In: Proceedings of the the 7th International Conference on Real-Time Computing Systems and Applications RTCSA'00, December 2000, ©IEEE Computer Society

Bounding the Execution Time of Real-Time Tasks on Modern Processors*

Stefan M. Petters
Institute for Real-Time Computer Systems
Prof. Dr.–Ing. Georg Färber
Technische Universität München, Germany
Stefan.Petters@rcs.ei.tum.de

Abstract

Finding a tight estimate on the execution time of tasks in embedded hard real-time systems is gaining complexity and importance. Uptodate processors are equipped with acceleration techniques to bridge the gap between the fast core frequency of the processors and the comparably slow main memory. On the other hand the application field of real-time systems is continuously widening to more complex systems. To keep up with the evolution of processors it is proposed to lay more weight on measurement and less on modeling. By analyzing the control flow graph the compiler uses for optimization, systematic information on how the code has to be measured can be gained. Using this information the code can be automatically instrumented and measured to bound the execution time of the real-time tasks.

1 Introduction

Modern processors work at core frequencies which are way beyond the capabilities of the main memory. Thus more and more intelligence is integrated into the hardware of the processors in order to close the gap between the speed of the processor and that of the main memory. Uptodate architectures include features like e.g. caches, speculative execution and branch prediction. Intel's P6 family even mimics externally a CISC machine but has internally a RISC core which interprets the incoming code and reorders the internal code for optimal execution. Meanwhile the domain of real-time systems widens to more complex applications. In the automobile industry more and more functionality is realized in software (e.g. ESP and x-by-wire) and video based robotics is moving to safety critical application fields like e.g. teleoperation and telesurgery.

To tackle the problem of increasing software and hardware complexity it is proposed in this paper to use measurements instead of modeling the processor to the very detail. Currently the focus is laid on the Intel P6 architecture, since it is one of the most complex processors, but the methods are easily portable.

In the next section related work is resumed followed by a detailed description of the method. In Section 4 the results of case studies are presented before future work is indicated in the last section.

2 Related Work

Chapman, Burns and Wellings utilize formal proof methods in [1]. The program is run by symbolic execution. In the beginning all variables are undefined and as the program is executed, the range of values of a variable is reduced whenever possible. This approach suffers from an exploding state space and thus is only feasible for small scale software systems.

White, Müller and Harmon have modified the compiler to emit additional control flow information very similar to our technique in [8]. They model the cache and the processor very detailed. With 512 bytes the cache used in this paper is not very large. The problem is also the complexity of the method. The work of Ferdinand and Wilhelm in [3] bases on the previous approach. They use data dependency analysis and a technique, usually utilized for program restructuring, to determine the worst case bounds on cache misses. As this method fits only restricted classes of programs it is complemented by persistence analysis.

Ermedahl et al. combine their approaches in [2]. A

*The work presented in this paper is supported by the *Deutsche Forschungsgemeinschaft* as part of a research programme on "Rapid Prototyping for Embedded Hard Real–Time Systems" under Grant Fa 109/11-2.

high level analysis searches for mutual exclusive paths and the number of loop iterations. Very similar to [1] a kind of symbolic execution is used. The low level architecture dependent part is limited to analyzing caching and pipelining single basic blocks which contain no loops or procedure calls and only static variables. The analyzed programs were 10 to 80 lines long.

The work in [9] aims not only at the WCET but also on the power consumption of an embedded System. Wolf and Ernst limit the search to feasible paths by using program path analysis. An additional narrowing of the search domain is gained by investigating the context dependent control flow. In contrast to the approach presented in this paper the actual WCET is determined using a simulator or an emulator.

Lee et al. regard the effect of caches in preemptive task systems in [4]. Traditional schedulability analysis does not cover the time of cache restoration after preemption. First the number of useful blocks is computed. This is the set of memory blocks needed by the program for further execution. Then a phasing of tasks is introduced to eliminate infeasible task interactions.

The work of Lundqvist and Stenström in [5] describes the problem of timing anomalies of modern processors corresponding to the characterization in [7]. They present a method to avoid the timing anomalies by modifying the code. Thus it is possible to use the *conventional* WCET/BCET determination approaches. A major drawback of the described method is that it makes compiler optimizations nearly impossible to use.

3 Measurement approach

For the formal proof that a given real-time system meets all its deadlines the WCET of all real-time tasks has to be known. To perform the schedulability analysis for earliest deadline first scheduling considering mutual exclusive tasks the BCET has to be known as well. To efficiently determine WCET and BCET for complex hardware/software systems a measurements is proposed. Figure 1 illustrates all necessary operations which are explained in the following sections.

3.1 Requisite Knowledge

One of the basic ideas of the measurement approach is that detailed modeling of the processor and the mandatory subsystems is avoided. Nevertheless, knowledge about the processor, the memory subsystem and other components of the real-time system is essential to get reliable and tight bounds for the execution time of a piece of code.

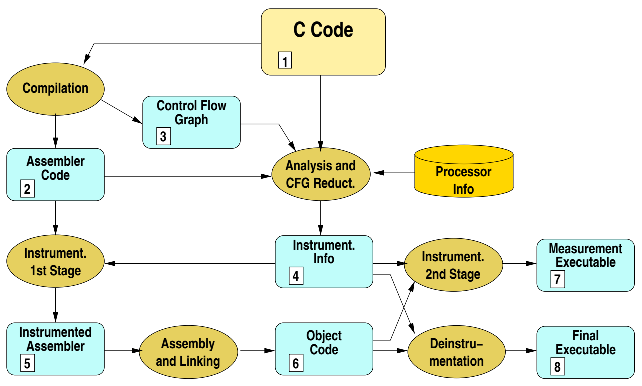

Figure 1. Tool Chain

First of all the type and impact of all acceleration techniques utilized by the processor has to be known. In the following a few examples shall demonstrate the detail level and the usage of this knowledge. For the P6 family processors, it can be seen from the data sheets that the processor interprets the external x86 CISC code to internal RISC format and stores it in an *Instruction Pool*. Up to 50 external CISC instruction can be interpreted and the resulting code executed in any order which preserves the semantics of the code. This leads to three important aspects during analysis and measurement: Due to the parallel execution of some instructions, additional code may lead to shorter instead of longer execution times by avoiding pipeline stalls. In order to get an exact measurement, a serializing instruction has to be inserted around the measurement code, i.e. an instruction, that causes the RISC execution units to execute all code in the *Instruction Pool* before taking the time stamp. The overestimation by forcing the serialization has to be taken into account for the BCET (cf. Sec. 3.5).

Translation Look-aside Buffers (TLB) have to be considered by invalidating the TLB entries for WCET measurements. Other examples of such techniques are Branch Target or Return Stack Buffers.

The individual size and type of all caches (instruction, data or unified cache) have to be known as well as their cache-line size, associativity and replacement scheme. This is necessary to enforce a reproducible and known state of the system during measurement.

The knowledge of the type of main memory (e.g. SRAM), the used access modes (burst or single access) and the access cycles may be necessary to add/subtract a correct safety margin on the measured values to produce the WCET/BCET. Additionally the refresh rate and the duration of a refresh cycle are of interest for dynamic RAMs. Latencies of all other components like system bus (e.g. PCI bus) is also have to be known.

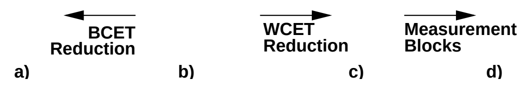

Figure 2. Example of a CFG Reduction

3.2 Control Flow Graph (CFG) Analysis

In order to reduce the number of paths under investigation CFG analysis is used. The compiler was modified to emit the CFG. To be able to make the necessary changes to the compiler the GNU compiler *gcc* was used. This CFG $\boxed{3}$ allows an easy mapping of the assembler code $\boxed{2}$ to the source code $\boxed{1}$ even when all compiler optimizations are enabled. Thus the approach can be combined with a method to avoid mutual exclusive paths and extract the bounds on the loop-count for the loops like in e.g. [2] and [9]. Currently, the loop-counts have to be specified in annotations.

The CFG is reduced as depicted in Figure 2. A clip of a CFG is shown as basis in Fig. 2b). A path can only be omitted for WCET/BCET measurement, when the difference between this and alternative paths consists of more than a few additional assembler instructions, since e.g. pipeline stalls may lead to a longer execution time for a path with additional instructions. Thus, two paths remain to be measured for the BCET in Fig. 2a) and the path from node 27 to node 29 is still under investigation for the WCET measurement (Fig. 2c).

The analysis of the CFG, the assembler code and the subsequent reduction of the CFG results in a description of paths to be measured and additional instrumentation which has to be included into the assembler code $\boxed{4}$. Each path description corresponds to one path and specifies the necessary instrumentations.

3.3 Handling the Complexity

A common problem in WCET/BCET analysis is the exploding state space for complex systems. The approach followed here is to trade some of the tightness of the bounds on the execution time for a lower complexity by splitting the application into measurement blocks (cf. Fig. 2d). These measurement blocks are analyzed and measured separately.

For the measurement of the execution time, each measurement block has a measurement routine put in front and after the code to be measured. Additionally this routine enforces all useful data and code out of the cache. In [7] the code just wrote back and invalidated the cache. The construction of this routine is critical and requires knowledge about the acceleration techniques of the processor as stated in Section 3.1. The advantage of filling the cache with useless data and code instead of leaving the cache clean lies in the avoidance of an additional penalty to the WCET. The cache invalidation and the flush of the execution units lead to an overestimation of the execution time, which has to be corrected for BCET measurements by subtraction of a time bonus (cf. Sec. 3.5).

The placement of the measurement block boundaries is critical. Since an inapt placement of these lead to a severe loosening of the execution time bounds, the partitioning is done by hand.

Loops inside code can be divided into two categories. The first are loops where the execution path inside the loop does not depend on input data. Such loops can often be found in e.g. image processing and signal processing algorithms. The effect of input data on the number of loop iterations has to be eliminated, i.e. maximized for WCET measurement and minimized for BCET measurement.

The second category of loops embodies looser bounds on execution time. Especially loops with a large number of iterations lead to an enormous overhead for instrumentation inside the loop body. In some cases it is even necessary to measure the worst and best case of one loop iteration and multiply the result with the number of iterations. The resulting underestimation of the BCET caused by the subtracted cache bonus and overestimation of the WCET due to the virtually not used cache are considerable.

3.4 Instrumentation and Measurement

The instrumentation is done in two stages. In the first stage additional code $\boxed{5}$ is inserted into the assembler output of the compiler $\boxed{2}$. This is necessary to force the loops to their specified execution count and to add the code that triggers the measurement which is described in greater detail below.

The optimizations of the compiler leave the resulting code for the loop controlling part often in a way, that is hardly to interpret automatically. Thus during measurement the loop is controlled by an instrumented control structure which is placed around the entire original loop. After this stage the instrumented assembler code is assembled and linked to an executable [6].

The second part of the instrumentation is done on the object code and forces the execution of the selected measurement paths [7]. During the measurement phase this is done repeatedly to be able to measure all possible paths. The original loop controlling structures inserted by the compiler are disabled during this stage by substituting the controlling conditional jumps by `nops`. Thus the code loop is controlled by the instrumented code which enforces the maximum loop-count. For the alternatives the conditional jump is replaced either by an unconditional one or by `nops`.

After all measurements have been made, the code has to be de-instrumented [8]. Normally this is avoided for the production code, since it is in most cases done by hand and therefore error prone. In contrast to that the de-instrumentation is strictly necessary in this approach to revert the code to a state as intentioned in the source code. A software monitor is used for the measurements in contrast to previous work in [6] . At each measurement block boundary a call to a small measurement routine is inserted as described previously in this section. The routine takes two time stamps. One at the start of the routine, to complete the measurement of the last measurement block and one at the end of the routine, to start the measurement of the next.

To achieve a high resolution of the time stamps the processors internal cycle counter is utilized. After the first time stamp is taken, the cache is filled with data and code which is useless for the application under investigation. This is done by executing code which utilizes all instruction caches and loading data which covers all data caches available. The memory utilized for this holds the measurement records taken by the routine. A fake modify on the data in the cache forces that the cache is written back before replacing it by the application data. By doing this the worst case cache state is created. The Branch Target Buffer are initialized to a known state and for the WCET measurement the TLB is invalidated.

3.5 Time Penalty and Bonus

To use the measured times for BCET and WCET estimation, a few corrections are necessary. A list of necessary and possible corrections will be given. Not all of these will apply on every given processor.

The WCET of each measurement block can be reduced by the BCET of the measurement routine. On the P6 architecture this is only useful for small measurement blocks. Another influence of the instrumentation is the substitution of conditional jumps by unconditional ones or `nops`. For each the difference between the measured operations and the conditional jump with the branch target buffer (BTB) set to the wrong destination has to be added to the measured execution time. The effect of the BTBs on conditional jumps remaining in the code have to be considered as well.

We have to assume that the measured WCET was not affected by the memory refresh of the DRAM and we need to correct it by the penalty T_{rp} which is computed using the following equation:

$$T_{rp} = T_r * \lceil \frac{WCET_{meas}}{T_c} \rceil$$

Where $WCET_{meas}$ is the measured WCET, T_c is the interval between two memory refresh cycles and T_r is the time for a memory refresh cycle. Typical values for T_r are in the order of magnitude of 100 ns. The cycle period T_c is usually some 10 us.

Given that there are latencies in the bus system and other system components, the those latencies not reliably covered by the measurement must be added to the WCET. For example, the latency penalty for the PCI bus is gained in a straightforward way by multiplying the worst case number of accesses with the worst case latency. The modeling of more complex latencies introduced by direct memory access or the latencies of the CAN bus is beyond the scope of this paper.

In contrast the BCET only has to be corrected by time bonuses. Other than for the WCET, we have to assume that each measurement block was affected by the maximum impact of memory refresh cycles. The bonus T_{rb} is computed analogous to T_{rp} for the WCET.

The WCET of the measurement routine has to be subtracted and the effect of the useless data in cache at the beginning of each measurement block taken into account for the BCET. All data referenced and code has to be assumed to be in cache. To fix this, the cache refill time has to be subtracted from the measured BCET. In a simple approach the refill time of the complete cache utilized by the program is subtracted. A greater accuracy can be gained by only subtracting the amount of cache used in the part of the task under investigation. By applying the method of *useful blocks* described in [4], only the cache used in the measurement block has to be covered.

Analogous to the WCET those latencies covered by the measurement and possibly absent during execution have to result in a bonus as well. Finally the stall of the execution units by the measurement code and the prolonged execution by the loop instrumentation have to be taken into account.

4 Measurement Results

For the measurements an Intel Pentium II Processor with 233 MHz was used. A matrix multiplication program was chosen to measure the deviation of medium sized application. The execution time of this example deviated between 286219 and 287284 cycles. This corresponds to an error of 0.3 %.

As an example from the image processing field, an application was implemented which segments 128x128 pixel images and recognizes 3 different types of objects and the orientation of two of them (the third is rotary symmetrical). Since is no search for the orientation of the third item, the BCET is rather small compared to the WCET. The measured BCET was 241529 cycles. Correction the result with a complete cache refill time would leave the corrected BCET with 0. Scaled to the actual size of the task, which was 73 KByte for code and data, the bonus for possibly cached data and code would be 1.5 10^5 cycles. The additional bonuses for a memory refresh cycle and the measurement routine are already covered by the rounded up cache bonus. Thus the measured BCET would be adjusted to half its value for later use.

To evaluate the results measurements with uninstrumented code and real data were taken. The WCET as forced by the instrumented code was 198424706 cycles. Compared to the execution time of 198387085 cycles with real data the bound is extremely tight. The best case was achieved when leaving out the code responsible for cache destruction and code serializing and doing several measurements in a row. The best case taken from these measurements with real data was 215231 cycles. The corrected save estimate of the BCET was only cycles. The reason for the estimate being only 42,5% of the the real BCET lies in the fact that 48 KByte of data is not touched in the best case, but is covered in the cache bonus of 73 KByte. An analysis of memory usage could further minimize the gap between real BCET and the BCET used for analysis.

5 Conclusion

In this paper an approach for a feasible worst case and best case execution time estimation is presented. For the analysis the control flow graph generated by the compiler is used and thus the analysis of fully optimized code is possible. A method for automated instrumentation, measurement and de-instrumentation is introduced. It has been shown how the complexity of large tasks can be reduced by partitioning. While the WCETs can be bounded very efficiently and tightly by the presented measurement methods, due to the necessary time bonus for the caches, the BCETs leave room for optimization.

Future work will focus on including the aspects of an underlying real-time operating system. Evaluation of the method on other processing hardware is currently a work in progress. As mentioned before the underestimation of the BCETs has to be reduced.

References

[1] R. Chapman, A. Burns, and A. Wellings. Integrated program proof and worst–case timing of SPARC Ada. In *Proceedings of the ACM SIGPLAN Language, Compiler, and Tool Support for Real-Time Systems (LCTS) workshop*, Orlando, Florida, June 1994.

[2] J. Engblohm, P. Altenbernd, and A. Ermedahl. Facilitating worst–case execution time analysis for optimized code. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, Berlin, Germany, June 1997.

[3] Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real–time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, Montreal Canada, June 19–20 1998.

[4] C.–G. Lee, J. Hahn, Y.–M. Seo, S.L. Min, R. Ha, S. Hong, C.Y. Park, M. Lee, and C.S. Kim. Bounding cache-related preemption delay for real–time systems. In *18th IEEE Real–Time Systems Symposium*, San Francisco USA, December 3–5 1997.

[5] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the IEEE Real–Time Systems Symposium*, Phoenix, AZ, December 1999.

[6] Stefan Petters, Annette Muth, Thomas Kolloch, Thomas Hopfner, Franz Fischer, and Georg Färber. The REAR framework for emulation and analysis of embedded hard real–time systems. *Design Automation for Embedded Systems*, 5(3):237–250, August 2000.

[7] Stefan M. Petters and Georg Färber. Making worst case execution time analysis for hard real–time tasks on state of the art processors feasible. In *Proc. of the 6th Int. Conf. on Real–Time Computing Systems and Applications*, Hongkong, December 13–15 1999.

[8] R. White, F. Mueller, C. Healy, D. Whalley, and M. G. Harmon. Timing analysis of data caches and set–associative caches. In *3rd IEEE Real–Time Technology and Applications Symposium*, Montreal Canada, June 9–11 1997.

[9] F. Wolf and R. Ernst. Execution cost interval refinement in static software analysis. *Journal of Systems Architecture, The EUROMICRO Journal*, Special Issue on Modern Methods and Tools in Digital System Design, 2000.