainly in how to estimate execution times. In CBS, it can be said that any task's execution time is considered fixed (maximum budget Q_s). On the other hand, in the stepwise deadline update, sub instances are given individual estimated execution times. If the estimated execution times for all sub instances are set to some (same) fixed value, the stepwise deadline update is essentially the same as CBS.

4. Multivalued WCET

4.1 Proposed Framework

The overall framework for multivalued WCETs is shown in Figure 2 and explained in more details below and in the following figures.

4.1.1 Stages 1–3: Traditional structural static WCET analysis stages

The code to be analyzed is first input to the lower three stages of the framework, similar to regular structural WCET analyzers. In those three stages the code is checked exhaustively using regular static WCET analyzer techniques to get a single WCET estimation value according to the models surveyed in related literature [7]. Low level timing analysis of the system obtained with some measuring executions is performed and then combined with the enriched CFG from the earlier control-flow analysis of the first stage. Implicit path enumeration technique (IPET) is preferred for the binding rather than tree or path binding methods, which is an extended solution of an integer linear

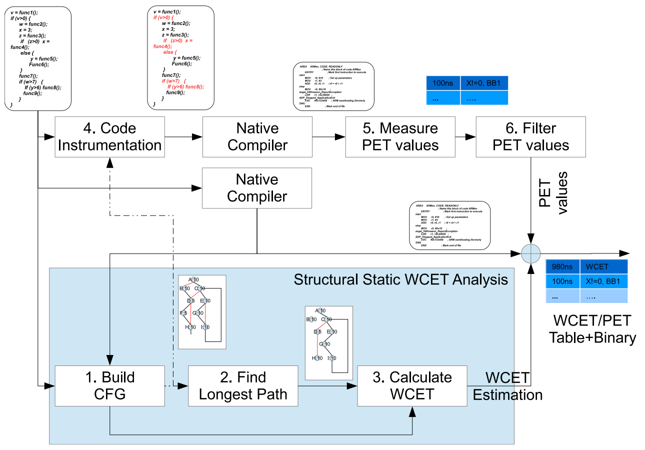


Fig. 2: Multivalued WCET framework.

programming (ILP) model of the code [7].

4.1.2 Stage 4: Perform – Code Instrumentation

To estimate the execution times of the other shorter paths, separate analysis of those paths is needed. That analysis can run in parallel together with the stages mentioned above. The cost of traditional WCET analysis prohibits running it several times. The search space is also prohibitively large to explore exhaustively. Efforts to obtain several execution time estimations for the same code under different execution scenarios are quite limited. Those different execution scenarios considered here could possibly run through execution paths other than the longest path (worst case) but might include it.

Unlike parametric WCET and such previous efforts, this research uses concolic execution techniques already used in software testing and validation. The number of values obtained from concolic execution can be controlled depending on the accuracy needed. The binary of the analyzed code is run under the control of a concolic simulator. The concolic simulator forces the execution of the binary through different paths of the CFG. In this stage, the simulator starts by analyzing the code and/or CFG and then performs code instrumentation to insert collectors for tracing and profiling.

4.1.3 Stage 5: Measure – PET values

After each run of the generated binary the trace is analyzed to find the values of the control variables and the corresponding execution path. The simulator uses a constraint solver to calculate other values of the control variables so it can direct next execution into a different path. The simulator then starts a new run to get the next trace information. This stage is repeated a number of times depending on the accuracy needed. The trace includes extra information about the path begin followed to obtain the execution time recorded. The execution time recorded is referred to as Path Execution Time (PET). The recorded path information can be encoded and used at run-time for further improving the run-time scheduling.

Naturally the recorded PET values would be lower than the WCET estimation obtained from stage 3. Even if the longest path was the one just encountered, static WCET provides a safe overestimation of the actual WCET. If required, the path corresponding to the estimated WCET can be used to guide the simulator. This can avoid recalculating the longest path by the simulator or exploring similar paths with minor differences. A more accurate estimation of WCET can also be obtained, after taking into account all the other possible environment factors, as memory hierarchy and sharing contention.

4.1.4 Stage 6: Generate – WCET/PET table

A number of the collected PET values are selected as representative paths of the code, together with the WCET. Simply, the obtained PET values could be quantized with some accuracy to aggregate similar PET values. A more sophisticated selector utilizes the histogram of the PET values. The local maximal points are selected up to the predefined number of PET values required. Wavelet-like selector can also be useful to get ranges of high occurrence rather than values of high occurrence.

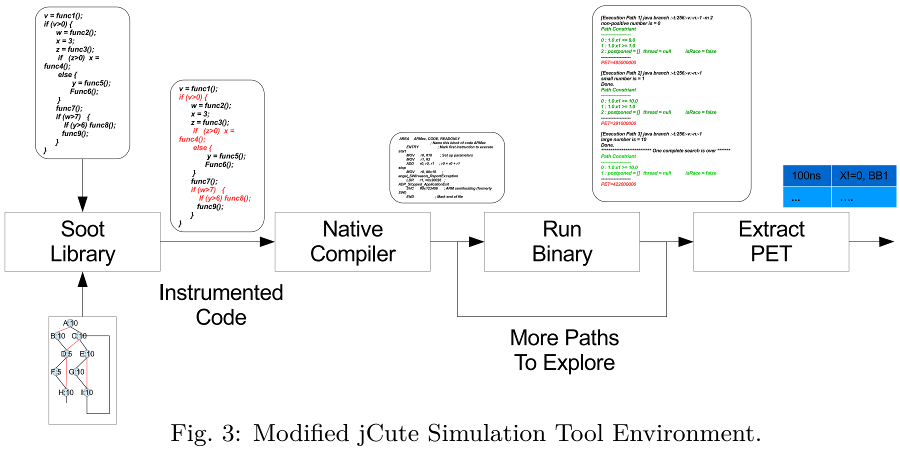
The selected PET values are to be packed together with the WCET estimation from stage 1 to 3 and passed down with the normal binary generated from the code without instrumentation. The table is simply referred to as WCET/PET table, which is utilized by the scheduler with stepwise deadline update in this work.

4.2 Simulator

Figure 3 shows the environment for generating WCET/PET table. The modified jCUTE [31] simulator first applies the soot library to perform code instrumentation at control flow points (conditional statements) for tracing. The instrumented code obtained is fed into a normal compiler to get a machine dependent binary. The simulator starts a loop to run the binary a number of times. Every time the binary is given values for the input (control) variables.

The simulator checks the trace generated to find the explored path in the previous run and the generated conditions through that path. The solution obtained from lp_solve [32] is then used to force the next execution to a different path. The control flow points and variables are chosen in a depth-first recursion for changing the values of one or more of those variables.

The simulator keeps track of the branches tree to record the paths (branches) already investigated not to repeat paths. The simulator detects when the whole branch tree has been covered and exits. The simulator also exits after covering a fixed number of paths or after a predetermined time, to avoid getting lost in enormous or infinite CFG trees. Experiments showed that most of the examples with 20 paths or more were found to have above 1000 paths. A reasonable number of paths (100) is chosen to be able to get a suitable sample to extract representative PET. Currently the extra control information about the branch directions and variables traversed in the explored path are overlooked but stored for future extensions, shown in green in the code of Figure 3. The simulator ends up by



```
v = func1();
if (v>0) {
      w = func2();
      x = 3;
      z = func3();
       if   (z>0)  x =
func4();
       else {
             y = func5();
            Func6();
      }
      func7();
      if (w>7)   {
         If (y>6) func8();
        func9();
      }
}
```

```
v = func1();
if (v>0) {
      w = func2();
      x = 3;
      z = func3();
       if   (z>0)  x =
func4();
       else {
             y = func5();
            Func6();
      }
      func7();
      if (w>7)   {
         If (y>6) func8();
        func9();
      }
}
```

```
AREA     ARMex, CODE, READONLY
                         ; Name this block of code ARMex
        ENTRY                  ; Mark first instruction to execute
start
        MOV      r0, #10         ; Set up parameters
        MOV      r1, #3
        ADD      r0, r0, r1      ; r0 = r0 + r1
stop
        MOV      r0, #0x18       ;
angel_SWIreason_ReportException
        LDR      r1, =0x20026    ;
ADP_Stopped_ApplicationExit
        SVC      #0x123456       ; ARM semihosting (formerly
SWI)
        END                    ; Mark end of file
```

```
[Execution Path 1] java branch :-t:256:-v:-n:-1 -m 2
non-positive number is = 0
Path Constriant
-------------------
0 : 1.0 x1 <= 9.0
1 : 1.0 x1 >= 1.0
2 : postponed = []   thread = null         isRace = false
-------------------
PET=485000000

[Execution Path 2] java branch :-t:256:-v:-n:-1
small number is = 1
Done.
Path Constriant
-------------------
0 : 1.0 x1 >= 10.0
1 : 1.0 x1 >= 1.0
2 : postponed = []   thread = null         isRace = false
-------------------
PET=391000000

[Execution Path 3] java branch :-t:256:-v:-n:-1
large number is = 10
Done.
*************************** One complete search is over *******
Path Constriant
-------------------
0 : 1.0 x1 >= 10.0
1 : postponed = []   thread = null         isRace = false
-------------------
PET=422000000
```

100ns	X!=0, BB1
...	....

Fig. 3: Modified jCute Simulation Tool Environment.

extracting a list of PET values from the execution trace, shown in red.

4.3 Filtering PET lists

Following the simulator, the obtained PET list is usually quite long and not useful. First the PET values in the list are quantized to filter minor time measuring anomalies. The histogram of filtered values is then constructed, and the PET values with the highest occurrences are selected as the representative PET values. Alternatively a wavelet filter can be used to find PET ranges of high density rather than simple peaks. The average of those periods would be selected as the representative PET.

Either way the selected representative PET values are combined with WCET in the WCET/PET table to be the multivalued WCET. The table will be packed with the code binary (without instrumentation) and kept in the Task Control Block (TCB) during execution.

5. Evaluation

In this section, the proposed technique which uses multivalued WCET information is compared with TBS, CBS, and the improved TBS in terms of aperiodic responsiveness and scheduling overheads through simulations. In the simulation, synthetically generated periodic tasks are used. On the other hand, aperiodic tasks are supposed to have execution times which are obtained from the source code analysis.

5.1 Benchmark applications

Table 1 shows the benchmark application programs that are regarded as aperiodic tasks in our evaluation. The source codes of the benchmark programs are from three different projects: Mälardalen Real-Time Research Centre [33], Embedded System Research Group [34] and jCUTE [31].

Each source code was run by jCUTE to maximum of 100 times. Some benchmarks explores all the possible paths below 100 and quits. Most of those that reach 100 paths without quitting were found to have over 10,000 paths. Thus, only 100 executions were chosen to get representative paths. For every benchmark the number of paths discovered using jCUTE and the corresponding execution times on the test platform are reported.

The PETs obtained are quantized to filter small variances from similar paths. Then, the representative PET values are scaled down to one hundredth considering that the measured values are from Java environment and each program is fed with relatively large inputs. In simulation for real-time scheduling, each aperiodic task is supposed to spend the same execution time as one of the obtained PET values according to the occurrence probability obtained from the paths' analysis.

Table 1: Benchmark programs.

Program	Source	Paths	Description
bs	[33], [34]	100	Binary search an array
BSTree	[31]	100	Binary search Tree class
cnt	[33]	100	Counts non-negative numbers in matrix
DSort	[31]	89	JPF Dsort class
duff	[33]	100	Using Duffs device from Jargon file to copy an array
fft1	[33], [34]	100	Fast Fourier Transform
BinTreeBare	[31]	100	Binary Tree
NS	[31]	100	Network messaging
XACML-Policy	[31]	13	XACML policy for teacher student grading
ShortestPath	[31]	100	Shortest path calculation
Sort	[31]	100	Sort an array of processes for scheduler
TMN	[31]	27	ITU-T TMN guard-key management

5.2 Aperiodic responsiveness

Periodic task sets are synthetically generated by using probabilistic distributions for their periods and execution times. The total processor utilization factors (U_p) by periodic tasks are 60% to 95% at 5% intervals. Each periodic task has its period which is decided by uniform distribution between 1 and 100 ticks. Its WCET and actual execution time are equal and obtained by uniform distribution between 1 and 1/3 of the periods.

The total utilization by aperiodic tasks is around 5% in the observation period (100,000 ticks). All the aperiodic servers are supposed to have the utilization of $U_s = 1 - U_p$. The WCET of each aperiodic task is the longest path obtained from the analysis times a safety factor of 1.5. Each job of an aperiodic task has its actual execution time decided based on the representative PETs and their occurrence probability. The arrival intervals are decided by Poisson distribution with the average of $(WCET/0.05)$. For each U_p, all combinations of 10 periodic task sets and 10 aperiodic task sets (total 100 task sets) are simulated and the average value of aperiodic response times is shown.

For the estimated execution times in the stepwise deadline update technique, we applied two options. First is to assign the average execution time to C_k^1 and WCET to C_k^2, which corresponds to our previous work in [10]. The other is the proposed technique in this paper where C_k^1 is the shortest PET among the representative PETs obtained, C_k^2 is the next shortest one, and so on. For CBS, the maximum server budget, Q_s, is set to the average execution time of all the jobs of the target aperiodic task. The server period becomes $T_s = \lceil Q_s/U_s \rceil$.

The original TBS (TBS), our previous technique (ATBS), the proposed solution with multivalued WCET (ATBS+MWCET), Improved TBS (TB*), and CBS are compared. For space considerations, only some selected results are shown in Figure 4. For all the benchmarks except XACMLPolicy and TMN, the proposed method (ATBS+MWCET) gives the best average response times. Although TB* is an optimal method among TBS-based techniques, its optimality is guaranteed only for tasks that always spend their WCETs. For XACMLPolicy and TMN, the actual execution times are in most cases near to WCETs. Therefore, TB* exhibits the best results for these benchmarks. On the whole, when U_p is 95%, ATBS+MWCET reduces the average response time up to 51.2%, 47.9%, and 26.2% compared to TBS, TB*, and CBS, respectively.

5.3 Scheduling overhead

Scheduling overhead for all the evaluated algorithms are estimated from information obtained during the simulation runs above. The number of cycles taken for overhead operations are calculated separately based on simulation summing deadline updates (based on the proposed multivalued WCET) and also queue manipulations. Latencies of arithmetic, logical, memory references and control operations are based on Cortex-A9 [35], [36]. It is assumed that the processor's clock frequency is 100MHz and the tick length is 1 millisecond.

Table 2 shows the maximum overhead per tick indicating how much the scheduling process occupies in a tick period with $U_p = 90\%$. The overhead in TBS, ATBS, ATBS+MWCET, and CBS are negligible, less than 0.1%. It is clear that TB* generates much more overhead than the others. As a whole, the proposed method exhibits shorter average response times with small overhead.

6. Conclusion

This paper proposes improving responsiveness in TBS with multivalued WCET and applying it to the stepwise deadline update technique. This gives a deadline to each job (sub)instance tailored to its actual execution time, instead of a simple single value WCET. The result shown expedited shorter response times.

Table 2: Scheduling overheads (%) for $U_p = 90\%$.

Program	TBS	ATBS	ATBS+ MWCET	TB*	CBS
bs	0.023	0.025	0.024	7.2	0.013
BSTree	0.022	0.024	0.024	6.1	0.013
cnt	0.023	0.024	0.024	4.5	0.015
DSort	0.023	0.024	0.024	5.7	0.015
duff	0.023	0.024	0.024	4.3	0.012
fft1	0.022	0.023	0.023	5.7	0.012
BinTreeBare	0.023	0.024	0.024	5.8	0.014
NS	0.022	0.024	0.024	5.6	0.014
XACMLPolicy	0.023	0.025	0.025	4.3	0.016
ShortestPath	0.023	0.025	0.025	5.4	0.015
Sort	0.023	0.025	0.025	5.4	0.014
TMN	0.023	0.025	0.025	4.2	0.016

The simulation based evaluation with information obtained from analyzing actual programs codes showed that the proposed technique certainly reduces average response times of aperiodic tasks compared to TBS, improved TBS, and CBS with negligible scheduling overhead.

In the combination of stepwise deadline update and multivalued WCET, we simply applied the PET values increasingly. In the future, we will try different strategies to reflect the occurrence probability obtained from the paths' analysis. More important, combining the proposed multivalued WCET with multicore scheduling algorithms for real-time systems will be investigated.

References

[1] de Niz, D., Lakshmanan, K. and Rajkumar, R.: On the Scheduling of Mixed-Criticality Real-Time Task Sets, Proceedings of Real-Time Systems Symposium, IEEE Computer Society, pp. 291–300 (2009).

[2] Baruah, S., Li, H. and Stougie, L.: Towards the design of certifiable mixed-criticality systems, Proceedings of Real-Time and Embedded Technology and Application Symposium, IEEE Computer Society, pp. 13–22 (2010).

[3] Buttazzo, G. C.: Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Springer, 3rd. edition (2011).

[4] Spuri, M. and Buttazzo, G. C.: Efficient Aperiodic Service under Earliest Deadline Scheduling, Proceedings of Real-Time Systems Symposium, IEEE Computer Society, pp. 2–11 (1994).

[5] Liu, C. L. and Layland, J. W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, Journal of the Association for Computing Machinery, Vol. 20, No. 1, pp. 46–61 (1973).

[6] Lundqvist, T. and Stenström, P.: Timing Anomalies in Dynamically Scheduled Microprocessors, Proceedings of Real-Time Systems Symposium, IEEE Computer Society, pp. 12–21 (1999).

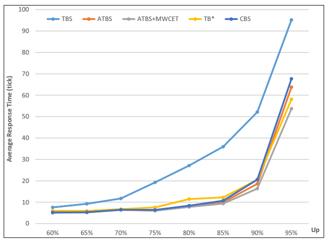
[7] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J. and Stenström, P.: The Worst-Case Execution Time Problem – Overview of Methods and Survey of Tools, ACM Trans. Embedded Computing Systems, Vol. 7, No. 3, pp. 1–53 (2008).

[8] Silberschatz, A., Galvin, P. B. and Cagne, G.: Operating System Concepts, John Wiley & Sons, Inc., 8th. edition (2009).

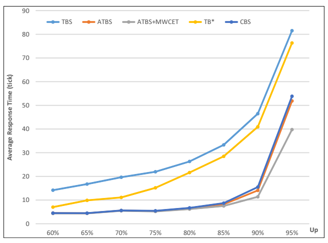
[9] Tanaka, K.: Adaptive Total Bandwidth Server: Using Predictive Execution Time, Proceedings of 4th IFIP TC 10 International Embedded Systems Symposium, Springer, pp. 250–261 (2013).

[10] Tanaka, K.: Adaptive Real-Time Scheduling for Soft Tasks with Varying Execution Times, Journal of Information Processing, Vol. 22, No. 2, pp. 152–159 (2014).

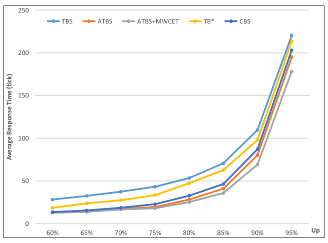
[11] Tanaka, K.: Real-Time Scheduling for Reducing Jitters of Periodic Tasks, Journal of Information Processing, Vol. 23, No. 5, pp. 542–552 (2015).



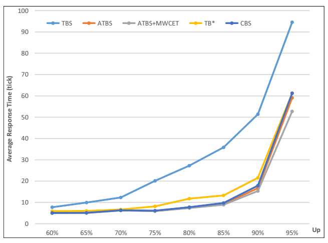
(a) duff



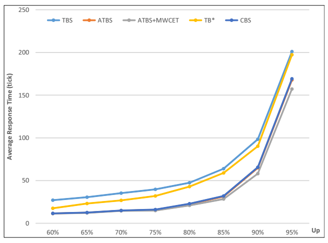
(b) fft1



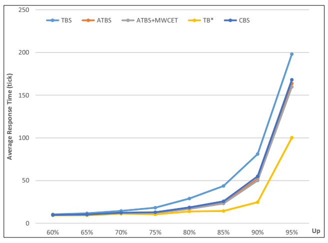
(c) JcuteBinTreeBare



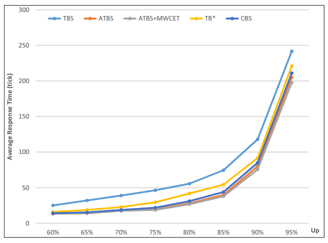
(d) cnt



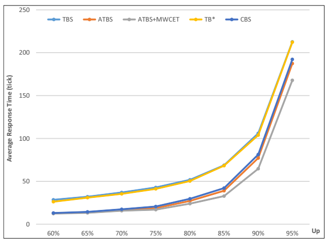
(e) NS



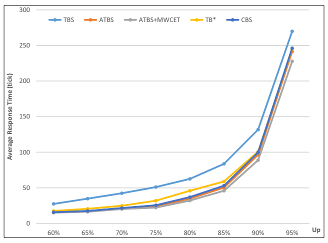
(f) SampleXACMLPolicy



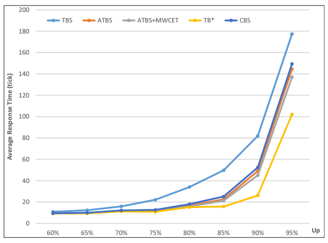
(g) ShortestPath



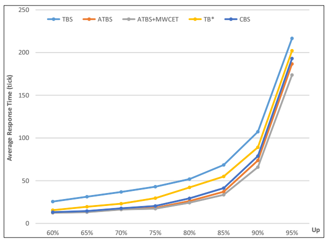
(h) BSTree



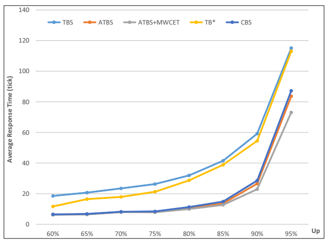
(i) Sort



(j) TMN



(k) DSort



(l) bs

Fig. 4: Average response times

[12] Lehoczky, J. P., Sha, L. and Strosnider, J. K.: Enhanced Aperiodic Responsiveness in Hard Real-Time Environments, Proceedings of Real-Time Systems Symposium, IEEE Computer Society, pp. 261–270 (1987).

[13] Sprunt, B., Sha, L. and Lehoczky, J.: Aperiodic Task Scheduling for Hard-Real-Time Systems, Journal of Real-Time Systems, Vol. 1, No. 1, pp. 27–60 (1989).

[14] Lehoczky, J. P. and Ramos-Thuel, S.: An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems, Proceedings of Real-Time Systems Symposium, IEEE Computer Society, pp. 110–123 (1992).

[15] Abeni, L. and Buttazzo, G.: Integrating Multimedia Applications in Hard Real-Time Systems, Proceedings of Real-Time Systems Symposium, IEEE Computer Society, pp. 4–13 (1998).

[16] Davis, R., Tindell, K. and Burns, A.: Scheduling slack time in fixed priority pre-emptive systems, Proceedings Real-Time Systems Symposium, Raleigh Durham, NC, USA, pp. 222–231 (1993).

[17] Caccamo, M., Buttazzo, G. and Sha, L.: Capacity Sharing for Overrun Control, Proceedings of Real-Time Systems Symposium, IEEE Computer Society, pp. 295–304 (2000).

[18] Lin, C. and Brandt, S. A.: Improving Soft Real-Time Performance Through Better Slack Reclaiming, Proceedings of Real-Time Systems Symposium, IEEE Computer Society, pp. 410–421 (2005).

[19] Abeni, L. and Buttazzo, G.: Resource Reservation in Dynamic Real-Time Systems, Journal of Real-Time Systems, Vol. 27, No. 2, pp. 123–167 (2004).

[20] Spuri, M. and Buttazzo, G.: Scheduling Aperiodic Tasks in Dynamic Priority Systems, Journal of Real-Time Systems, Vol. 10, No. 2, pp. 179–210 (1996).

[21] Buttazzo, G. C. and Caccamo, M.: Minimizing Aperiodic Response Times in a Firm Real-Time Environment, IEEE Trans. on Software Engineering, Vol. 25, No. 1, pp. 22–32 (1999).

[22] Buttazzo, G. C. and Sensini, F.: Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environment, IEEE Trans. on Computers, Vol. 48, No. 10, pp. 1035–1052 (1999).

[23] Dertouzos, M. L.: Control Robotics: The procedural Control of Physical Processes, Information Processing, Vol. 74, pp. 807–813 (1974).

[24] Engblom, J., Ermedahl, A., Sjödin, M., Gustafsson, J. and Hansson, H.: Worst-Case Execution-Time Analysis for Embedded Real-Time Systems, International Journal on Software Tools for Technology Transfer, Vol. 4, No. 4, pp. 437–455 (2003).

[25] Lv, M., Zhang, Y., Deng, Q. and Zhang, J.: A Survey of WCET Analysis of Real-Time Operating Systems, Proceedings of International Conference on Embedded Software and Systems, IEEE Computer Society, pp. 65–72 (2009).

[26] Bygde, S. and Lisper, B.: Towards an Automatic Parametric WCET Analysis, Proceedings of International Workshop on Worst-Case Execution Time Analysis (2008).

[27] Godefroid, P., Klarlund, N. and Sen, K.: DART: Directed Automated Random Testing, Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation, Vol. 40, No. 6, ACM, pp. 213–223 (2005).

[28] Ermedahl, A., Gustafsson, J. and Lisper, B.: Deriving WCET Bounds by Abstract Execution, Proceedings of International Workshop on Worst-Case Execution Time Analysis, pp. 72–82 (2011).

[29] Gustafsson, J., Ermedahl, A., Sandberg, C. and Lisper, B.: Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis using Abstract Execution, Proceedings of Real-Time Systems Symposium, IEEE Computer Society, pp. 57–66 (2006).

[30] Cadar, C. and Sen, K.: Symbolic Execution for Software Testing: Three Decades Later, Communications of the ACM, Vol. 56, No. 2, pp. 82–90 (2013).

[31] : jCUTE, Open Systems Laboratory (online), available from ⟨http://osl.cs.illinois.edu/software/jcute/⟩ (accessed Jul 8, 2018).

[32] : lp_solve reference guide, lp solve (online), available from ⟨http://lpsolve.sourceforge.net/⟩ (accessed Jul 8, 2018).

[33] : The Worst-Case Execution Time (WCET) analysis project, Mälardalen Real-Time Research Centre (online), available from ⟨http://www.mrtc.mdh.se/projects/wcet⟩ (accessed Jul 8, 2018).

[34] : Embedded System Research Group, Seoul National University (online), available from ⟨https://cse.snu.ac.kr/en/research-group/embedded-system-research-group⟩ (accessed Jul 8, 2018).

[35] ARM: ARM Cortex-A9 Technical Reference Manual, Revision:r4p1 (2012).

[36] ARM: Cortex-A9 Floating-Point Unit Technical Reference Manual, Revision:r4p1 (2012).