Timing Analysis and Timing Predictability Extended Abstract

Reinhard Wilhelm*

Informatik, Universität des Saarlandes, Saarbrücken

Abstract. Hard real-time systems need methods to determine upper bounds for their execution times, usually called worst-case execution times. This paper explains the principles of our Timing-Analysis methods, which use Abstract Interpretation to predict the system's behavior on the underlying processor's components and use Integer Linear Programming to determine a worst-case path through the program. Under the assumption that non-trivial systems are subject of the analyses, exhaustive analyses can not be performed and some uncertainty about the system's behavior remains. Uncertainty, i.e., lack of information about a system's execution states incurs cost in terms of precision of the upper and lower bounds on the execution times. Some cost figures are given for missing information of different types. These are measured in machine clock cycles. It is (intuitively) argued, that component-based software design and the use of middleware may induce intolerable costs in terms of precision.

1 Execution-Time Variability

Hard real-time systems need methods to determine upper bounds for their execution times, usually called worst-case execution times, WCET. Based on these bounds, a schedulability analysis must check whether the underlying hardware is fast enough to execute the system's task such that they all finish before their deadlines. This problem is nontrivial because performance-enhancing architectural features such as caches, pipelines, and all kinds of speculation destroy the traditional compositional methods to determine bounds on execution time. These used so-called Timing Schemata [Sha89] for the statements of the programming language describing how to use the bounds for the constituents of the statements to compose bounds for the statement.

For example, upper bounds for a conditional statement would be computed according to the rule:

$$u\text{-}bound(\textbf{if } c \textbf{ then } s_1 \textbf{ else } s_2) = u\text{-}bound(c) + max\{u\text{-}bound(s_1), u\text{-}bound(s_2)\}$$

Execution times for individual instructions were assumed to be constant and available from a table in the manual of the processor.

* Work reported herein is supported by the Transregional Collaborative Research Center AVACS of the Deutsche Forschungsgemeinschaft and by the European Network of Excellence ARTIST2.

F.S. de Boer et al. (Eds.): FMCO 2004, LNCS 3657, pp. 317–323, 2005.

The above mentioned architectural features introduce "local non-determinism" into the processor behavior; local inspection of the program can not determine the contribution of an instruction to the program's overall execution time. The execution history determines this contribution. It depends on whether in the actual state the instruction's memory accesses hit or miss the cache, whether the pipeline units needed by the instruction are occupied or not, and whether branch prediction succeeds or fails. The variability of an instruction's execution time currently is roughly two orders of magnitude, with an increasing tendency.

This variability of execution times exists on all levels of granularity [TW04], not only for the individual memory access or the single instruction, but also for a context switch, for a function call, for a task or a distributed system of tasks, the communication of a message over a channel, and the delivery of a requested service on top of some middleware.

Lack of information about a system's execution time results from uncertainty of the system environment (input data, timing of input events) and from the interference on shared resources. For example, the variation in execution times for individual instructions results from the competition between different memory accesses and instructions for the caches and for functional units. The variation of communication times is caused by competition for communication channels with restricted bandwidth. This variance is at the heart of non-predictability, since the safe strategy to deal with uncertainty is to assume worst cases and thus overestimate real execution times. Overall, the remedy is over-provisioning.

1.1 Timing Analysis

Methods have been developed and tools based on them have been implemented that bound the variance of instructions' execution times [FHL+01, Wil05]. State-

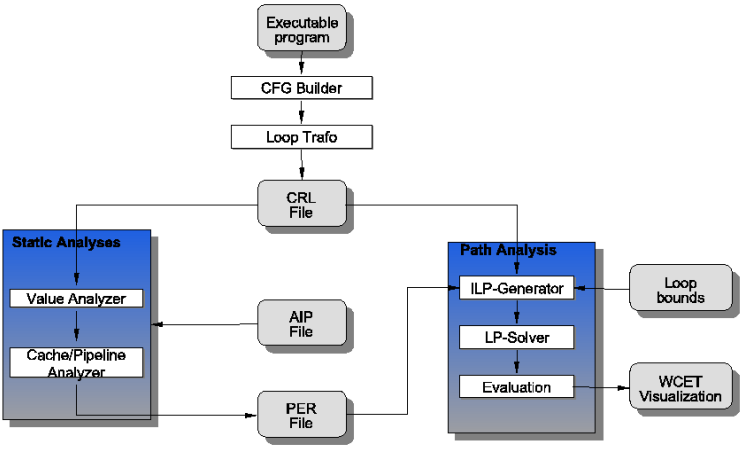

Fig. 1. Architecture of a Timing Analysis tool

of-the-art Timing-Analysis methods split the task into a sequence of subtasks, starting with a number of static analyses, which are based on the theory of Abstract Intepretation [CC77]. The first attempt to determine properties of each task's control flow and the effective addresses of its memory accesses. They use a variant of interval analysis [CH78] to determine the contents in processor registers and the values of variables. These analyses are followed by another one attempting to predict each task's behavior on the processor components such as caches and pipelines. Result of this phase are upper bounds on the execution times of basic blocks. The control flow of each task is translated into an integer linear program with the execution time of the program over all paths as objective function. Maximizing this objective function determines a worst-case path [LMW99, TFW00]. A more or less standard tool architecture has evolved shown in Figure 1.

2 Cost of Uncertainty

Software systems of realistic sizes to be run on powerful processor architectures exhibit state spaces that are too big to be exhaustively analyzed. That's why the above listed sequence of static program analyses use abstraction to reduce this space. The analyses compute at each program point invariants in the form of over-approximations of the set of execution states that can be reached when program execution reaches this point. In general, several invariants are computed for a program point, one for each *context*, i.e., control flow path by which this point can be reached. Differentiating by contexts is absolutely mandatory to obtain enough precision. Each invariant expresses the computed information about the processor's components, e.g. contents of caches, occupancy of pipeline units, state of the branch predictor etc. This information is then used to exclude the possibility of cache misses, pipeline stalls etc. and thereby safely reducing the assumptions about the execution times of instructions.

The information contained in the invariants is by necessity incomplete. First, information about the program's interference with the environment is not available, which may influence the program's execution. This requires that information on several possible control-flow paths be merged. Second, the analyses are based on an abstract model of the processor. This must be conservative with respect to the timing behavior of the concrete processor, but may abstract from details. Third, language features and software design methods can build unsurmountable barriers for even the most powerful analyses as we will attempt to show next.

Figure 2 shows the basic notions we are dealing with. Firstly, the program exhibits a variability in execution time depending on input, thus may have a range of execution times between a best-case and a worst case execution time. However, these two extreme cases can in general not be determined. With the methods described above, one can determine safe lower and upper bounds. Any worst-case guarantee can only be an upper bound. Useful bounds are not too far away from the best-case and worst-case times, resp. A system exhibiting predictable behavior will allow the analyses to arrive at precise bounds.

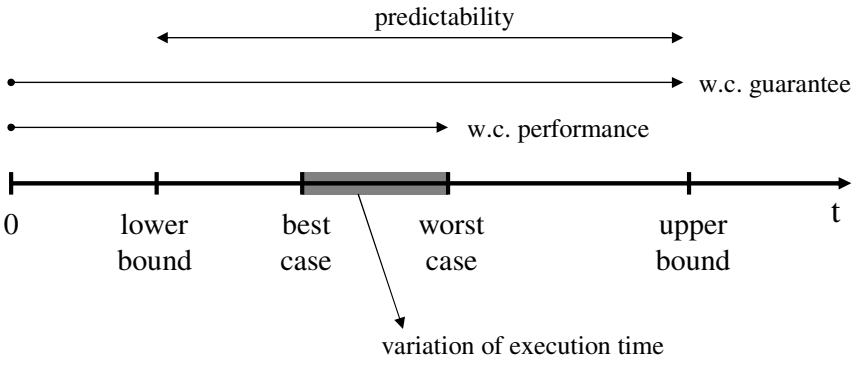

Fig. 2. Basic terms

We will now consider several types of missing information and their costs in terms of precision. Costs are measured in machine cycles. The figures given are taken from a currently popular architecture, a Motorola PowerPC processor equipped with a realistic memory system. A cache analysis will have computed safe, but approximate information about the cache contents at each program point. Missing information about whether an accessed memory block is in the cache has to be accounted for by the cache miss penalty, which is roughly 40 cycles. Depending on the write-back strategy of the cache, we may need to also account for a write back, which means adding another 40 cycles.

Furthermore, assume that the clever designer decided to use virtual memory. This comes with a translation-lookaside buffer (TLB). A TLB miss requires 12 reads and 1 write. Assuming that one can not safely exclude that these reads and writes miss the cache. This means that a TLB miss that can not safely be excluded has to be accounted for with at least $13 \times 40 = 520$ cycles. A page fault costs around 2000 cycles.

Assume, that an object-oriented language has been used for the implementation of the system's tasks. Dynamic method invocation uses a data structure to identify the actual method to activate. An efficient implementation spending some memory overhead for virtual-function tables [WM97] still needs two indirect references to activate the correct method. If we can not exclude the possibility that these two table lookups cause page faults, a method invocation costs 2×2000 cycles in the worst-case.

Now come the *second-order effects*. Let us consider a pointer to data whose value can not be statically inferred. When analyzing an access through this pointer, the analysis must assume accesses to all sets of the cache. With an access to a known address, the cache analysis removes some information, namely the memory block that may be replaced, but it also gains some information, namely the memory block that was loaded into the cache. With unknown access, the analysis only looses information; it must reflect, that one memory block may be removed from each set of the cache, but does not know where the memory block moves into the cache. This is a second-order effect, because it ruins the

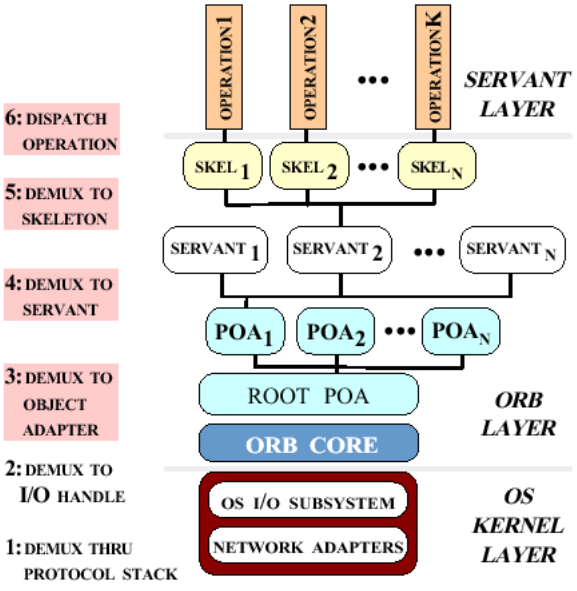

Fig. 3. The ZEN open-source RT-CORBA middleware. Picture taken from [KKSC03]

information about cache contents for future accesses. Even worse are statically unresolvable function pointers. Their damaging effect is of even higher order, because for each indirect call through such a function pointer the worst-case damage for all the potentially called functions has to be assumed.

Middleware has its motivation in the potential for reuse of software components. Object request brokers have to be tailored for the use in real-time systems. In CORBA, requests for services may be served remotely over the net. No time guarantees can be given in this case. A real-time version of CORBA called RT CORBA has been developed. Attempts have been undertaken to increase its predictability [KKSC03]. Figure 3 shows the demultiplexing steps in CORBA request processing. Demultiplexing uses a recursive data structure. To achieve predictability, recursion depth of this data structure has been statically bounded. Still, traversing it to demultiplex a service request potentially causes several page faults and cache misses at very high costs in the case of uncertainty about the memory state.

3 On the Multiplicative Nature of Uncertainty in Layered Systems

Real-life systems are not monolithic, but mostly structured into a layered hierarchy. Often, several layers interfere on a shared resource increasing the variability of execution times. We have already seen, how the brokerage of a service by a

middleware may cause page several faults. The page fault may cause a TLB miss, which in turn may cause several cache misses. Variabilities on different layers combine in a multiplicative way.

The sense behind all this is the following observation:

Observation 1. *At each level, uncertainty has to be accounted for in terms of a number of steps of lower levels. Let us assume that a step on level n costs* m_n^{n-1} *steps of level* $n-1$*. Then it costs* $\prod_{1 \leq i \leq n-1} m_{i-1}^i$ *machine cycles.*

Hence, in the worst case, a negative amplification of mechanisms on different layers will happen.

It should however be stressed that many modern powerful processors exhibit timing anomalies [LS99], which relate execution times on the different levels in more complex ways than the observation indicates. Timing anomalies are counter-intuitive dependencies of a program's execution times on the execution times of individual instructions. Faster execution of an instruction can lead to a longer execution time of the program, and slower execution of an instruction to shorter time for the program. Timing anomalies often are caused by cyclic influences between system components. One such dependency is the following: the contents of the instruction cache determines whether instruction fetch hits or misses the cache, a cache miss may prevent a branch prediction, a wrong branch prediction may ruin the instruction-cache contents. The resulting timing anomaly is, that the local worst case, a cache miss, prevents a branch misprediction, which would have caused a greater damage than the cache miss. Thus, the program runs faster in the case of the cache miss.

4 Towards a Rational Basis for Design

A promising approach to increase predictability would encompass all system layers. It would start with a multiplicative term of the kind given in Observation 1. As stated above, the design would have to exclude timing anomalies or bound the damage on cycles of dependencies. The design would then soften this multiplicative rule by reducing the factors on some layers, if it could make these layers behave predictably by static implementation decisions. This is a successful strategy to achieve a synergistic, quantifiable reduction of variability.

I admit that this looks like a dream in the light of the fact that the trends in systems development go towards less and less predictable systems. Experience with industrial projects lead me to believe that a discipline *Design for Predictability* is needed to reverse these trends and arrive at systems whose behavior as far as consumption of time, space, and energy is predictable.

Acknowledgements

Joint work on increasing the predictability of real-time systems is enjoyed with Lothar Thiele. Contributions to the discussion have come from Stephan Thesing, Oleg Parshin, Reinhold Heckmann, and Peter Marwedel.

References

[CC77] Patrick Cousot and Radhia Cousot. Static Determination of Dynamic Properties of Generalized Type Unions. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, volume 12(3), pages 77–94, Raleigh, NC, March 1977.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[FHL+01] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *EMSOFT*, volume 2211 of *LNCS*, pages 469 – 485, 2001.

[KKSC03] Arvind S. Krishna, Raymond Klefstad, Douglas C. Schmidt, and Angelo Corsaro. Towards predictable real-time java object request brokers. In *Real Time Technology and Applications Symposium*, pages 49–. IEEE, 2003.

[LMW99] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. *Design Automation of Electronic Systems*, 4(3):257–279, 1999.

[LS99] Thomas Lundquist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *20th IEEE Real-Time Systems Symposium*, 1999.

[Sha89] Alan C. Shaw. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15(7):875–889, 1989.

[TFW00] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and Precise WCET Prediction by Separate Cache and Path Analyses. *Real-Time Systems*, 18(2/3):157–179, May 2000.

[TW04] Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Systems*, 28:157 – 177, 2004.

[Wil05] Reinhard Wilhelm. Determination of bounds on execution times. In Richard Zurawski, editor, *Embedded Systems Handbook*, pages 14–1,14–24. CRC Press, 2005.

[WM97] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison Wesley, 1997.