Bounding The Execution Time of Method Calls
Combining Static Analysis and Dynamic Class Loading

Patrik Persson
Department of Computer Science, Lund University
Patrik.Persson@cs.lth.se

Extended Abstract, February 2000

Background
Object-oriented languages are becoming more and more used in real-time software development, but simpler languages such as C or assembly code are more common. One reason is that the timing properties of object-oriented programs are relatively complex to predict. Nevertheless, as increasingly complex embedded systems become ubiquitous in our society, modern languages will be needed to develop their software. Object-orientation is indeed already popular for analysis and design of embedded real-time systems [1]; now we need the language support for it.

More specifically, we address worst-case execution time (WCET) analysis for object-oriented languages. Such analyses are necessary for safe scheduling in real-time systems. We base our approach on the Java language, which has received much attention in the community of embedded real-time systems. Java is compiled to bytecode, a stack-based code model for a Java virtual machine (JVM). The JVM is almost always a piece of software which can be implemented on more or less any hardware platform.

Java’s bytecode approach was originally taken to support code mobility over the Internet (applets), but has some interesting applications for embedded systems. Java is designed to support dynamic class loading, that is, to load code while the system is running. Such dynamic loading is desirable in some embedded systems, such as industrial robots and telephone exchanges, which often cannot be taken off-line without significant economic penalties.

We want to use Java for embedded real-time systems. While we want to perform static analysis to ensure predictable scheduling, we also want to retain the dynamic class loading. As we will show, these demands at first appear contradictory, but can be combined in a manner consistent with software engineering practice.
Timing Issues in Object-Oriented Languages

Although object-oriented languages (such as Java) share much of their semantics with procedure-oriented languages such as C or Pascal, a few key properties of these languages are of special importance in WCET analysis:

**Garbage collection.** A traditional garbage collector imposes unpredictable delays in the garbage-collected process, which is clearly intolerable in a hard real-time system. A number of real-time garbage collection techniques have been developed to remedy this problem. These techniques require information about the memory usage of the real-time process in question, something we have treated in our previous work [3].

**Virtual method calls.** Unlike ordinary function or procedure calls, the code to execute upon a virtual method call is not statically known: it is determined at run-time. Static WCET prediction of such a call requires special techniques.

We will now elaborate on WCET prediction of virtual method calls.

**WCET Analysis of Method Calls**

An automatic WCET analysis of method calls is actually possible in some cases. If information about all possibly called implementations of a particular methods are available for analysis, one safe WCET estimation is the longest WCET of these implementations.

This approach assumes that any of the existing implementations may be called from a given call site. Such an approximation may be improved by using existing type analysis techniques (e.g., [4]) developed for optimizing compilers. A similar approach is taken in [2].

However, a global analysis is not always possible. In particular, Java supports dynamic class loading, which means that new implementations of a given virtual method may be introduced at run-time. To handle such an environment, another approach is required.

**Timing Constraints as Method Signature Information**

We will now present an approach to handling timing requirements on virtual methods in the context of dynamic class loading. We base this approach on two observations:

- The WCET of a method call should be known to the programmer implementing that call, even if the implementation is not yet available. If it is not, the programmer has no way of fulfilling these requirements.
• When a new class is loaded, it must not break the timing assumptions of the existing system. A new implementation of a virtual method should not cause any code calling that method to miss its deadline.

These requirements are analogous to those for the type system in any statically-typed programming language. For example, assume that the function \( f \) accepts a single integer argument. This information is used both for semantic analysis of calls to \( f \) (to ensure that a call passes an integer argument) and of \( f \) itself (to ensure that it accepts an integer argument).

Timing constraints, such as bounds on WCETs of virtual methods, should be treated in a manner similar to such type information. We thus argue that timing constraints on virtual methods should be expressed in the method’s signature, along with the types of the parameters and the return value.

We express this information as annotations in the form of “tagged comments”. Such comments can be parsed and understood by a WCET analysis tool, yet ignored by a traditional compiler.

Example: An Embedded Controller

To give concrete form to the discussion above, we outline a small example of an embedded controller. The framework allows a method \( \text{display} \) in the operator interface to be called (to display, e.g., the control parameters) upon each execution of the controller. The controller should be possible to use with a variety of operator interfaces, but we want to bound the WCET of \( \text{display} \) to maintain predictability of the controller.

Figure 1 is an outline of how such a design can be expressed in our signature-based approach.

Conclusion

We have presented an approach to managing bounds on WCETs of virtual method calls in the context of dynamic class loading, something that must be handled if object-oriented languages are to be used properly for hard real-time systems. The approach gains from using a specialized class loading mechanism which can check and handle these timing requirements. Such mechanisms are well catered for in the current Java platform (in the form of specialized class loaders).

Our specification-oriented approach is consistent with established software engineering principles, considering the similarity to static typing in programming languages.
class PIDController extends RealTimeThread {
    private double uc, y, u, v;
    private GUI myGUI = null;
    ...
    public synchronized void setGUI(GUI g) { myGUI = g; }
    public synchronized GUI getGUI() { return myGUI; }
    public void run() {
        long t = currentTime();
        for(;;) {
            y = IO.getY();
            uc = IO.getUc();
            calc_output();
            IO.setU(u);
            update_states();
            GUI g = getGUI();
            if (g != null) g.display(uc, y, u); // (A)
            t += 100;
            waitUntil(t);
        }
    }
}

class GUI {
    ...
    abstract public void display(double uc, double y,
                                  double u)
        /**< time-bound 25ms */;
    // (B)
}

class SomeGUI extends GUI {
    public void display(double uc, double y,
                         double u) {
        // Real-time stuff goes here.
        // (C)
    }
}

Figure 1: Expressing WCET bounds for the operator interface in an embedded controller. The call at (A) has a bounded WCET, since the top-level declaration of the called method at (B) has a WCET bound associated with it. The implementation at (C) must adhere to this bound.
References


