ACET Based Scheduling of Soft Real-Time Systems: An Approach to Optimise Resource Budgeting

X. Guo, M. Boubekeur, J. McEnery, and D. Hickey

Abstract— The worst-case execution time assumption for scheduling of real-time systems often lead to a waste of resources. In hard real-time systems these types of estimates are essential to guarantee temporal requirements are met. However in soft real-time systems using other measurements, such as average-case timing, to complement the worst-case estimates can lead to better utilisation of resources while ensuring most, if not all, deadlines are met.

In this paper we propose a methodology to optimize resource budgeting by integrating ACET information as a base for scheduling of soft real-time systems. We demonstrate the usability of the approach and illustrate it via a typical Real-Time Java programs.

Keywords—ACET, OCET, Scheduling, Soft Real-Time Systems, Timing Analysis, WCET.

I. Introduction

Currently in real-time systems, in general cost estimations are based exclusively on worst-case execution time (WCET) [1]. This of course is essential in hard real-time systems where missing a deadline can have catastrophic consequences. However there exist real-time systems where an occasional missed deadline is acceptable.

For certain tasks in a real-time system the WCET may overshoot the actual time of a large proportion of the executions of the task. This occurs when one execution requires an abnormally large amount of time relative to the other cases. Consequently budgeting on the WCET leads to a large waste of resources on most executions of the task. In such cases additional information to complement the WCET to calculate more flexible cost estimates would be advantageous, e.g. [2]. Indeed, an accurate measurement of a tasks average-case execution time (ACET) can assist in the calculation of more appropriate deadlines.

Manuscript received August 29, 2007. This work was supported in part by the Science Foundation Ireland and Irish Research Council for Science, Engineering and Technology under the Embark Initiative reference number RS/2005/118.

X. Guo was with the Centre for Efficiency-Oriented Languages, Computer Science Department, University College Cork, Ireland. She is now with the Cork Constraint Computation Centre, Computer Science Department, University College Cork, Ireland (e-mail: gx1@cs.ucc.ie).

M. Boubekeur was with the Centre for Efficiency-Oriented Languages, Computer Science Department, University College Cork, Ireland. He is now with the Cork Complex Systems Lab, Computer Science Department, University College Cork, Ireland (e-mail: m.boubekeur@cs.ucc.ie).

J. McEnery is with the Centre for Efficiency-Oriented Languages, Computer Science Department, University College Cork, Ireland (email: j.mcenery@cs.ucc.ie).

D. Hickey is with the Centre for Efficiency-Oriented Languages, Computer Science Department, University College Cork, Ireland (email: d.hickey@cs.ucc.ie).

Table 1: Task execution times

	Task 1	Task 2	Task 3
BCET	10ms	15ms	20ms
ACET	20ms	30ms	15ms
WCET	70ms	60ms	75ms

For example, let us consider three tasks in a real-time system: task 1, task 2 and task 3. The best, average and worst case execution time estimations for each of them are displayed in Table 1. The gap between ACET and WCET based cost estimates is illustrated in Fig. 1.

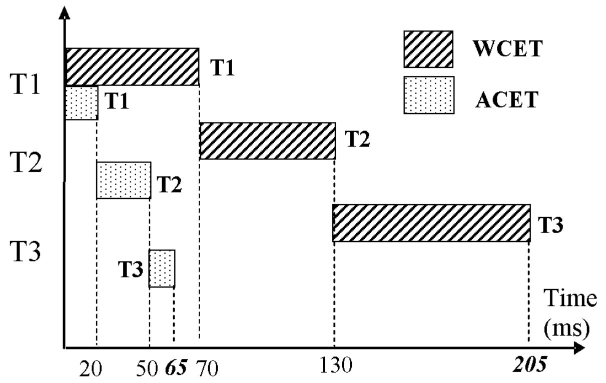

Fig. 1: WCET and ACET cost estimates

The goal of our research work is to improve scheduling by incorporating ACET information. In this paper we describe a methodology for ACET based scheduling of real-time systems.

In order to achieve this we calculate a parameter called OCET (Optimal Case Execution Time). This parameter is automatically obtained using a function which takes as parameters: a user defined acceptable missed deadline percent, WCET and ACET. To determine timing performances (WCET and ACET) we consider two classes of methods: static timing analysis methods and measurement-based methods.

The need for optimal resources budgeting is well researched; it has already been discussed in [3]. One option is to use alternatives to WCET for scheduling. The authors in [4] propose to cut the processing time assigned to a task below its worst-case needed to avoid under-utilization of resources. Within this time an acceptable percent of executions successfully complete, while the remaining computations will be processed later. This can be used for imprecise computation and algorithms that can be interrupted at any time. In [5], the hard and soft real-time performance of sorting algorithms were investigated and compared with their average performance. The author intended to work out how to select the right algorithm for an application based on demand performance criterion. [6] describes the use of decision-theory to optimize the value of computation with uncertain and varying resource limitations. Also in [7], a system for estimating ACET has been developed to aid in resource allocation for *distributed systems*. In industry, in general the estimations of ACET, obtained by measurement, are sometimes used to improve cost estimates.

The remainder of the paper is organised as follows: in section II we give a brief description of the Real-Time Specification for Java. In section III we give an overview of the methods to obtain ACET. Section IV describes a new methodology to incorporate ACET information in real-time scheduling for soft real-time systems. In section V we present the results of our approach on a typical real-time implementation of a sorting algorithm. We end in section VI by giving a discussion of our work and outlining future perspectives.

II. Real-Time Specification for Java (RTSJ)

The RTSJ is designed to allow programmers to engineer large scale real-time systems in a modern, type-safe programming environment. Features such as memory safety, checked exceptions, and a rigorously specified memory model, make Java a good programming language for developing mission critical applications. RTSJ can be viewed as Java SE with new features that provide real-time benefits. Here is a summary of the additional features of RTSJ:

RealtimeThread is used for soft real-time scheduling, this class uses a real-time garbage collector.

- NoHeapRealtimeThread is used for hard real-time scheduling and synchronisation, in this case the garbage collector is not used.
- Twenty-eight new levels of strictly enforced priority levels. The highest priority JRTS thread is truly the highest priority thread in the system and will not be interrupted.
- ScopedMemory memory contains live objects, but these are destroyed immediately when all schedulable objects leave the scope. Scopes allow the developer to control precisely when objects are created and destroyed without the interference of garbage collection.
- ImmortalMemory memory contains objects that are not destroyed until the end of the application.
- Asynchronous event handlers handle external events and allow one to schedule the application's response without spoiling the temporal integrity of the overall system.
- Asynchronous transfer of control allows one to transfer control from one thread to another or quickly terminate a thread.
- High-resolution timers have true nanosecond accuracy.
- Direct physical memory access is safely allowed.

III. Methods to determine ACET

There are mainly two major approaches for ACET and WCET analysis: (1) static analysis of programs and (2) measurement-based techniques.

A. *Automatic static analysis techniques*

Static analysis aim to determine the behaviour of a given task by analyzing the code and its execution environment: analyze the set of possible control flow paths and combine control flow with some model of the hardware architecture (abstract in general) to obtain timing performance. Here we consider two of the main techniques used for automatic ACET calculation.

1) *Lambda-Upsilon-Omega (LUO)*

LUO is a well researched average-case analysis tool, developed by INRIA, France [8]−[10]. It is an academic prototype designed to perform automatic average-case analysis of well-defined classes of algorithms.

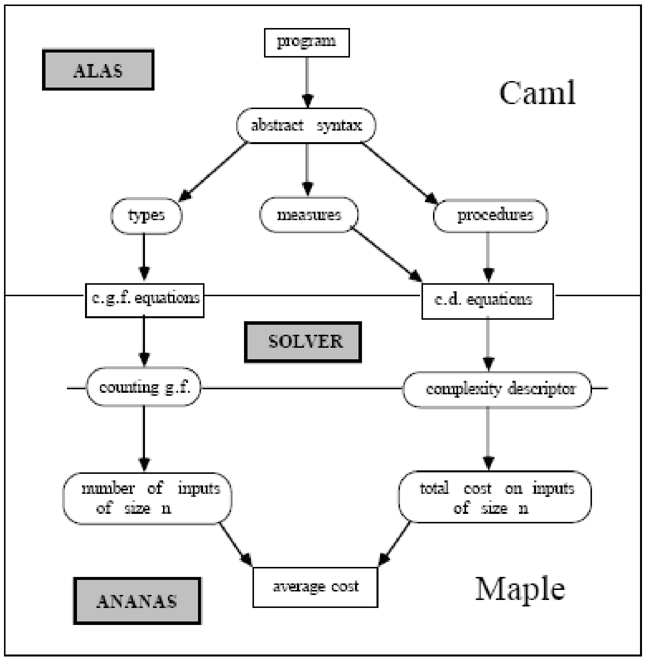

Fig. 2: LUO structure

The LUO system is based on the combination of two ideas:

1. Computing automatically the generating function equations of the program. This is based on the relationship between data structure and algorithm specifications.
2. Extracting the asymptotic form of the generating function coefficients.

The LUO system consists of three major components: the Algebraic Analyzer (ALAS), the SOLVER, and the Analytic Analyzer (ANANAS). Fig. 2 shows how the average cost is derived from a program through the different components.

2) *Modular Quantitative Analysis tool (MOQA)*

MOQA [11]–[13] is a tool dedicated to automatic average-case analysis.

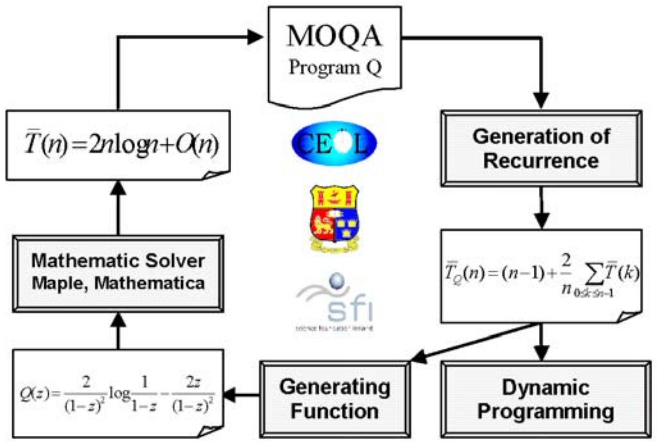
MOQA
Program Q
Generation of Recurrence
$\bar{T}_Q(n)=(n-1)+\frac{2}{n}\sum_{0\le k\le n-1}\bar{T}(k)$
Dynamic Programming
Generating Function
$Q(z)=\frac{2}{(1-z)^2}\log\frac{1}{1-z}-\frac{2z}{(1-z)^2}$
Mathematic Solver Maple, Mathematica
$\bar{T}(n)=2n\log n+O(n)$

Fig. 3: MOQA Process

As shown in Fig. 3, MOQA allows the generation of recurrence equations for the Average-Case Time from MOQA programs. Once a recurrence is obtained, a secondary recurrence can be derived specifying the standard deviation. Fig. 3 also indicates standard approaches which can be followed to either completely solve the recurrence equations through generating functions and a mathematical software package such as Maple, or to obtain information on the average time for inputs within a given size bound through the standard approach of computing the recurrence via dynamic programming ([14]).

B. Measurement based techniques

Measurement-based analysis techniques are still current practice in industry. These techniques do not always give satisfactory results for either ACET or WCET because it is impossible to enumerate all possible (infinitely many) inputs for a program and measure each execution time. Measurement-based analysis techniques raise the question of whether or not there exists a more pessimistic scenario in which the execution time exceeds the WCET observed so far.

One major problem with this approach is that the execution time of basic blocks differs when different input parameters are provided. Also, different inputs cause different paths to be taken to reach basic blocks and the correlation between input and execution time is not obvious.

The worst-case behaviour of a basic block can be exposed easily, but this is generally considered as not being sufficient since it is hard to prove that this is the case. However, when considering ACET, measurement-based techniques offer more accurate results. This is particularly true, as shown in later sections, when considering a well distributed input sample.

For ACET, if the required measurements consider timing of separate program components, combining these measured times to obtain the result for the complete task is a key issue. This is guaranteed by compositionality, which is very important property in program analysis. The essence of compositionality is to derive compositional analyzers, where the analysis of a program can be obtained by composing the analysis of its sub-components.

In general, measurement-based analysis can provide a picture of the actual variability of the execution time. They can also provide validation for static analysis approaches.

IV. ACET BASED SCHEDULING OF SOFT REAL-TIME SYSTEMS

It is often not viable or simply undesirable to schedule tasks using pessimistic WCET estimates. As soft real-time systems are capable of withstanding occasional deadline misses the developer may often choose an arbitrary value for the deadline. The selection of this value may be based on the WCET or simply an educated guess which is then tested to confirm that an unacceptable number of deadline misses are not observed. We will see later that WCET based scheduling alone is not optimal for soft real-time systems which require high resource utilization.

In order to reduce the burden on the developer and to improve the precision of the task deadlines we have developed an ACET based scheduling technique. We propose the use of a new timing analysis variable OCET, Optimal Case Execution Time, as the parameter for scheduling rather than using WCET. In soft real time systems and with an efficient OCET, we can increase resource utilization while minimising the potential number of deadline misses. OCET is calculated using an acceptable deadline miss percent as chosen by the designer, WCET and ACET.

A. Methods to calculate OCET

The acceptable deadline miss rate represents an upper bound on the percent of allowable missed deadlines for a given task. For example, if a task has an acceptable deadline miss rate of one percent this implies that:

1. The OCET is set high enough to ensure that no more than one deadline is missed on average in every one hundred executions of the task.
2. The OCET is set low enough to ensure maximum resource utilization while obeying the acceptable deadline miss constraints.

In the following section we outline the two methods we have used to calculate OCET.

1) *Asymptote Method*

In the asymptote method, the value of OCET is initially set to the value of ACET. The OCET is then gradually increased until it reaches the acceptable deadline miss rate or the WCET. This is implemented using the following equation:

$$\mathrm{OCET} = \mathrm{ACET} + (\mathrm{WCET} - \mathrm{ACET}) * \mathrm{y} \qquad (0 \le y \le 1)$$

From Fig. 4 we can see that as the value of y gradually moves from 0 to 1, the value of OCET moves from ACET to WCET.

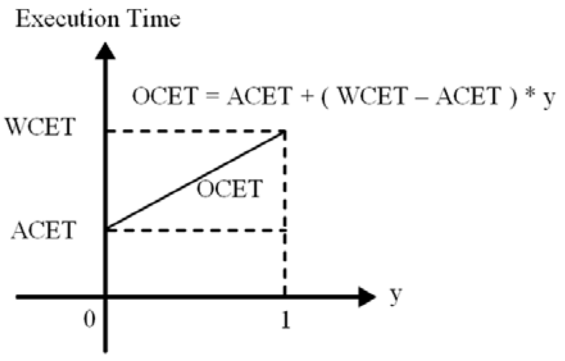

Fig. 4: Asymptote method

We use *givenPercent* to represent the acceptable missed deadline percent provided by the designer. The *missPercent*$_i$ is the actual missed deadline percent which results from using OCET$_i$. Each time we calculate OCET$_i$ we count the number of execution times that are greater than the OCET$_i$ and work out the missed deadline percent (*missPercent*$_i$). The developer chooses an increment value which is used as the initial value of y. The larger the increment value the faster that OCET is found, however the smaller it is the closer it will be to its theoretical value. Therefore it is vital, while using this algorithm, to choose an appropriate increment value.

2) *Binary Search Method*

An efficient method to obtain the OCET is to use a binary search algorithm. This is achieved by applying the following equation where *UpperBound* and *LowerBound* are set to WCET and ACET respectively:

$$OCET_n = \frac{UpperBound + LowerBound}{2}$$

Using OCET$_1$ as the deadline we execute the task noting the number of deadline misses. This value is recorded as a percent of the total number of executions and stored as missPercent$_1$. If missPercent$_1$ is equal to the acceptable missed percent (givenPercent), OCET$_1$ is chosen as the OCET.

However, if missPercent$_1$ is less than the acceptable missed percent, then we know that the OCET must exist somewhere in the range between the LowerBound and OCET$_1$. Therefore we can eliminate the other half of the range from our search by setting UpperBound to OCET$_1$. In case that missPercent$_1$ is greater than the acceptable missed percent, we proceed in a similar way by setting LowerBound to OCET$_1$. This process is illustrated in Fig. 5. We simply repeat the search using the same technique by applying the equation.

Continuing this binary search we will get OCET$_2$, OCET$_3$, OCET$_4$, …, OCET$_i$ … as well as missPercent$_2$, missPercent$_3$, missPercent$_4$, …, missPercent$_i$, …, until the missPercent$_k$ is equal to the givenPercent. In this case OCET$_k$ is chosen as the OCET. If missPercent$_k$ does not equal givenPercent but is within a pre-defined acceptable margin then we stop searching. In this case if the missPercent$_k$ < givenPercent then the OCET is set to OCET$_k$. If the missPercent$_k$ > givenPercent OCET is set to OCET$_{k-1}$.

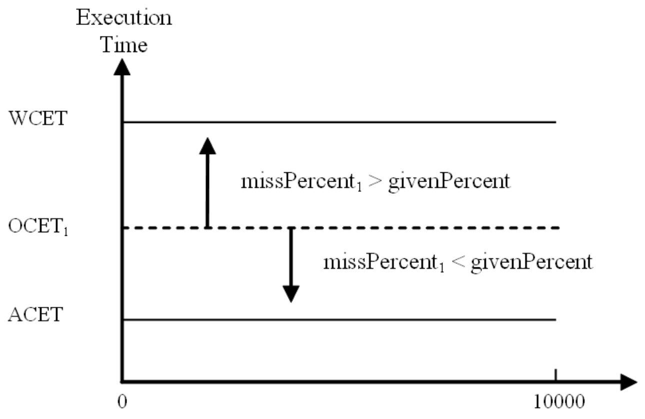

Fig. 5: Binary search method

Using either of the two methods described above we can obtain the OCET if we know the ACET, WCET and the acceptable missed deadline percent. In our example the binary search method has been applied for illustration purposes to a typical Real-Time Java sorting program.

B. *ACET based scheduling for RTSJ*

The Real-Time Specification for Java (RTSJ) [15] provides an integrated approach to scheduling periodic threads and monitoring their CPU execution. In our study we use a PeriodicParameters object which takes a start time, period, cost, deadline, overrun handler and miss handler as parameters.

PeriodicParameters are a subclass of the ReleaseParameters class. The parameter deadline is the latest permissible completion time measured from the release time of the associated invocation of the periodic real-time thread. The missHandler is invoked if the *run* method of the periodic real-time thread is still executing after the deadline has passed. The parameter cost here presents the processing time per release and the overrunHandler is invoked if an invocation of the periodic real-time thread exceeds cost in the given release. It supports a cost enforcement model and also a deadline monitoring model.

As we get the OCET based on the execution time, we can assign it to the cost parameter and count the overrun real-time threads to see how the OCET influences the execution of the real-time thread. The obtained OCET can also be used as the value of the deadline parameter.

However, during our experiments we found that the overrunHandler is released when cost overruns occurred but the real-time thread remains executable until the execution on the list finished. When a deadline was missed, the missHandler is released and the real-time thread is automatically descheduled. The missHandler must reschedule the real-time thread by calling the SchedulePeriodic method, allowing execution to continue. After calling the SchedulePeriodic method, execution on the list which did not finish in the former period will continue to execute in the next period until it finishes. This means one time execution of a list may cause several deadline misses.

Since we require the number of cases that do not complete execution within OCET, we choose to use the costoverrun handler in our study. We set the cost within the OCET value and assign values to deadlines ensuring that all executions complete within their deadlines.

In our experiments we used linear time memory areas (LTMemory) which are scoped memory areas that are guaranteed by the system to have linear time allocation and are not subject to garbage collection.

V. OCET SCHEDULING EXAMPLES

In the previous section we have described the methods to determine the ACET and OCET of a given task. In this section we will demonstrate the usability of the method within RTSJ, considering quicksort and quickselect as the example algorithms using the binary search method described in section IV part A to calculate the OCET. The advantages and disadvantages of using OCET for RTSJ scheduling will also be examined.

We execute our quicksort and quickselect tasks on samples of 10,000 randomly generated lists on the Java RTS, which is Sun Microsystems implementation of the RTSJ. The platform chosen for the examples was a Sunfire V240 running Solaris 10. The worst-case and average-case execution times of the algorithms are obtained using the measurement based approach as described in part B of section IV. The OCET is then calculated using the ACET, WCET and the acceptable missed percent, where the latter is provided by the designer.

A. *Generating input data*

The quality of the experiments are highly dependent on the distribution of the data in each of our lists. We choose Jakarta Commons Math [16] which is a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language. The Commons Math random package includes utilities for generating random numbers.

The random data we generate using Jakarta Commons Math makes a well balanced distribution for the input. Fig. 6 shows the timing results that execute (a) quicksort and (b) quickselect algorithms in our experiments as examples. Maintaining a large sample and list size minimises the possibility of jitter.

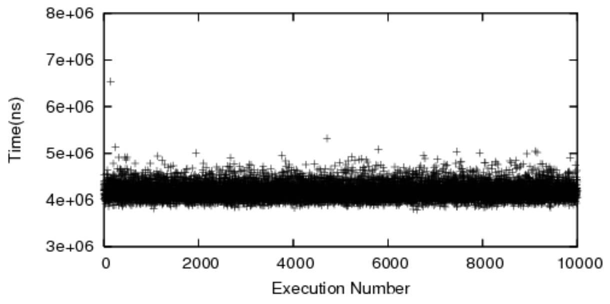

Fig. 6 (a): Execution times distribution for quicksort

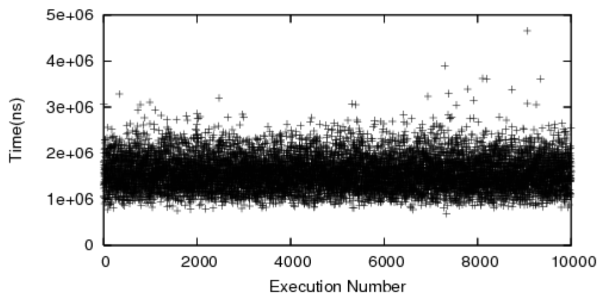

Fig. 6 (b): Execution times distribution for quickselect

B. *Quicksort*

The quicksort algorithm, developed by C.A.R. Hoare [17], is one of the simplest and most efficient algorithms for sorting. The executions of the algorithm fall into three steps:

1. Pick an element, which is called the pivot, from the list.
2. Reorder the list so that all elements which are less than the pivot come before the pivot, all elements greater than the pivot come after it. Elements which are equal to the pivot can go either way. After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

Here is a sample for Real-Time Java quicksort code:

```
private class QuickSort extends RealtimeThread{

  public void run(){
    algSort(list,0,count-1); }

  public int part(int a[], int low, int high){
    int pivot, p_pos, I;
    p_pos = low;
    pivot = a[p_pos];
    for (I = low + 1; I <= high; i++){
      if (a[i] > pivot){
        p_pos++;
        swap(a, p_pos, i);
      }
    }
    swap(a, low, p_pos);
    return p_pos;
  }

  private void algSort(int a[], int low, int high){
    int pivot;
    if (low < high){
      pivot = part(a, low, high);
      algSort(a, low, pivot – 1);
      algSort(a, pivot + 1, high);
} } }
```

We developed a sample Real-Time Java application to execute the quicksort algorithm. We use real-time threads in our implementation. A real-time thread is one kind of schedulable object which associates release, scheduling, memory and processing group parameters. Release parameters, among them, characterize how often a schedulable object is released and provide an estimate of the worst-case processor time needed for each release, and a relative deadline by which each release must have completed.

1) *Calculation of OCET*

While executing the application we measure the ACET and WCET. OCET is determined by the WCET, ACET and the acceptable missed percent. OCET values are calculated using the binary search method described earlier in section IV part A. Here we examine the WCET, ACET and OCET on variable sized lists while maintaining a fixed acceptable missed deadline percent.

Fig. 7 shows the worst, average and optimal case execution times obtained in our experiments. It can be observed that there is a significant gap between the WCET and both the ACET and the OCET. This confirms that in such systems optimal-case execution time leads to a large saving of resources while minimising deadline misses to an acceptable level.

It should be noted in Fig. 7 that the ACET and the OCET displayed are straight line graphs however the WCET graph is jagged. It is widely accepted that the WCET is often unlikely to occur even when a significantly large random input sample is used. Even if the input which produced the WCET for the given algorithm is known and used it may not be possible to determine if that execution was indeed the worst case. This is caused by the complexity which arises if the underlying processor architecture uses speculative components such as caches and pipelines. Due to the difficulty involved in proving that the actual WCET has been observed, many systems are scheduled using the worst observed execution time. For hard real-time systems it is often necessary to add a safety margin to the observed WCET in order to guarantee deadlines are met. The intricacy involved in calculating the true WCET is another motive for looking at an alternative scheduling foundation.

It can be seen from Fig. 7 that OCET is closely coupled with the ACET rather than the WCET. This demonstrates the significance of using ACET in scheduling for soft real-time systems as opposed to most real-time methodologies which are based merely on WCET values.

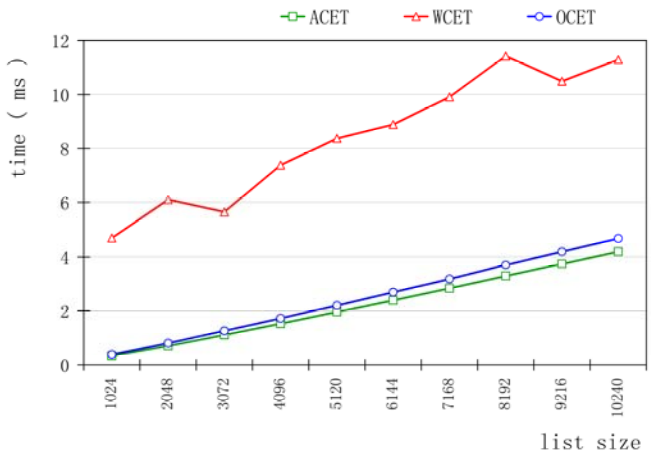

Fig. 7: Comparison of ACET, WCET and OCET

2) *Analysis of the missed deadline percents*

In this section we analyse the relationship between the acceptable missed deadline percent and the OCET. We also test to ensure that the actual number of missed deadlines does not exceed the threshold we have set.

The relationship between the acceptable missed deadline percent and the corresponding OCET values are presented in Fig. 8. As the acceptable missed deadline percent moves towards zero the OCET tends towards WCET. Conversely as the acceptable missed deadline percent increases the OCET tends towards the ACET which in this case is 0.706ms. As the WCET is 6ms its inclusion in the graph would skew the effectiveness of the graph and has therefore been omitted.

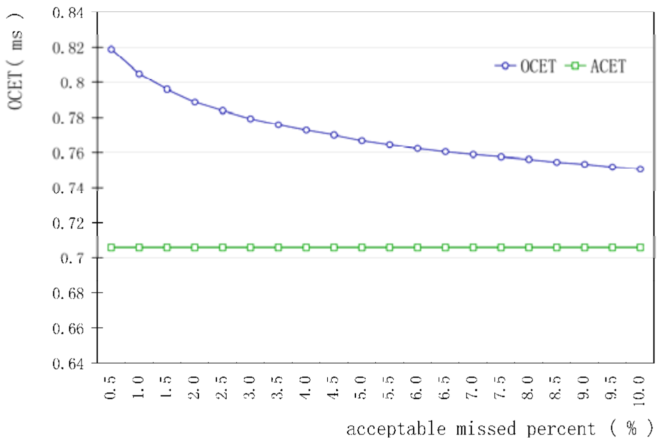

Fig. 8: Relationship between acceptable missed percent and OCET

Fig. 9 presents the acceptable missed deadline percent and the actual missed deadline percent we obtain while calculating the OCET. In each instance the actual number of missed deadlines is less than or equal to the acceptable missed deadline percent.

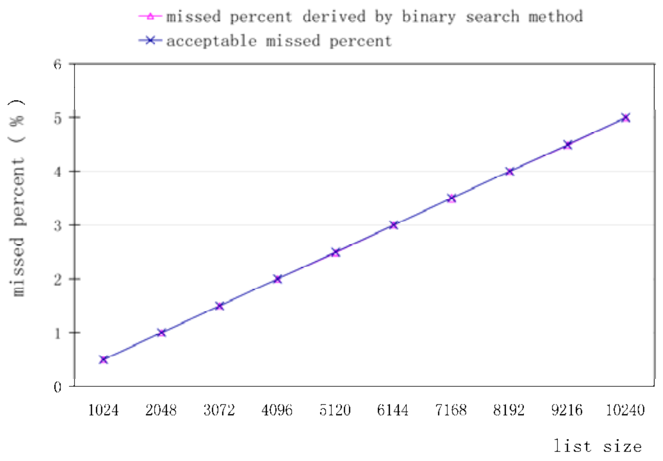

Fig. 9: Missed percent for OCET calculation

3) *Performance of OCET scheduling*

The derived OCET is used as the cost value in the periodic real-time thread to execute the quicksort algorithm and count the number of times the thread overruns. The number of overruns is used to calculate the actual missed deadline percent as observed when using the OCET as the cost value. Similarly, the actual missed deadline percent should be equal to or less than the acceptable missed deadline percent. Fig. 10 shows the acceptable missed deadline percent and the actual missed deadline percent as observed when scheduling using the OCET. It can be seen in the graph that the missed percent obtained by scheduling with the OCET is less than the acceptable missed percent.

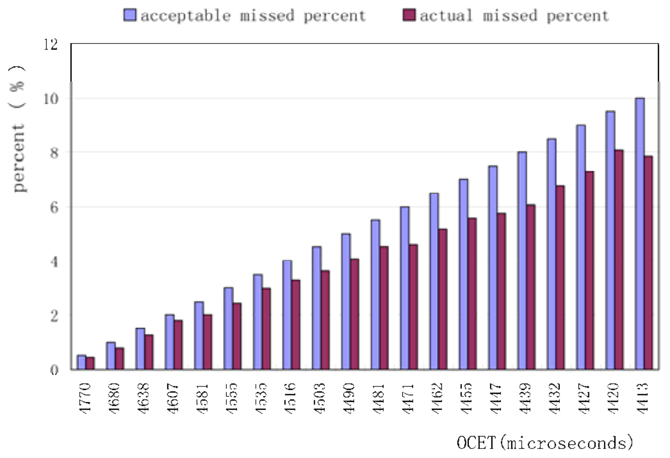

Fig. 10: Acceptable missed percent vs actual missed percent

C. *Quickselect*

Our new approach for ACET based scheduling has also been used on quickselect algorithm as example. Quickselect is one of the most efficient algorithms for quickly finding the *k*-th smallest element of an unsorted list of n elements. It is trivial with pre-sorted data to find the *k*-th smallest element, but it is harder when the data is not sorted. Quickselect works by sorting only the areas it needs to find the desired element. After a quickselect, the data will be roughly sorted and the desired element will be in its correct position. To implement a quickselect we carry out the following steps:

1. Choose a pivot element.
2. Move the pointer from the left of the list to the right, checking each element in turn. If an element whose value is less than the value of the pivot it is moved to the less-than value list. Similarly, if it is greater than the value of the pivot it is moved to the greater-than value list.
3. If *k* is less than or equal to the size of the less-than list, then the *k*-th smallest element you are searching for resides in the less-than list. In this case we return to step 1, using the less-than list as our full list. However, if *k* is greater than the size of the full list minus the size of the greater-than list, then the *k*-th smallest element exists in the greater-than list. In this case we return to step 1, using the greater-than list as our full list and reducing the value of *k* by the size of the full list minus the size of the greater-than list. If neither is true, then the *k*-th smallest element is the pivot value.

The input data for quickselect, consisting of a list and value *k* are randomly generated using Jakarta Commons Math [16]. We use real-time threads in our implementation to execute the quickselect algorithm, the same manner as the previous example.

Fig. 11 presents the worst and average case execution times of the quickselect algorithm as observed in our experiments along with the optimal case execution time as calculated using the binary search method. It can be seen that there is a significant gap between the WCET and both the ACET and the OCET. The OCET is closely coupled with the ACET rather than the WCET.

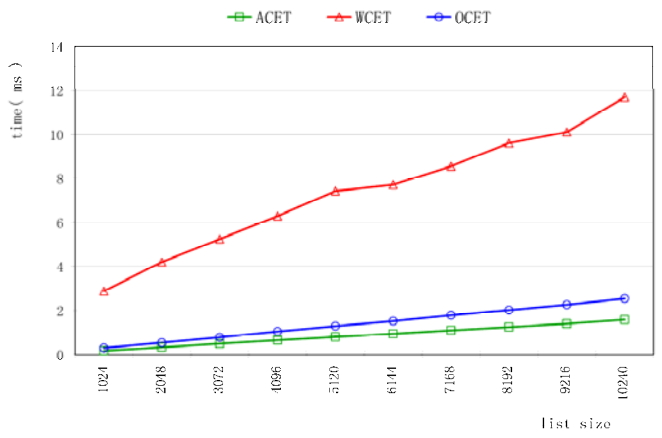

Fig. 11: Comparison of ACET, WCET and OCET

Fig. 12 shows the relationship between the acceptable missed percent and the corresponding OCET values. In the case of the quickselect algorithm, the figure shows that when the acceptable missed deadline percent moves towards zero the OCET tends towards WCET. As the acceptable missed deadline percent increases, the OCET tends towards the ACET. Again, the WCET has been omitted form Fig. 12, as it is greater than 4ms and its inclusion would skew the effectiveness of the graph.

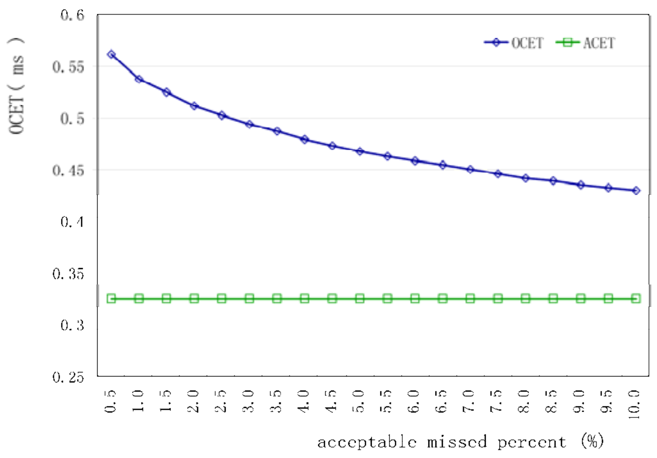

Fig. 12: Relationship between acceptable missed percent and OCET

Fig. 13 presents the acceptable missed deadline percent and the actual missed deadline percent of quickselect algorithm we obtain while calculating the OCET. In each case the actual number of missed deadlines is less than or equal to the acceptable missed deadline percent.

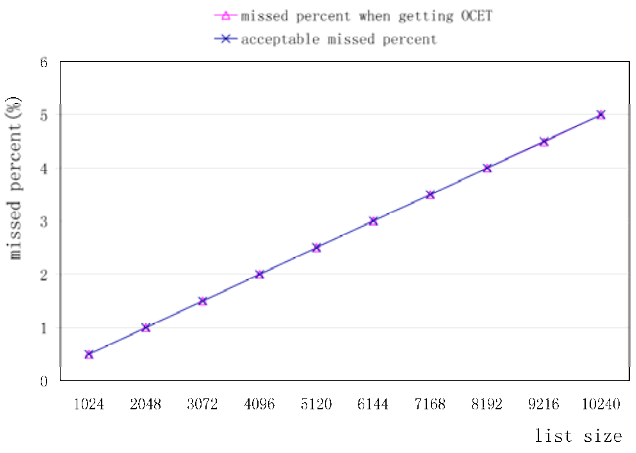

Fig. 13: Missed percent for OCET calculation

Fig. 14 describes the relationship between the acceptable missed deadline percent and the actual missed deadline percent when scheduling using the obtained OCET for the quickselect algorithm. It can be observed that all of the actual missed deadline percentages are equal to or less than the acceptable missed deadline percent.

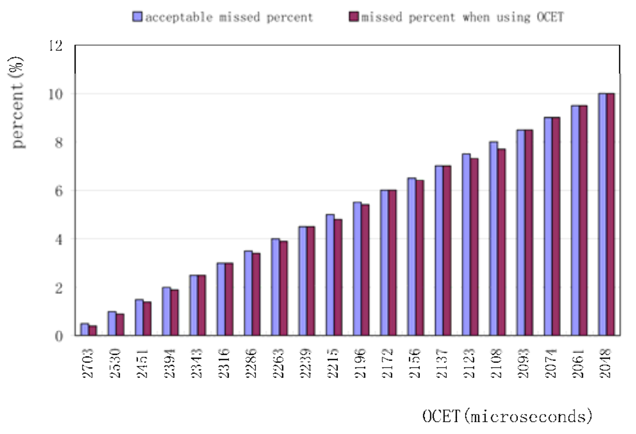

Fig. 14: Acceptable missed percent vs actual missed percent

VI. Conclusions and Future Work

In this paper we have outlined a methodology we have developed for ACET based scheduling of soft real-time systems. This has been illustrated through an evaluation study using well known sorting and selection algorithms as examples. The results demonstrate the benefits of this method for soft real-time scheduling.

In future research we intend to evaluate our approach using different scheduling algorithms as shown in [18]. We intend also to integrate static techniques to derive OCET. This can be achieved using probability density functions to obtain the distributions of executions times.

Acknowledgment

We would like to thank Sun Microsystems for providing us with the hardware, software and technical information which has made this research possible.

References

[1] H. Jamal, Z. A. Khan and M. M. Rahmatullah, “FPGA Based Hardware Scheduler for Multiprocessor Systems”, WSEAS Applied Computing Conference (ACC '08), Istanbul, Turkey, May 27-30, 2008.

[2] D. Mittermair and P. Puschner, Which Sorting Algorithms to Choose for Hard Real-Time Applications. In Proc. Euromicro Workshop on Real-Time Systems, p. 250-257, Toledo, Spain, June 1997.

[3] P. Puschner, A. Burns, “Time Constrained Sorting – A Comparison of Different Algorithms”, 11th Euromicro Conference on Real-Time Systems, 1999.

[4] P. Puschner, A. Burns, Time Constrained Sorting- A Comparison of Different Algorithms, 11th Euromicro Conference on Real-Time Systems, 1999.

[5] P. Puschner, Real-Time Performance of Sorting Algorithms, Real-Time Systems, Volume 16, Number 1, 1999 , pp. 63-79(17).

[6] Eric Horvitz, Reasoning Under Varying and Uncertain Resource Constraints, In Proc. of the 7th National Conference on Artificial Intelligence, pages 11-116, Minneapolis, MN, USA, 1988. Morgan Kaufmann, San Mateo, CA.

[7] Vivek Sarkar, “Determining Average Program Execution Times and their Variance”, ACM SIGPLAN 1989 Conference on Programming language design and implementation.

[8] P. Flajolet, J. S. Vitter, “Average-Case Analysis of Algorithms and Data Structures, Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity”, Elsevier, 431-524, 1990.

[9] P. Flajolet, B. Salvy, P. Zimmerman, “Automatic average-case analysis of algorithms”, Theoretical Computer Science 79, 37 - 109, 1991.

[10] P. Flajolet, R. Sedgewick, An Introduction To The Analysis of Algorithms, Addison Wesley, 1995.

[11] M. P. Schellekens, “A Modular Calculus for the Average Cost of Data Structuring”, Springer book to appear 2008, 250 pages.

[12] M. P. Schellekens, “A randomness preserving product operation”, Extended version accepted for publication on ENTCS, Elsevier's series "Electronic Notes in Theoretical Computer Science", 2007.

[13] M. Boubekeur, D. Hickey, J. Mc Enery and M. Schellekens, "A Modular Average-Case Timing of Real-Time Languages", WSEAS Transactions on Computers, Issue2, Volume 1, December 2006, ISSN: 1991-8755.

[14] A. Aho, J. Hopcroft and J. Ullman. Data structures and algorithms. Addison-Wesley Series in Computer Science and Information Processing, Addison-Wesley, 1987.

[15] http://www.rtj.org/

[16] http://jakarta.apache.org/commons/math/

[17] Hoare, C. A. R. Algorithm 65, FIND. Comm. Assoc. Comput. Mach. V4 321-332, 1961.

[18] V. Salmani, M. Naghibzadeh, et al., “Performance Evaluation of Deadline-based and Laxity-based Scheduling Algorithms in Real-time Multiprocessor Environments”, The 6th WSEAS International Conference on Systems Theory and Scientific Computation (ISTASC'06), August 18-20, 2006, Elounda, Agios Nikolaos, Crete Island, Greece.