Comparison of Trace Generation Methods for Measurement Based WCET Analysis (Preliminary Version) *

Stefan M. Petters
Department of Computer Science
University of York
United Kingdom
Stefan.Petters@cs.york.ac.uk

Abstract

Recent work on a measurement based worst case execution time estimation method uses observations of small units of the program. These observations are called execution traces and contain information of the execution path as well as the execution time of the units observed. This paper gives an overview on available options to extract the traces and highlights the advantages and disadvantages of these options.

1 Motivation

The measurement based worst case execution time (WCET) estimation method presented by Bernat et al. in [1] and [2] relies on the measurement of the execution time of small sections of code called traces as basic unit of the analysis. Within the approach, the observed traces are translated into execution time frequencies of the units. These execution time frequencies are interpreted as probability mass distributions and combined with a timing schema to provide a execution time profile of the worst case path through the program. The paper [1] focuses on the translation of traces to profiles and the combination of profiles rather than the production of the traces.

Within this paper we are adressing the different options for obtaining execution traces. Special attention will be given the factors of applicability, *overestimation* and *prolonged execution time*. Prolonged execution time indicates the impact of the method on the final executable code. For example added code will extend the execution time, if it stays in place. Opposed to this overestimation assesses the extra time within the measurements, which is not reflected in the execution time of the final executable code.

2 Discussion of Methods

Before going into the discussion of the methods a number of terms need to be defined. A measurement is made up of two observations. The points in the code where these observations are made are called *observations points* throughout this paper. The *observation interval* is the time which has passed between two observation points.

The exact meaning of a time stamp has to be considered as well. Within the paper a time stamp reflects usually a certain level of the execution pipeline. In out-of-order ex-

*The work presented in this paper is supported by the *European Union* under Grant Next TTA "IST-2001-32111".

ecution units of some processors[1], the observation point has to be either guarded by serialising instructions, or it has to be accepted, that the results of the measurements are fuzzy. Both cases add to the overall estimation of the execution time profiles. The serialising instruction disables the execution acceleration of the out-of-order execution engine, thus leading to a prolonged execution time, if the serialising instruction stays in place for the final production executable, or adds an unknown overestimation if it is somehow removed from the code. In the case of accepting more fuzzy results, a potentially considerable overestimation comes from the fact, that jitter of the time stamp at an observation point is added as well to the finished observation interval as well as the started observation interval.

2.1 Simulation

A simulator may be used to execute the program (cf. [1]). In this context we are only discussing *cycle accurate simulators* i.e. a simulator, which simulates not only the functional but also the temporal features of a processor accurately.

If such a simulator is available very accurate and usually perfectly reproducible measurements are possible. The granularity of measurements allows usually the time stamping of any individual assembler instruction. The missing requirement of instrumenting the code is a further plus for simulating the code. However, the cycle accurate simulator suffers from one of the major problems of static analysis. Additionally the cycle accuracy of a simulator usually only applies to the CPU itself and does not imply accurate timing of peripheral hardware e.g. anything from SDRAM to PCI bus hardware. As the simulation takes enlarges the execution depending on the complexity of the modelled processor by up to two orders of magnitude on similar hardware, the problem of time needed for the trace generation becomes an issue. This can be somewhat relieved by using a cluster of computers to do the simulations.

[1] These effects obviously only occur, if the time stamp is taken within the out-of-order execution core of the processor. The fetch and commit stages are always in-order execution units.

2.2 Light Weight Software Monitoring

Software monitoring relies on instrumentation code placed in the software to be investigated. The code inserted may record various data of the system. In our case we are interested in event triggered software monitoring in which the instrumentation code is placed at the desired observation points in the application code and is executed whenever the observation point is reached, opposed to periodic software monitoring (cf. [3]), in which the application code is interrupted in merely period distances.

While the term software encoding encompasses all kind of data collection, we are only intersted in identification of the code executed and the time stamp corresponding the execution of this code. The prefix *light weight* indicates, that the code inserted tries to minimise its impact on the execution time. As opposed to *heave weight* software monitoring, which tries to make a safe estimate by establishing the worst case state of a processor at the start of each observation interval. The time passed is usually taken from an internal cycle counter. The POSIX tracing standard as used by Terrasa et al. in [4] shows a implementation of such a light weight software monitor.

On the positive side it can be noted that the instrumentation code will be left in place after the measurements are completed. The instrumentation code may be quite simple, which makes a comparable quick port of the method to another architecture possible. A major drawback of this method is the additional variablity introduced by this method especially on high performance processors with caches. Either the memory area is mapped as non-cachable, which makes the access time in storing the sampled data very long or the cache access patterns add to the temporal variability of the code. The amount of memory necessary to store the trace data can be considerable, which adds to the effort of bounding the impact on code variability.

The code added may be quite computational expensive. The POSIX tracing standard allows for interfaces to actual collect the trace. This functionality leads to a prolongation of the execution time. This prolongation may be up to one order of magnitude if the distance between two observation points is just one basic block in control intensive applications[2].

2.3 Heavy Weight Software Monitoring

As has been explained in the previous section heavy weight software monitoring establishes the worst possible state at the beginning of each measurement. This has been used by Petters in [5]. This implies that an observation point implies two time stamps. One for the completion of an observation interval, prior to the disruption of the working sets of caches, branch prediction etc. and a second one starting a new observation interval after the disruption.

The code has to be replaced after the measurements have been completed. In order to limit the code to be replaced the main code of the instrumentqation should be implemented as a function which is called with an identifier from all observation points. In this case only the calls to the measurement routine have to be masked or the measurement routine is replaced with an return. Depending on the architecture the first renders the final operational executable usually faster than the second solution.

As an advantage the disruption of execution units may be preceeded by a write out of the obtained measurement data to disc. This limits the amount of memory necessary for the storage of the trace data. The overestimation by this method may be seriously. Depending on the size two orders of magnitude are possible, if one tries to trace individual assembler instructions. By following coding style and tracing larger code ($\tilde{4}$-10 basic blocks) the overestimation may be reduced to a factor of two. However, this raises the issue of test coverage. In [5] this has been solved by enforcing paths within the observation interval. Adding information about the path taken within the observation interval by introducing very small additional instrumentation code may solve this problem as well.

More than with the other methods this method raises the question on the validity of the obtained results under the fact that the code in the final production executable is not identical to that one under investigation during the measurements.

2.4 Hardware Supported Software Monitoring

As with the previous methods, instrumentation code is added to the application at the observation points. This code delivers the location data to a reserved external port. The timing is taken either by the hardware device probing the port or is taken from in internal cycle counter and written in a separate access. The necessary port pins for this method are usually quite costly and in the general case this will only be applicable on microcontroller and similare complex processors. High performance processors generally do not have free accessible pins and the ones accessible via buses raise questions on the time needed for transactions. The major advantage of monitoring method is the small impact of the code on the execution time.

2.5 (Software Supported) Hardware Monitoring

This option comes in two flavours. On one hand are bus monitors applied on one or more buses of the processor, on the other hand is hardware built in by processor manufactures to support debugging.

The first case has been used in the past by tracking instruction flow on the address bus of some processors. This has been inhibited by the use of instruction caches and has almost completely vanished as a tracing mechanism for the software.

[2] Control intensive applications consist of small basic blocks and are therefore vulnerable to heavy prolongation

However, in recent years debugging interfaces were equipped with additional features to allow for timing information to be extracted out of the code. Namely the Tricore OCDS and the more generally available NEXUS2 (cf. [6]). These interfaces allow the sampling of time at given points. The NEXUS2, for example, takes a time stamp at every taken branch instruction. Thus a complete trace is available for further analysis. The majore challenge is to tap into these ports and extract the data. Commercial tools initially intended for debugging provide interfaces to do this. The problem consists here of getting the raw data out of the tools.

One major advantage is, that no or only minimal software instrumentation is needed[3]. A problem lies in the potential bandwidth problem. If the observation intervals are too short, the trace data may not be completely transmitted over the debugging interface. In this case, the debugging tools try a interpolation of the measurements, which defeats the purpose of taking the traces in the first place. Some debugging tools offer to hold the processor, if two observation points are too close. However, this can only be applied to the processor itself and peripherals clocked directly with the CPU clock. More remote hardware may behave differently in the temporal domain, if the execution is put on hold at an arbitrary instant.

3 Conclusion

As it is, there is no one-fits-all solution to the problem of trace generation. While especially comercially supported hardware monitoring has some appealing advantages over the other methods, the still limited availability of processors makes it necessary to look at the alternatives.

[3] In some cases one might want to enforce a observation point. This is generally possible by adding hand crafted code

References

[1] G. Bernat, A. Colin, and S. M. Petters, "WCET analysis of probabilistic hard real–time systems," in *Proceedings of the 23rd Real-Time Systems Symposium RTSS 2002*, (Austin, Texas, USA), pp. 279–288, Dec. 3–5 2002.

[2] G. Bernat, A. Colin, and S. M. Petters, "pWCET: a tool for probabilistic worst case execution time analysis of real–time systems," technical report YCS353 (2003), University of York, Department of Computer Science, York, YO10 5DD, United Kingdom, Apr. 2003.

[3] L. Svobodova, *Computer Performance Measurement and Evaluation Methods: Analysis and Applications*. No. 2 in Elsevier Computer Science Library, New York: American Elsevier Publishing Company, Inc, 1976.

[4] A. Terrasa, I. Paches, and A. Garcia-Fornes, "An evaluation of the posix trace standard implemented in rt-linux," in *Proceedings of the IEEE International Symposium on Performance Analysis and Software*, 2000.

[5] S. M. Petters, *Worst Case Execution Time Estimation for Advanced Processor Architectures*. PhD thesis, Institute for Real–Time Computer Systems, Technische Universität München, Munich, Germany, Sept. 2002.

[6] IEEE-ISTO, *IEEE-ISTO 5001-1999, The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface*, 1999. Available at `http://www.nexus5001.org/`