Towards Modular Average-Case Timing in Real-Time Languages: An Application to Real-Time Java

M. Boubekeur*, D. Hickey†, J. Mc Enery‡, M. Schellekens†
Centre for Efficiency-Oriented Languages (CEOL)
Department of Computer Science
National University of Ireland, Cork

Abstract: In certain real-time systems worst-case execution time estimates 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. We propose a methodology to integrate modular average-case timing in Real-Time languages. Previously statically determining average-case time, if possible, required rigorous mathematical techniques. Our approach, which is based on a new programming paradigm called MOQA and is built on Real-Time Java, simplifies the process and allows for automation.

Key–Words: Real-Time Systems, Time Analysis, Average-Case Timing, Modular Timing, JRTS, RT-MOQA

1 Introduction

Currently in Real-Time Systems, in general, cost estimations are based exclusively on worst case execution time (WCET). This of course is essential in hard real-time systems where missing a deadline can have catastrophic consequences. However in soft real-time systems 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. 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. An accurate measurement of a task's average case execution time (ACET) can assist in the calculation. The need for this in real-time systems has already been discussed in [1]. In industry the estimations of ACET, obtained by measurement, are already sometimes used to improve cost estimates.

ACET analysis is one of the most thoroughly studied areas of Computer Science. As demonstrated in [2] it is notoriously difficult involving a variety of techniques which, typically, do not allow for automation. Currently algorithms must be analysed on a case-by-case basis and it is not feasible in general to express the ACET of algorithms via a recurrence equation, though excellent techniques are available to solve these recurrences.

One of the main properties of ACET which simplifies modular analysis of programs is IO-compositionality [4]. We explain this concept in Section 2 along with an explanation of why this property does not hold in general for WCET.

We do not advocate of course to eliminate WCET but to complement it with ACET information. Our aim is to develop a framework that satisfies two criteria: real-time constraints and modular average-case timing. To do so we will use the general MOQA language developed in Java which allows modular timing. MOQA is then transformed and adapted to the Real-Time Specification for Java (RTSJ) [3] criteria. The obtained result, RT-MOQA, is the first RT-framework dedicated to modular average-case timing of real time systems.

2 Modular Average-Case Timing

The modularity of MOQA programs is guaranteed by its compositionality property. Compositionality is a traditional notion of Programming Language Semantics which guarantees that the specification of each program can be (inductively) defined in terms of the specifications of the program's basic components.

In a similar spirit, MOQA involves a novel notion

*Funded by SFI, Industrial Collaboration Award.
†Funded by SFI, Investigator Award 02/IN.1/181.
‡Funded by the IRCSET Postgraduate Research Scholarship RS/2005/118.

of compositionality in a RT-context with respect to a time measure T: compositionality of a language with respect to T is informally defined as a property which guarantees that the time of each program can be determined from the time of its basic components. The notion is so natural that it may lead one to conjecture it holds for any time measure. We will see however that this is not the case in general. Yet compositionality is a central factor in determining the speed of the program to an excellent degree of precision because it is a computational necessity to determine this speed directly from the accurate speeds of the components.

To simplify the presentation we will focus in what follows on measuring the number of comparisons made during the execution of a comparison based algorithm. This is consistent with standard approaches. Of course other timing parameters such as assignments and swaps can be used to fine-tune matters.

We assume that every input has a given size *n* and that there is a finite set *In* of inputs. Typically these inputs are defined with respect to a given data structure. The execution time of a program P on an input *I* with respect to a time measure T is denoted by $T_P(I)$. The notation $T_P(n)$ indicates the restriction of the time analysis to the input set *In*.

The worst-case time of P for inputs of *A* is defined by:

$$T_P^W(A) = maximum\{T_P(I) | I \in A\}$$

We recall the standard definition of average-case time of P on inputs from *A*:

$$\overline{T}_P(A) = \frac{\sum_{I \in A} T_P(I)}{|A|}$$

2.1 Overview of MOQA

Firstly, without being overly precise, we will define compositionality as follows: $P_1;P_2$ indicates the sequential execution of program P_1 followed by the program P_2. Compositionality with respect to a time measure T holds if for every pair of programs $P_1;P_2$ the following property holds:

$$T_{P_1;P_2} = T_{P_1} + T_{P_2}$$

This definition however is a bit naive. Consider for instance the case of the average-time where the above notion of compositionality reduces to:

$$\overline{T}_{P_1;P_2}(n) = \overline{T}_{P_1}(n) + \overline{T}_{P_2}(n)$$

The following example shows that this notion of compositionality does not hold in general for the average time measure.

Example 1: Composition of Sorting Algorithms: Consider the sorting algorithms quick sort and merge sort, denoted by Q and M. Quick sort is well known to have O(nlogn) average-case comparison-time and $O(n^2)$ worst-case comparison time. We consider a quick sort version for which the worst-case time $O(n^2)$ occurs for sorted lists. It is well-known that:

$$\overline{T}_Q(n) \in O(nlogn) \text{ and } \overline{T}_M(n) \in O(nlogn)$$

Thus:

$$\overline{T}_M(n) + \overline{T}_Q(n) \in O(nlogn)$$

However:

$$\overline{T}_{M;Q}(n) \in O(n^2)$$

The problem is that merge sort is passing on a worst-case input of quick sort as the only input to the quick sort algorithm. A more fine-tuned book keeping of the outputs produced during a composition is required in order to remedy this problem. In order to set the stage for such fine-tuned book keeping we introduce the useful notion of a multi-set below and a related more refined definition of compositionality: that of IO-compositionality (Input-Output compositionality).

Multi-sets are used to record program outputs and their frequency of occurrence. A multi-set is a finite set-like object in which order is ignored but multiplicity is explicitly significant. Thus, contrary to sets, multi-sets allow for the repetition of elements. Therefore, multi-sets {1, 2, 3} and {3, 1, 2} are considered to be equivalent, but {1, 2, 2, 3} and {1, 2, 3} differ. We refer to the number of times an element *x* occurs in a multi-set as the *multiplicity M(x)* of the element *x*, i.e. *M(x)* = number of *x*-occurrences in the given multi-set.

We will use multi-sets for the purpose of output book keeping and use the following notation:

$O_P(A)$ = the output multi-set for a finite input set A.

In MOQA the output multi-set contains all the possible states each of the data structures can be in. We refer to these multi-sets as *random bags*.[1]

Example 2: Random Bags:
Let L_n = the *n*! lists of size *n*. Consider a program P which extracts the first two elements of a three element list. A list of two elements *a*, *b*, when identified up to order-isomorphism can only consist of two possible pairs: L_2 = {(a, b), (b, a)}. Below we illustrate that we obtain three copies of L_2 in the random bag produced::

[1] Bag is an alternative name used for multi-set.

(**1, 2**, 3) (**1, 2**) (**1, 3**, 2) (**1, 3**)
(**2, 1**, 3) (**2, 1**) (**2, 3**, 1) (**2, 3**)
(**3, 1**, 2) (**3, 1**) (**3, 2**, 1) (**3, 2**)

$$O_P(L_3) = \{(L_2, 3)\}$$

We can now give a more precise definition: a time measure T is **IO-compositional** iff for every input-set I:

$$T_{P_1;P_2}(I) = T_{P_1}(I) + T_{P_2}(O_{P_1}(I))$$

As remarked in the introduction, compositionality has the power to simplify analysis. Yet compositionality is thus far not an RT-tool of the trade. This is due to the fact that WCET is not IO-compositional [4][5]. It is easy to verify that in general the following inequality holds:

$$T^W_{P_1;P_2}(I) \leq T^W_{P_1}(I) + T^W_{P_1}(O_{P_1}(I)) \quad \star$$

However, in general

$$T^W_{P_1;P_2}(I) \neq T^W_{P_1}(I) + T^W_{P_1}(O_{P_1}(I))$$

The non-compositionality of worst-case time is reflected by the fact that for real-time languages, WCET is typically approached in practice in a non-exact way, i.e. by relying on upper bounds, in particular by using the right hand side of equation $\star$ above. Fortunately, there are various techniques to determine the WCET to a reasonable degree (e.g. [7]).

2.2 The MOQA Language

MOQA is a general "data-structuring language" introduced in [6]. MOQA has good generality and can incorporate most algorithms which restructure data, in particular of course sorting and searching algorithms.

In [5], in addition to IO-Compositionality, a formal notion of a "randomness preserving" operation is introduced. Randomness preservation is studied by Knuth in a pioneering paper ([8]). The lack of randomness preservation of algorithms such as heap sort is identified in [9] as a fundamental obstacle towards the derivation of a time-recurrence and hence prevents automated average-case analysis.

[5] puts the notion of randomness preservation on a formal basis and the main result guarantees that all MOQA programs are randomness preserving. Combining this result with the fact that randomness preservation implies compositionality, we obtain that all MOQA programs allow for the generation of time-recurrences and the techniques of Figure 1 apply. Note in [4] and [5] a solution for the average-time is obtained for a MOQA version of heap sort.

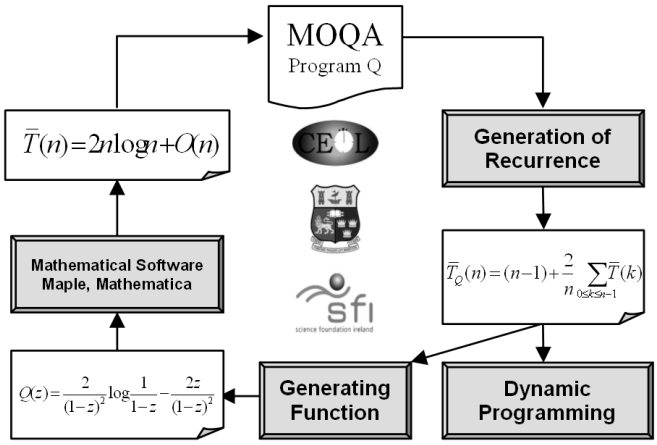

Figure 1: MOQA Process

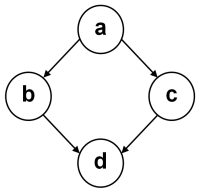

Figure 2: Series-parallel partial order

2.2.1 Example

The average-case time measure of quick sort is not only IO-compositional but also distribution-preserving. As a result it is possible to detect great regularity in the execution and clearly identifiable patterns in timing of quick sort [5].

It is possible, though we will not do this here, to derive from the this type of distribution preservation the well-known recurrence equation for quick sort's average time, which leads to the O(nlogn) solution.

2.2.2 Data Structures

The basic data structures in MOQA are Series-Parallel Partial Orders (SPPO) which is a partial order that only allows nodes to be in series, denoted by $\otimes$, or in parallel, denoted by $\|$. This means no cycles are allowed in MOQA's data structures. Figure 2 would be represented in series-parallel notation by a $\otimes(b\|c)\otimes$ d.

With the guarantee that operations on these data structures are randomness preserving and uphold the series parallel property, the ACET analysis of programs is greatly simplified - random bags can be determined and multiplicities calculated leading to easier derivation of the time.

3 Specification of the Real-Time Language for Modular-Timing

We incorporate the theoretical research in terms of modular quantitative analysis, in particular timing analysis, into Java for Real-Time Systems (JRTS) provided by Sun Microsystems. JRTS is an implementation of the RTSJ which incorporates the benefits of Java along with rigorously specified real-time features and so is suitable for engineering large scale real-time systems. In spite of these benefits, predictable average-case timing tools are not available for JRTS.

In this section we give an overview of how the general MOQA language developed in Java which allows Modular timing is transformed and adapted to the RT-languages criteria in JRTS.

3.1 Real-Time Features

We use JRTS's scoped and immortal memory as a reference memory model in RT-MOQA. This liberates the programmer from the unpredictability associated with garbage collection.

In the context of RT-MOQA loops and recursion are managed rather than forbidden as they are in other RT-languages. This is discussed in section 3.2.

For scheduling we are developing a strategy to allow us to move timing estimates between average and worst case depending on the nature of the system and the variation in the execution of the handled task. Knowing both the WCET and ACET also allows us to redesign certain algorithms in order to narrow the gap between the two measures, a situation advocated as advantageous for scheduling tasks in real-time analysis in [10].

The design of data structures in RT-MOQA is very suitable to concurrency. Concurrent tasks can access shared RT-MOQA data structures simultaneously for reading. However write access on the same part of a data structure must be synchronised.

Other important features, such as asynchronous events, specified in the RTSJ implementation are not considered at this stage.

3.2 Timing Predictability in RT-MOQA

In this section we discuss the unique features of RT-MOQA which achieve the enhanced predictability. We also discuss how RT-MOQA programs can be analysed automatically to determine this information.

3.2.1 Data Structures for Automated Timing

SPPOs in RT-MOQA can be classified using inductively defined types. This is necessary where a control statement produces many different structures in a random bag. An inductively defined type (IDT) represents a class of SPPOs in terms of base cases and constructors used for building the SPPO. An example of an IDT is heap-ordered trees:

$$HOT ::= null|node$$
$$HOT ::= node(HOT||HOT)$$

Note that although in the current version of MOQA SPPOs are the data structures, future implementations may include other data structures as long as the operations on them are random bag preserving.

3.2.2 Control Statements

Because of the nature of the data structures in RT-MOQA we can guarantee that the control statements in programs can have deterministic properties. This is very important for performing automatic static program analysis and for incorporating MOQA into real-time languages. For the latter it is well known that most real-time programs avoid the use of recursion and unbounded loops because they make estimating WCET very difficult. Despite this, in for example RT-Java, there are no restrictions on how a programmer can implement an algorithm using recursion or loops. With RT-MOQA however there are very clear rules as to how such algorithms can be incorporated and how cost estimates can be made.

***if-else* Statements:** There are strategies for calculating probabilities for different branches in a program's execution, for example when comparing data structure labels or when checking data structure sizes.

In general any type of conditional is allowed as along as random structure preservation is upheld. However there are some conditionals which cannot have probabilities automatically calculated, e.g. those which are dependent on the range of values input. For such cases the probabilities have to be provided by the user.

Recursion: For the purposes of real-time, it is possible in RT-MOQA to ensure that recursion is convergent, i.e. starting with some data structure, the operations executed on it guarantee that one of the base cases will be reached. For example, say we have a recursive algorithm with a parameter which is a list and a base case which checks if the size of the list is 1 or 0. Then each recursive call has to take a smaller list as input meaning the base case will be reached and termination is assured.

In order to allow MOQA recursion to be automatically analysed there are code templates, an example of which is shown in Figure 3.

Alg(x)
$p_1(x)$
if $cb_0(x)$ **then** $b_0(x)$
else if ...
else if $cr_0(x)$ **then** Alg_Calls$_0$(x')
else if ...
$p_2(y)$

Figure 3: Recursive Code Template

Loops: *for* loops are easily handled in static time analysis because their execution is bounded. *while* loops on the other hand in general are not easily analysable because determining the number of executions is difficult. In RT-MOQA we again only allow *while* loops that have conditions and bodies with operations on the data structures. Therefore we can guarantee termination and also derive timing information. Like recursion, RT-MOQA enforces templates on the structure of *while* loops that allows a similar representation of the required average execution time.

3.3 Automatic Timing Analysis

Figure 1 gives an overview of the automatic analysis of RT-MOQA programs. In brief, an analysis would proceed as follows:

1. The user selects a class and method from where the analysis begins. The code can be annotated with information to aid the analysis, e.g. probabilities which cannot be calculated automatically.
2. Soot ([12]) builds a call graph and a control-flow graph (CFG) for each method.
3. Distri-Track traverses the call graph and CFGs.
   (a) A *timed-state graph* is built for each method invocation based on the its CFG.
   (b) Each CFG branch has a probability calculated.
   (c) Edges representing iteration have the number of iterations attached.
   (d) Each node stores the time for the corresponding statement executed on the random bag at that point in the analysis.
4. The recurrence equation for the ACET is built from the timed-state graph.

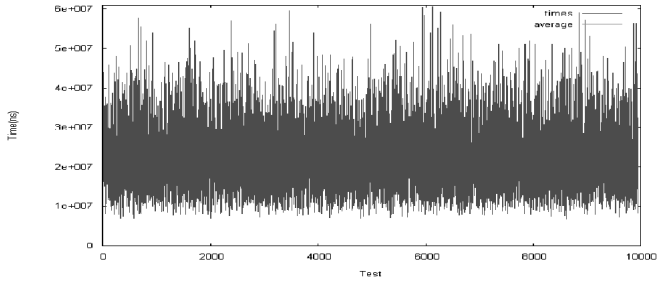

Figure 4: Input data distribution

5. Mathematical software or dynamic programming can solve the recurrence to get the ACET.

4 RT-MOQA Example and Preliminary Evaluation

In this section we give an evaluation of the RT-MOQA theoretical results for average-case timing by an experimental analysis of quick select.

Quick select is one of the simplest and most efficient algorithms in practice for finding specified order statistics in a given sequence. It uses the usual partitioning procedure of quick sort. It differs in that it recurses on one of the partitions containing the order statistic rather than on both.

Quick select in RT-MOQA is coded as a real-time thread which executes in scoped memory. The main operation of RT-MOQA used is the *spilt* operation which partitions the input SPPO into 3 new SPPOs, whose values are equal to, greater than, and less than the selected pivot.

4.1 Evaluation Results

The experiments are undertaken using the JRTS JVM on a Sunfire V240 running Solaris 10. We measure the average, worst and best-case execution times of the algorithm executed on a sample of 10000 randomly generated lists. The experiments are undertaken with lists of varying sizes. We compare the resulting average-case times with the number of comparisons calculated from the recurrence equations obtained by the RT-MOQA analysis tool.

The quality of the experimental average-case time results is highly dependant on the distribution of the input data. For our study we used the Jakarta Commons Math [11] for the random generation of inputs. Figure 4 shows the good distribution obtained for a sample of 10000 lists of size 2048

In Figure 5 the number of comparisons multiplied by a *work coefficient* is plotted with the experimen-

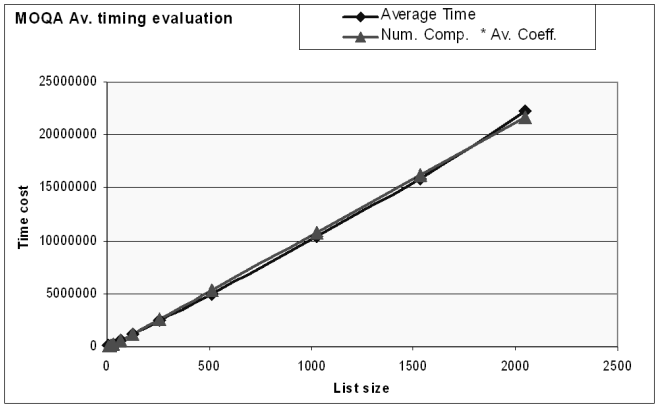

Figure 5: Experimental results vs. RT-MOQA results

tally obtained ACET. Note that we use the coefficient in our experimental evaluation to demonstrate that the number of comparisons is proportional to the actual ACET. The coefficient represents the average-case time of the code attached to each comparison. The graph shows that the results obtained by the analysis tool for RT-MOQA give good estimations of the actual ACET obtained experimentally. This can mainly be attributed to the exact ACET analysis achieved in RT-MOQA. Other factors are the good distribution of inputs and the RT-Java timing being very accurate.

5 Conclusion and Future Work

In this paper we described a new approach to incorporate automated modular average-case timing using MOQA in RT languages, specifically RT-Java. We explain how average-case timing is important in real-time systems for scheduling and budgeting of resources when used in conjunction with worst-case estimates. Our approach has been validated using typical data-structuring examples. However, RT-MOQA is continuously evolving into a more expressive and reliable RT-tool.

MOQA, as described in this paper is dedicated to data structuring programs. We will explore its extension to more classes of programs, for example low-level hardware programs which use series parallel techniques.

In the automatic analysis tool the timing will be improved to incorporate new measures which may be necessary in different situations. Also WCET analysis will be included.

With respect to the RT features of MOQA we intend to explore a garbage collection strategy for RT-MOQA, so that we can accurately time programs outside of scoped or immortal memory. We will investigate improving RT scheduling by developing a formal approach using ACET in combination with other parameters determined by the nature of the application.

The overall goal is that RT-MOQA will become a complete RT-tool including ACET and WCET analysis. With RT-MOQA a library of timed components will be developed.

References:

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

[2] 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.

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

[4] M. Schellekens, D. Hickey, G. Bollella, “ACETT, a Linearly-Compositional Programming Language for (semi-)automated Average-Case Analysis”, WIP Proceedings, 25th IEEE RTSS, Lisbon, 2004.

[5] M. P. Schellekens, Compositional Average-Case Analysis, preprint (under review with JACM), 2006.

[6] M. P. Schellekens, A Modular Calculus for the Average Cost of Data Structuring, Efficiency-Oriented Programming in MOQA, Springer book to appear, 160 pages.

[7] Raimund Kirner, Peter Puschner, “Classification of WCET Analysis Techniques”, 8th IEEE International Symposium on Object-Oriented Real-time distributed Computing, 2005 (ISORC’05).

[8] D. E. Knuth, “Deletions That Preserve Randomness”, IEEE Transactions on Software Engineering, Vol SE-3, No. 5, September 1977.

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

[10] 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.

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

[12] http://www.sable.mcgill.ca/soot/