A TOOL FOR AVERAGE AND WORST-CASE EXECUTION TIME ANALYSIS

David Hickey[1], Diarmuid Early[1] and Michel Schellekens[1]

Abstract
We have developed a new programming paradigm which, for conforming programs, allows the average-case execution time (ACET) to be obtained automatically by a static analysis. This is achieved by tracking the data structures and their distributions that will exist during all possible executions of a program. This new programming paradigm is called $\mathcal{MOQA}$ and the tool which performs the static analysis is called Distritrack. *In this paper we give an overview of both $\mathcal{MOQA}$ and Distritrack. We then discuss the possibility of extending Distritrack for static worst-case execution time (WCET) analysis of $\mathcal{MOQA}$ programs using the tight tracking of data structures already being performed.*

1. Introduction

Much work is being done on the development of ways to predict program execution times. The efforts are concentrated into two areas - *complexity theory* in which various time measures have been developed for counting the basic number of operations in a program and *real-time systems* in which constraints on the execution times of programs are imposed, e.g. deadlines, cost, etc.

In general however, the static analysis of programs to determine any property, one of which is time, is known to be very difficult in practice. Measuring ACETs automatically is no different. Some analysis techniques have been developed, e.g. [2], but these tend to be quite complicated involving many difficult mathematical techniques. Along with this, it is required that in some cases the algorithms are programmed in an unfamiliar style when compared to general programming languages.

$\mathcal{MOQA}$ involves a way to determine statically the distribution of all possible data structures at any point in a program. This makes an ACET analysis possible. The underlying mathematical techniques are less complicated than previous approaches and allow a common programming style. Currently $\mathcal{MOQA}$ programs are coded in Java using an API implementing $\mathcal{MOQA}$'s operations.

Distritrack is the tool that has been developed to automate the static ACET analysis of $\mathcal{MOQA}$ programs. It combines elements of a number of static analysis techniques in order to track the data structures and their distributions as set out in $\mathcal{MOQA}$. The output of an analysis is generally a recurrence equation representing the number of basic operations, e.g. comparisons, swaps or Java bytecode instructions, executed on the data structures. As future research, low-level timing information for specific hardware could be combined with this in order to determine the expected "real" ACET (i.e. clock cycles, milliseconds, etc.) for the program being considered taking into account caching, pipelining, etc.

ACETs can be used in conjunction with other execution time measures in soft real-time systems to

[1]Centre for Efficiency Oriented Languages, Department of Computer Science, National University of Ireland, Cork

ECRTS 2008
8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2008/1668

estimate deadlines. While deadlines determined in this way only guarantee enough time for a majority of their associated tasks, they may however lead to a significant improvement in the utilisation of system resources [7]. When deadlines are hard, WCET is a more suitable execution time measure. Like ACET, this is often difficult to obtain. Here we examine if the tight tracking of data structures that is performed by Distritrack might in fact also facilitate a WCET analysis.

This paper is organised as follows. In Section 2. $\mathcal{MOQA}$ is introduced. Then in Section 3. an overview of Distritrack is given along with details of how data structures are tracked. Section 4. gives an example of a $\mathcal{MOQA}$ program and the corresponding ACET output by Distritrack. Next in Section 5. we examine possible ways of extending Distritrack's current analysis in order to obtain WCET estimates. Finally in Section 6. some concluding remarks are given.

2. $\mathcal{MOQA}$

$\mathcal{MOQA}$ [9, 10] is a special purpose high-level language for data (re)structuring. Its data structures are simply specified as finite partial orders. Its operations are based on the classical abstract data type operations. However, each operation has been purposely designed to guarantee that data collections remain random throughout the computations. This in turn guarantees a modular ACET analysis.

In this section an overview of $\mathcal{MOQA}$ is given based on the main ideas discussed in [9].

2.1. Data Structures

The basic data structure in the current implementation of $\mathcal{MOQA}$ is a series-parallel partial order (SPPO) which is a partial order that only allows nodes to be in series, denoted by $\otimes$, or in parallel, denoted by $\|$. For example the SPPO in Figure 2(a) can be represented in series-parallel notation by $d \otimes ((b \otimes a) \| c)$. Sub-structures, which can be single nodes or more complex SPPOs, are called *components*.

The data of the language are labellings of the data structures. A data-labelling is simply an assignment of a finite number of values to each node of the data structure so that the directed links of the data structure are respected, i.e. if there is a directed link from a node x to a node y, then the label assigned to x must be less than the label assigned to y. These labels can be any value, e.g. natural or real numbers, words, other data structures containing data such as trees, etc. Any two labels need to be comparable with respect to a given order on labels. For instance, the order on natural number labels typically is the usual order on natural numbers.

$\mathcal{MOQA}$ programs compute over data-labellings, and will at each stage transform data-labellings to new data-labellings. In such computations it is important to identify the states that data-labellings can be in.

A *state* represents a collection of order-isomorphic data-labellings, i.e. data-labellings whose labels are arranged in the same relative order within data structures.

We illustrate this with the data-labellings for the tree of size 4 given in Figure 1. If we use four distinct values, say a, b, c, d, to represent the states of the same tree, where $a < b < c < d$, we have only three possible states as displayed in Figure 2. Note that the data-labelling in Figure 1(a) matches the state in Figure 2(a) and data-labelling in Figure 1(b) matches the state in Figure 2(b).

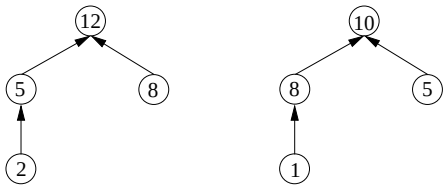

Figure 1. Data-labellings on data structures.

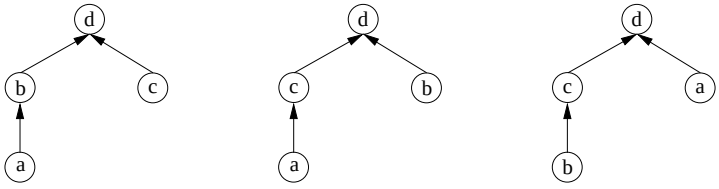

Figure 2. Data-structure states.

Essentially, states reflect the relative order that the labels can be in, on any given data structure. The values of the labels are irrelevant in this context, only their relative order is captured.

For any given data structure, the finite collection of the set of states over this data structure, is referred to as the *random structure* over the given data structure.

This amounts to the assumption that inputs for software are equally likely to occur in any of a given number of finite states. Random data can be concisely captured as above via the notion of a random structure. In practice of course, there may be several possible data structures. To represent this, the notion of random bags is introduced. A *random bag* consists of finitely many random structures, $R_1, \ldots, R_n$, each of which has a *multiplicity* K_i, where $i \in 1, \ldots, n$, which is a natural number used in the calculation of the probabilities involved in the distribution.

2.2. Operations

Operations in $\mathcal{MOQA}$ map input random bags to output random bags. Operations which correctly do this are *random bag preserving*.

The multiplicities of the input random bags are the key to the calculation of the ACETs. The ACET for an operation P with input random bag $R = \{(R_1, K_1), \ldots, (R_n, K_n)\}$ is

$$\overline{T}_P(R) = \sum_{i=1}^{n} \frac{K_i|R_i|}{|R|} \times \overline{T}_P(R_i) \tag{1}$$

where $|R_i|$ indicates the number of states in R_i, $\frac{K_i|R_i|}{|R|}$ is the probability of R_i occurring and $\overline{T}_P(R_i)$ is the ACET of executing P with input R_i.

Then, taking random bag preserving programs/operations P and Q such that executing P on random bag R results in random bag R', the ACET of executing P followed by Q is:

$$\overline{T}_{P;Q}(R) = \overline{T}_P(R) + \overline{T}_Q(R') \tag{2}$$

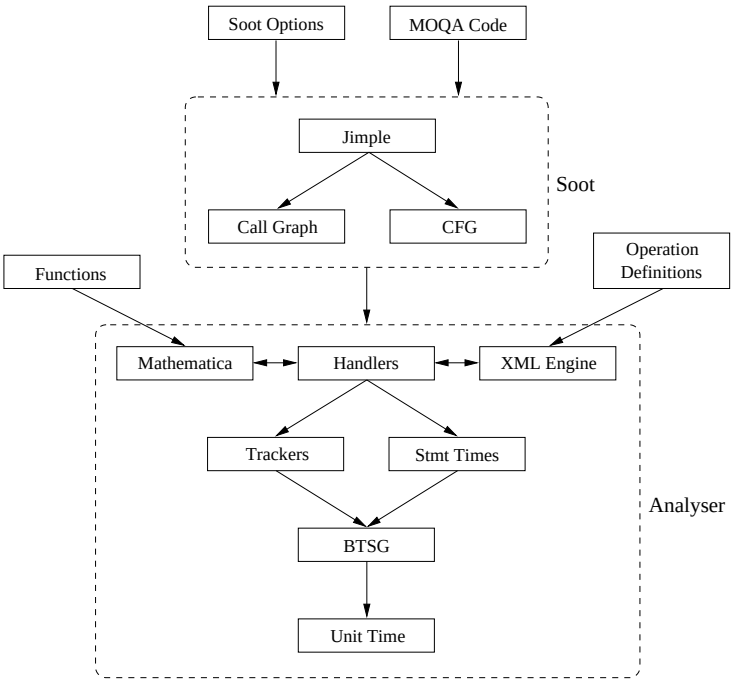

Figure 3. Distritrack architecture.

For the purpose of this paper we will focus on two main $\mathcal{MOQA}$ operations. For a complete description of the operations, designed to capture traditional data structuring operations in a randomness preserving fashion, we refer the reader to the Springer book [9].

Here we focus on the $\mathcal{MOQA}$ deletion operation $\overline{Del}$, the $\mathcal{MOQA}$ product operation $\bigotimes$ and the $\mathcal{MOQA}$ split operation.The $\mathcal{MOQA}$ product operation enables the user to "merge" two $\mathcal{MOQA}$ data structures into a new $\mathcal{MOQA}$ data structure. For the specific case of list labellings, this operation corresponds to the traditional merging of two lists. The product of a single element data structure with a larger data structure amounts to the classial insertion operation of inserting an element into a given data structure. The $\mathcal{MOQA}$ product operation uses the traditional PushUp and PushDown operations on labels as part of its internal working. The $\mathcal{MOQA}$ deletion operation enables the user to remove a label from a given labelling in such a fashion that the original data structure is reduced to a random bag of new data structures, with labellings not containing the deleted label. The $\mathcal{MOQA}$ split operations simply reorders label relative to a given label, similar to the partitioning operation of standard Quicksort. Details for these operations are provided in [9]. Most applications of the operations reduce to the specific cases outlined above and hence the application of the operations in practice are a great deal simpler than the definitions over general random structures as presented in [9]. We will indicate later on how to handle the worst-case analysis for the case of these two operations.

3. Distritrack

3.1. Overview

Figure 3 gives an overview of the design of Distritrack.

The most important aspect of Distritrack [3] is its ability to track the $\mathcal{MOQA}$ data structures. This is

the fundamental requirement in $\mathcal{MOQA}$ which allows the ACET of its operations to be calculated. To allow this, *data structure representations* were formulated. These reflect the series-parallel nature of $\mathcal{MOQA}$'s data structures and facilitate the application of the *composition laws* (cf. Chapter 6 of [9]) and the evaluation of formulae for multiplicities and the numbers of states.

At any point in a program each variable referencing a $\mathcal{MOQA}$ data structure has a *random bag tracker* associated with it. A random bag is represented as a collection of the data structure representations, in effect corresponding to random structures, which together represent all possible states of the corresponding data structures.

To achieve this, the static analysis performed by Distritrack takes each statement in the code and *handlers* simulate its effects on the actual data structures by altering the corresponding data structure representations in the random bags being tracked. This can be viewed as an abstract interpretation of the semantics of $\mathcal{MOQA}$ operations. The "abstract" semantics are encapsulated in XML and processed by Distritrack's XML engine.

In order to be able to analyse the code effectively on a statement by statement basis, Distritrack performs a flow analysis [5] of the program. This is done by the construction of a control-flow graph and call graph for the program using a tool called Soot [8]. Information on the analysis is encapsulated in another graph called a BTSG.

Distritrack gives special attention to statements such as `for` loops[2], recursive calls and `if` statements which affect the control flow. The first two complicate the tracking of the data structure representations because the effects of a statement can not generally be analysed in isolation and have to be simulated for a symbolic number of executions, e.g. n. To solve this problem the use of *recursive data structures* (RDS) [4] was incorporated.

RDSs are especially suited for recursion. In $\mathcal{MOQA}$ theory there are two templates defined for recursion based on the series-parallel nature of the data structures. For Distritrack these have been generalised to give a more standard template for recursion. It is called *series-parallel recursion* and is defined informally as follows:

$$Q(Y) = R(Y); P(Q(Y_1); \ldots; Q(Y_k))$$

where R transforms SPPO Y into $Y_1, \ldots, Y_k$ which can be in series or parallel and P processes the results of the recursive calls. Either P or R can be optionally excluded.

We also developed a set of rules for the construction of RDS definitions directly from the code associated with `for` loops. These rules are quite powerful but also require templates to be imposed on the loop bodies.

Processing `if` statements and more specifically the conditions they depend on was very challenging. However for a limited category of conditions calculating probabilities and determining the effects on data structure representations is possible to automate in Distritrack. When a probability is not possible to calculate *cases* are incorporated into the formulae for the ACETs, effectively resulting in separate ACETs for true and false branches.

Evaluating the formulae at various points in the analysis is achieved by interacting with Mathematica.

[2]Other types of loops like `while` are currently not considered.

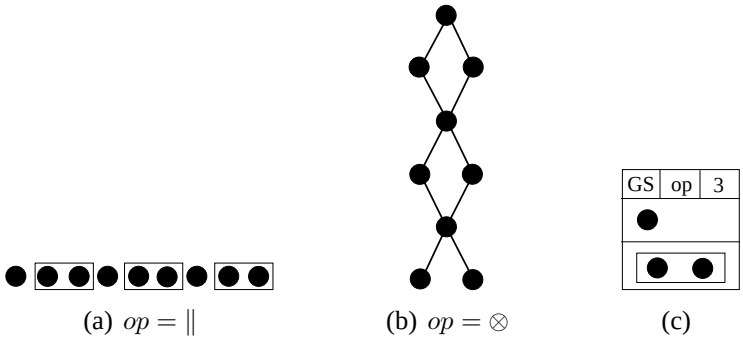

Figure 4. Group structure with a repeat value set to 3.

The final output of Distritrack are ACET formulae built in a modular way from the program statements and RDS formulae if required. These will generally be recurrence equations.

The analysis performed by Distritrack is quite flexible. The analysis itself requires little user interaction with only some guidance on the processing of RDSs being provided through code annotations. The tool can handle not only all the features of $\mathcal{MOQA}$ but also many aspects of the Java programming language. The analysis is interprocedural and can span multiple Java classes. Constructors, overloaded methods, class hierarchies and many other complex features can be handled.

Here we give an overview of the data structure representations that are incorporated into Distritrack for tracking the random bags. We will also discuss how some of the required information for calculating ACETs can be derived from these representations.

3.2. Data Structure Representations

The means by which Distritrack tracks values during its analysis of $\mathcal{MOQA}$ programs is through the use of *trackers*. The most important of these are *random bag trackers*. A random bag tracker is a set of *random structure representations* built using *fundamental SPPO (series-parallel partial order) representations*. A multiplicity is attached to each random structure representation.

Currently the fundamental SPPO representations incorporated into Distritrack are *empty structure*, *basic structure*, *primitive structure* and *group structure*. An empty structure contains nothing and a basic structure represents a single data structure node. A primitive structure contains exactly two components directly reflecting the binary nature of the series and parallel operators. Group structures are n-ary structures, i.e. they can contain an arbitrary number of components, all joined either in series or in parallel. The components within the group structure can have a *repeat value* set which determines the number of occurrences of the components defined in the group structure. Figure 4 shows an example of this. The repeat value can also be set to a symbolic value. Thus group structures form an important aspect of Distritrack's symbolic analysis.

In practice the tracking of the data structures is quite complicated and requires some more sophisticated representations, including RDSs as mentioned above.

The tracking of data structures can be compared to *shape-analysis* [11]. For example, the data structure representations can be viewed as *shape graphs* and the use of symbolic values for repeat values can be viewed as *summarization*. In both cases an abstract interpretation of the operations that modify

the shape graphs is used.

3.3. Calculating Operation ACETs

For a SPPO representation the most important values that need to be obtained are summarised as follows:

$|s|$ The size of the entire SPPO s.

$|M(s)|$ The size of the set of maximal nodes (informally defined as having no parents) in s.

$|m(s)|$ The size of the set of minimal nodes (informally defined as having no children) in s.

$L(s)$ The number of states possible on s.

Composition Laws The ACET to manipulate labels in an SPPO based on the series-parallel structure. Currently the ACET is defined as the number of comparisons required, which is suitable for the data restructuring algorithms currently implemented in $\mathcal{MOQA}$. For example the simple composition law σ_{up}, which gives the average number of comparisons to push the minimum label from a minimal node up to a maximal node, is defined as follows ($\bullet$ is a single node):

$$\begin{aligned}\sigma_{up}(\bullet) &= 0\\ \sigma_{up}(A \otimes B) &= \sigma_{up}(A) + \sigma_{up}(B) + |m(A)|\\ \sigma_{up}(A \| B) &= \frac{|A|\sigma_{up}(A) + |B|\sigma_{up}(B)}{|A| + |B|}\end{aligned}$$

Multiplicity The multiplicity of the random structure.

Many of these will be derived in a recursive way, with empty and basic structures providing the base cases. The multiplicities are determined by the definitions of operation behaviour in the abstract semantics supplied to Distritrack.

As an example lets look at primitive structures. A primitive structure can be represented as follows: $ps = s_1$ *op* s_2, where *op* is either $\otimes$ or $\|$ and s_1, s_2 are two components. The following lists the recursive way in which the size related values are obtained:

- $|ps| = |s_1| + |s_2|$
- If *op* is $\otimes$ then $|M(ps)| = |M(s_1)|$. If *op* is $\|$ then $|M(ps)| = |M(s_1)| + |M(s_2)|$.
- If *op* is $\otimes$ then $|m(ps)| = |m(s_2)|$. If *op* is $\|$ then $|m(ps)| = |m(s_1)| + |m(s_2)|$.

Functions for counting the number of states and the composition laws are binary operations based on the series parallel nature of the data structures. Therefore they can very naturally be applied to primitive structures.

Though more complicated to derive, these values can also be obtained for group structures.

With these formulae, the ACET for an operation can then be obtained using Equation 1 where the ACET on each random structure is derived using the composition laws.

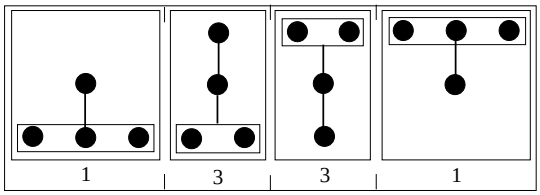

Figure 5. Output of $\mathcal{MOQA}$'s split operation on an atomic random structure of size 4.

4. Example

Listing 1 gives the code for Quicksort implemented in $\mathcal{MOQA}$. For simplicity the code for Java 5's generics is omitted. The input to method1 is considered always to be a list.

Line 9 is an annotation which tells Distritrack to build a RDS definition for the first of the method's parameters. In this case the RDS is a a simple single random structure - the sorted list. The representation for this is built into Distritrack and is called *Linear*. In general however Distritrack will generate a new RDS definition based on the code of the recursive method. All annotations to Distritrack are optional and, except for those related to generating RDSs, give information to Distritrack which may make the output simpler or the analysis more efficient.

Line 13 contains the $\mathcal{MOQA}$ operation split which partitions the input list around a random pivot. Figure 5 shows the output random bag of split for an input list with 4 nodes. The number under each random structure represents its multiplicity. In practice Distritrack tracks the output for symbolic sizes.

Lines 15 and 16 then recurse on the resulting partitions similar to the non-$\mathcal{MOQA}$ version of Quicksort code.

Listing 1. Quicksort in MOQA.

```
 1 public class QuicksortTest {
 2
 3    public OrderedCollection method(
 4          OrderedCollection oc) {
 5       quicksort(oc);
 6       return oc;
 7    }
 8
 9    @Transform(param=0, rep=RDSBuild.SR, name=``Linear'')
10    private void quicksort(OrderedCollection oc) {
11       if(oc.size() > 1) {
12          NodeInfo partitionNI = oc.getDirectNodeInfoIter().next();
13          OrderedCollection partition = oc.split(partitionNI);
14          Iterator aboveAndBelow = partition.getDirectSubsetIter();
15          quicksort(aboveAndBelow.next());
16          quicksort(aboveAndBelow.next());
17       }
18    }
19 }
```

Listing 2. Quicksort ACET Mathematica package.

```
quicksort[n1_] := Which[Greater[n1,1],Plus[-1,n1,
   Sum[Times[Power[n1,-1],quicksort[Plus[-1,n1,Times[-1,r0]]]],{r0,0,Plus[-1,n1]}],
   Sum[Times[Power[n1,-1],quicksort[r0]],{r0,0,Plus[-1,n1]}]],
   True,0];
```

```
method[n0_] := quicksort[n0];
```

Listing 2 gives the Mathematica package generated by Distritrack for the ACETs of the methods analysed in the Quicksort example. The ACET of the quicksort method corresponds to the standard Quicksort recurrence:

$$qs[n] = n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} qs[i]$$

5. Extending Distritrack for WCET Analysis

Adapting Distritrack for a WCET analysis requires new composition laws which calculate the worst-case number of basic operations executed on a random structure when a $\mathcal{MOQA}$ operation is encountered. Effectively these will select one of the states represented within the random structure which gives the largest execution time.

To illustrate how this can be achieved we briefly discuss the worst-case execution times for the two main operations discussed in the present paper: the product operation $\bigotimes$ and the deletion operation $\overline{Del}$.

5.1. Worst Case Running Times of Basic Operations

5.1.1. Delete

Let R be a random structure with an underlying partial order A. If we call $\overline{Del}(r, k)$ on each labeled SPPO r in R, the greatest number of comparisons made by any operation call is $\delta^W_{up}(A, k)$.

The δ^W_{up} function satisfies the following series-parallel recurrences (where A and B are non-empty, disjoint partial orders):

1. $\delta^W_{up}(A \otimes B, k) = \begin{cases} \delta^W_{up}(A, k) + |B_{min}| - 1 + \delta^W_{up}(B, 1) & k \leq |A| \\ \delta^W_{up}(B, k - |A|) & k > |A| \end{cases}$
2. $\delta^W_{up}(A \| B, k) = \max\left(\delta^W_{up}(A, \min(k, |A|), \delta^W_{up}(B, \min(k, |B|)\right)$
3. $\delta^W_{up}(\bullet, k) = 0$

5.1.2. Product

Let R be a random structure with an underlying partial order A. If we replace the smallest label on each labeled SPPO in R with a label which is larger than k members of the label set and smaller than the others, and then call PushUp on the node with that label which simply pushes up the label, the greatest number of comparisons made by any operation call is $\pi^W_{up}(A, k)$. We define $\pi^W_{down}(A, k)$ in a similar manner by replacing the *largest* label and calling Push*Down* to push down a label.

If we similarly replace the smallest label on each labeled SPPO in R with a label greater than k members of the label set and call a PushUp, the new label may be the label of a maximal node in the

output labeled SPPO. If this happens, then the greatest number of comparisons made by any PushUp operation in these cases is $\mu^W_{up}(A,k)$. If not, then $\mu^W_{up}(A,k) = -\infty$. We define $\mu^W_{down}(A,k)$ in a similar manner by replacing the *largest* label and calling Push*Down*.

Let $T^W_P[I_1, I_2]$ be the worst-case running time for the unary product on the components I_1 and I_2 over all labellings in the random structure R with underlying structure $I_1 \| I_2$. Then we have

$$\begin{aligned} T^W_P[I_1,I_2] \leq \; & (\min(|I_1|,|I_2|+1))(|I_{1,max}|+|I_{2,min}|-1)+ \\ & + \sum_{i=1}^{\min(|I_1|,|I_2|)} \left[\pi^W_{down}(A,i) + \pi^W_{up}(B,|B|+1-i)\right] \end{aligned}$$

The π^W_{up} and μ^W_{up} functions satisfy the following series-parallel recurrences (where A and B are non-empty, disjoint partial orders):

1. $\pi^W_{up}(A \otimes B, i) = \begin{cases} \max\left(\pi^W_{up}(A,i), \mu^W_{up}(A,i) + |B_{min}|\right) & i \leq |A| \\ \pi^W_{up}(A,|A|) + |B_{min}| + \pi^W_{up}(B, i-|A|) & i > |A| \end{cases}$
2. $\pi^W_{up}(A \| B, i) = \max\left(\pi^W_{up}(A, \min(i,|A|)), \pi^W_{up}(B, \min(i,|B|))\right)$
3. $\mu^W_{up}(A \otimes B, i) = \begin{cases} -\infty & i \leq |A| \\ \mu^W_{up}(A,|A|) + |B_{min}| + \mu^W_{up}(B, i-|A|) & i > |A| \end{cases}$
4. $\mu^W_{up}(A \| B, i) = \max\left(\mu^W_{up}(A, \min(i,|A|)), \mu^W_{up}(B, \min(i,|B|))\right)$
5. $\pi^W_{up}(\bullet) = \mu^W_{up}(\bullet) = 0$

5.2. Extending Distritrack

The advantage of extending Distritrack for WCET analysis is that the input-output trace that it currently undertakes leads to very accurate WCETs. With the new composition laws Distritrack can compute the WCET for each random structure representation in a random bag being tracked as input into an operation. This can be done without altering the way in which the random bag trackers are generated. When an entire method/program is analysed, the information obtained for each operation is combined and the sequence of operation WCETs for the overall WCET can be derived. Existing WCET tools already incorporate techniques for finding the maximum time required for different execution paths in a program, for example [1, 6]. These techniques could be applied in a similar fashion to determine the WCET from the times associated with the random structures.

As an alternative, Distritrack could maintain the WCET to build a random structure up to each point in the program analysis. Say operation op_i is being analysed and its input is the random bag R. Let $WCET_{i-1}(R_j)$ be the WCET required to build random structure R_j within R by the $i-1$ operations before op_i and $T^W_i(R_j)$ be the WCET of executing op_i on R_j. Each random structure in the output random bag resulting from the execution of op_i on R_j will be associated with the WCET $WCET_{i-1}(R_j) + T^W_i(R_j)$.

Then, after the last operation in a path of execution in a program, the WCET of that path will be the maximum WCET value from the random bag output from the operation.

Other than this, the static analysis currently performed by Distritrack can remain largely unchanged.

This however does not use all the information provided by the data structure representations built by Distritrack. The multiplicities may sometimes be useful in obtaining time estimates for the inputs to an operation which are "most likely" to occur. In [7] this is shown to be important when, using the WCET alone to determine deadlines in a real-time system, there is a large waste of resources when the input that causes it occurs very infrequently. Therefore Distritrack can, for example, drop a WCET value if the WCET occurs in a random structure that has a probability of occurring less than $\frac{1}{20}$. This of course is only relevant for soft real-time systems.

However multiplicities in this case only lead to a partial solution. While the states within a random structure have a uniform distribution, we currently do not have information on the distribution of the execution times over the states. It may be possible however to develop ways of extracting such information, again similar to the way the current composition laws derive ACETs.

Multiplicities may also indicate which random structures contain the worst case state for an operation. It has been observed that the WCET generally occurs in a random structure which contains the largest *atomic* (single nodes in parallel) components. In terms of *divide and conquer*, this makes sense. As it turns out, the output from operations that create atomic components in series appear to always have the lowest multiplicity attached to the random structure containing the largest atomic components. This may be because these random structures contain more states and therefore fewer copies occur. An example of this can be found in the output of $\mathcal{MOQA}$'s split operation. For size 4, in Figure 5 it can be clearly seen that the random structures containing the largest atomic components have the lowest multiplicity. These also form the worst case input into the recursive call of quick sort.

6. Conclusion

In this paper we have given an overview of a new programming paradigm called $\mathcal{MOQA}$ and a corresponding tool called Distritrack. Distritrack performs a static analysis of $\mathcal{MOQA}$ programs, tracking the data structures and their distributions as they are input to and output from program statements in order to derive the ACET.

We discussed how Distritrack can have its ACET static analysis extended to derive WCET and other time measures using the information provided by the tracking of the data structures. We have provided some initial ideas to serve as a basis on which to investigate this further, supporting a future implementation of the tool in which the estimates provided for improved resource budgeting in soft real-time applications can be supplemented with accurate WCET deadlines for hard real-time applications.

The price of the accurate results obtained by Distritrack are some limitations on the static analysis which are required to obtain the tight tracking of the data structures, e.g. the analysis must be context-sensitive in that all program paths have to be analysed separately.

References

[1] R. Chapman, A. Wellings, and A. Burns. Integrated program proof and worst-case timing analysis of spark ada. In *Proceedings of the Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[2] Philippe Flajolet, Bruno Salvy, and Paul Zimmermann. Automatic average-case analysis of algorithms. *Theor. Comput. Sci.*, 79(1):37–109, 1991.

[3] David Hickey. Distritrack: Automated average-case analysis. In *QEST '07: Proceedings of the Fourth International Conference on Quantitative Evaluation of Systems*, pages 213–214, Washington, DC, USA, 2007. IEEE Computer Society.

[4] C. A. R. Hoare. Recursive data structures. *International Journal of Parallel Programming*, 4(2):105–132, 1975.

[5] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[6] Peter Puschner. Worst-case execution time analysis at low cost. *Control Engineering Practice*, 6:129–135, Jan. 1998.

[7] Peter Puschner and Alan Burns. Time-constrained sorting – a comparison of different sorting algorithms. In *Proc. 11th Euromicro International Conference on Real-Time Systems*, pages 78–85, Jun. 1999.

[8] McGill University Sable. Soot, a java optimization framework. www.sable.mcgill.ca/soot.

[9] M. P. Schellekens. *A Modular Calculus for the Average Cost of Data Structuring*. Springer, August 2008. http://www.springer.com/computer/foundations/book/978-0-387-73383-8.

[10] M. P. Schellekens *$\mathcal{MOQA}$; unlocking the potential of compositional static average-case analysis.* In *Journal of Logic and Algebraic Programming*, accepted for publication, to appear.

[11] Reinhard Wilhelm, Shmuel Sagiv, and Thomas W. Reps. Shape analysis. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 1–17, London, UK, 2000. Springer-Verlag.