THE WORST CASE EXECUTION TIME - OVERVIEW OF METHODS AND SURVEY OF TOOLS

1. Problem statement

Timing analysis attempts to determine bounds on the execution time of a task when executed on a particular hardware. This process is a necessary step in the development and validation process for hard real-time systems. The challenge of the problem is due to the fact that the execution time of an individual instruction is context-dependent. A deeper look to the tasks will reveal that they spend most of their execution time in loops and in recursive functions, which makes determining the loop bounds and recursion depth a crucial step to get bounds on the execution time. There are two different classes of methods to calculate the execution time.

Static methods: It relies on taking the task code itself, analyzing the set of possible Control-Flow Graph CFG paths, combining CFG with some (abstract) models of the hardware architecture, and finally obtaining upper bounds for this combination.

Measurement-based methods: They execute the task (or task parts) on the given hardware or a simulator for some set of inputs.

2. Methods in depth

2.1. **Static Methods.** Analyzing the source code is always the easiest way to avoid the complexity of working with machine code. The problem with this approach is the difficulty of mapping the results to the machine-code program because compilation, in particular code optimization and linking may change the control-flow structure. Among this class of methods: value analysis, control-flow analysis, processor-behavior analysis, estimate calculation and symbolic simulation.
Static methods use abstraction to cover all possible context dependencies in the processor behavior. The price static-methods pay for this *safety* is the necessity for processor-specific models of processor behavior, and possibly imprecise results such as overestimated WCET bounds. In favor of static methods is the fact that the analysis can be done without running the program to be analyzed - which often needs complex equipment to simulate the hardware and peripherals of the target system.

Date: Feb 18th 2011.

2.2. **Measurement-based methods.** The end-to-end measurements of a subset of all possible executions produce estimates or distributions, that are not actual bounds for the execution times. Each subset of input has its own relevant context. The context-subset problem could be attacked by running more tests to measure more contexts or by setting up a worst-case initial state at the start of each measured code segment. Unless all possible execution paths are measured or the processor is simple enough to let each measurement be started in a worst-case initial state, some context-dependent execution-time changes may be missed and the estimate produced is unsafe. The advantage claimed for these methods is their simplicity to be applied to new target processors, because they do not need to model processor behavior. Another advantage is their ability to produce more precise than bounds we get from static methods.

2.3. **Research.** For the *static methods*, users can improve the precision (tighten the bounds) by annotations that exclude infeasible executions from the analysis. For the *measurement-based methods*, users can add test cases to include more possible executions in the measurements. Some measurement-based tools also allow annotations for the estimate calculation to exclude infeasible executions or to include more executions by defining larger loop bounds than have been observed. Separation of contexts to improve the precision of the analysis.

3. Tools

Each tool has its own limitation and set of characteristics that is determined by the analyzes the tool can apply to the code, and its ability to deal with different architecture components (cache, pipelines and periphery). There are several features whose use will easily ruin precision. For example, pointers to data and to functions that cannot statically be resolved, and the use of dynamically allocated data. Most tools will expect that function calling conventions are observed. Some tools forbid recursion. Currently, only mono-processor targets are supported. Most tools only consider uninterrupted execution of tasks.

4. Evaluation

The analysis and tools presented in the literature are not integrated with scheduling. In a multithreaded application, context-switching will be a crucial factor in the behavior of the application, especially that the preempted thread will face cache-miss when it resumes execution. In such multithreaded environment, static methods will not be practical. On the other hand, the measured-based methods can give better estimate, especially that, the overhead of the context-switching will be evaluated during the end-to-end measurement. Furthermore, I think that we can enhance the measure-based method by building a *deterministic environment*, which is a system that keeps recording several application's runs. The determinism will give detailed description of the program execution, an exact details about the platform behavior of the environment in real world, and finally a complete simulation of the scheduling integration. Based on the logs generated from that system, we can enumerate all the possible scenarios and get an exact value for execution time of each possible scenario. It is also possible that this will help in identifying bugs and bottle necks during runtime.