CS 590: Software for Embedded Systems
Report on *The Worst-Case Execution Time Problem – Overview of Methods and Survey of Tools*

Summary

Determining an upper bound on embedded software execution time is an important task. It is necessary, not only to ensure a safe and functional system, but in many cases for certification by agencies. With all of the great modern processor improvements (eg., caches, pipelining), we tend to achieve better performance in almost every case, but we sacrifice predictability of execution times. This loss of predictability can lead to gross overestimation of execution times. Consider making a naïve assumption that every operand must be fetched from SDRAM, consuming 60 cycles (p. 39). Under this assumption, any instructions which find their operands cached will execute in less than 2% of the "maximum" time!

This example illustrates the need for more complex models which take more of the processor features into account. Although it is intractable to enumerate all possible execution paths in a program of any degree of complexity (even if we know they terminate), incremental improvements in static analysis and processor modeling allow certain invariants to be determined for certain sets of contexts. In the example above, if we know that the address of the operand in question was definitely written to two instructions prior, we can make the assumption that its value will be in cache (save for some pathological cache).

Other advances in static analysis techniques have improved the ability to model control flow and limit the set of possible paths. For instance, many of the tools surveyed in this paper support automatic loop bounds detection to varying degrees, as well as infeasible path detection. For cases where such helpful knowledge is not able to be determined by the analysis tool, most provide the ability to add annotations to the source or machine code in order to provide similar knowledge.

Once we have an idea of possible execution paths, we need to estimate the worst-case execution times of the basic blocks which are along those paths. There are two radically different approaches. In order to achieve safe (overly high) estimates, we must use a tool which employs *static* methods, usually symbolically executing the code on an abstraction of the CPU in question. On the other hand, in order to achieve under-estimated (but presumably closer to realistic) estimates, we may employ a *dynamic* measurement-based tool. These tools depend on running the machine code with definite inputs and values (instead of abstract) on either a simulation of the CPU or the real thing. Although the tool may try many combinations of input, it is usually impossible to guarantee that we have measured the *worst possible* execution time.

Evaluation

Can this technology be used practically? It seems that the term "practical" is very relative and subjective when it comes to embedded systems programming. Lee and Seshia (p. xiii) refer to current technology as "today's (rather primitive) means of accomplishing […] joint dynamics."

As mentioned in the paper (p. 38), WCET analysis tools are “routinely” used in some industries, so there must be some usefulness to them. Personally, if I were told by my supervisor that my next task was to perform WCET analysis on a section of code, my stomach would sink for a moment, given what this paper expresses about the relative youth of analysis tools and their precision. Here what I see as the main limitations of the state of the art, and how they may be addressed:

- CPUs appear to be “grey boxes” to tool designers, and whether producing abstract models or cycle-accurate simulations of CPUs, designers could get details wrong. Remedy: More interaction on the part of CPU vendors. Having better support for WCET tools may have the added benefit of increasing the market share of both companies.
- WCET tools cannot analyze certain constructs (eg., general function pointers), and may give unnecessarily loose bounds when analyzing others. Remedy: Programmers may restrict their use of certain features in order to make the code more amenable to analysis, and at the same time, analysis tools should add support for more annotations. Ultimately, if your code is too complex for a computer to analyze, either it cannot be analyzed by humans, or the humans should give the computer hints.
- Speed of WCET analysis is far from ideal in many cases (p.24, limitations section, for example). Remedy: There is no easy solution to this one, and it boils down to the general solution to “my computer is too slow!” — improved algorithms and improved hardware.

This is a highly important research and development area, as trust in the performance of embedded systems is required in order for them to gain higher prevalence in our world. The better these tools are, the more easily we will be able to trust systems, and the more quickly we will adopt amazing new technology.