Distritrack: Automated Average-Case Analysis

David Hickey
Centre for Efficiency-Oriented Languages
Department of Computer Science
National University of Ireland, Cork
email: d.hickey@cs.ucc.ie

Abstract

MOQA is a language specifically designed to facilitate average-case execution time analysis. It incorporates some innovative techniques to make this possible. Following on from this, a tool called Distritrack has been built to perform the analysis of MOQA programs, adhering strictly to the language specification.

1 Introduction

There are numerous reasons why ACET is important. In real-time systems budgeting on worst-case times (WCET) can lead to a large waste of resources. In such cases additional information like ACET complementing the WCET can allow more flexible cost estimates. This is especially useful for soft real-time. Knowledge of the ACET can also aid in power analysis for embedded systems.

One of the main properties which simplifies modular timing analysis of programs is IO-compositionality [4], which MOQA incorporates. As a result the time of a MOQA program can be expressed as a summation of the time of its basic components, where the time is interpreted in the standard way as the number of comparisons executed. To time the basic components MOQA uses an innovative technique to track all the possible states the data structures can be in and their distributions.

Following on from the development of the language, a tool called Distritrack has now been developed to perform the analysis of MOQA programs. In this paper this tool is explained.

2 Distritrack

To aid the explanation of Distritrack the simple example of MOQA code shown in Fig. 1 is used. The version of MOQA considered is implemented in Java 5.0. For clarity the code for generics is omitted.

```
1: public void method1 (LPO lpo) {
2:     OrdCol oc = method2 (lpo , 2);
3: }
4: private OrdCol method2 (LPO lpo , int k) {
5:     Node pivot = lpo.getNodeIter() .next();
6:     lpo.split (pivot);
7:     OrdCol top = lpo.getSubsetIter() .next();
8:     OrdCol result;
9:     if (top.size () == k)
10:        for (int i = 0; i < k; i++)
11:           result = top.product (result ,
12:                     top.getNodeIter() .next());
13:      else
14:         result = top;
15:      return result;
16: }
```

Figure 1. MOQA code example.

If the user chooses to begin the analysis at method1, Distritrack assumes the input lpo is an atomic order, which is MOQA terminology for an unsorted list, and assigns it a size, say n.

2.1 Architecture

Fig. 2 gives an overview of the design of Distritrack. The MOQA code is pre-processed using a Java optimisation tool called Soot [3]. This transforms the Java code into an intermediate representation called Jimple whose features facilitate an analysis such as that performed by Distritrack. Soot also builds a call graph for the application and control flow graphs for each method. Fig. 3 shows a simplified version of the control flow graph (CFG) for method2. The numbers refer to the line numbers in Fig. 1.

The analyser traverses the call graph and CFGs to derive the ACET of the code. Its main processing engine incorporates Handlers which deal with the different Jimple statements and expressions. Each handler contains a method handle which takes the Jimple code of a CFG node as input. Trackers track variable values, e.g. MOQA's data struc-
Figure 2. Distritrack architecture.

Figures, and other important aspects of the program, e.g. the number of times a loop will iterate. The handle method either creates new trackers or transforms a set of existing trackers according to the behaviour of the Jimple code.

When the handler for invoke expressions identifies a MOQA operation the XML engine obtains a definition of the operation’s behaviour. The definitions map the inputs to the outputs and attach the information required for the analysis, which includes the operation’s time function. Currently Distritrack incorporates definitions for all the MOQA operations. However it is likely that other operations will be developed. This is why XML is used rather than hard coding the behaviour of the operations into Distritrack’s Java code.

As the handlers manipulate trackers, they also interact with the Mathematica kernel [2] through J/Link [1] where necessary. As mentioned in the description of the XML definition of operations, functions are specified. These are found in a Mathematica package along with all the functions used in calculating the times over the data structures.

Behind the analysis a Branched Time State Graph (BTSG) is built for each method from its CFG. Each path in a BTSG corresponds to one branch of execution in a method and must have its own set of trackers. A branch splits into two branches when a branching statement such as an if statement is encountered. Attached to each statement is the set of branches which it affects. When a statement is being handled, the trackers which it modifies must be updated in each one of those branches. For example take line 15 of the code in Fig. 1. The return statement has two predecessors as can be seen in the CFG in Fig. 3. The statement therefore has the union of its predecessor’s branches associated with it. A graphical representation of the BTSG of the example code is shown in Fig. 4.

Inevitably Distritrack is affected by the state explosion that is a common problem in the static analysis of branching code. To offset this somewhat, Distritrack efficiently manages how trackers for the different branches are stored in memory. However it is still possible that very large programs may become untraceable.

Other information that needs to be tracked with each branch includes the branch condition and the sum of the ACETs of the branch’s statements reflecting the compositionality of MOQA programs. Also, if necessary, the probability of the condition is tracked. For certain conditions on the data structures the probability can be calculated automatically. For other conditions user information is required.

Along with the control statements shown above Distritrack can also handle recursive methods.

3 Conclusion

Distritrack is still evolving as MOQA develops further. More operations and new data structures will be incorporated. Distritrack also needs to take more fine-tuned measurements counting all the different statements.

References