

ESTIMATION OF WCET USING A LITTLE LANGUAGE TO DESCRIBE MICROCONTROLLER AND DSP ARCHITECTURES

Adriano TAVARES, Carlos COUTO

Department of Industrial Electronics, University of Minho, 4800 Guimarães, PORTUGAL,
<atavares, ccouto>@dei.uminho.pt

Abstract. *A method for analysing and predicting the timing properties of a program fragment will be described. First a little language implemented to describe a processor's architecture is presented followed by the presentation of a new static WCET estimation method. The timing analysis starts by compiling a processor's architecture program followed by the disassembling of the program fragment. After sectioning the assembler program into basic blocks, call graphs are generated and these data are later used to evaluate the pipeline hazards and cache miss that penalize the real-time performance. Some experimental results of using the developed tool to predict the WCET of code segments using some Intel microcontroller are presented. Finally, some conclusions and future work are presented.*

Key Words. *Little Language, Worst-Case Execution Time, Machine Description, Language Paradigm, Timing Scheme, Timing Analysis*

1. INTRODUCTION

Real-time systems are characterized by the need to satisfy a huge timing and logical constraints that regulate their correctness. Therefore, predicting a tight worst case execution time of a code segment will be a must to guarantee the system correctness and performance. The simplest approach to estimate the execution time of a program fragment is for each arithmetic instruction, counting the number of times it appears on the code, express the contribution of this instruction in terms of clock cycles and update the total clock cycles with this contribution. Nevertheless, these approaches are unrealistic since they ignore the system interferences and the effects of cache and pipeline, two very important features of some processors that can be used in our hardware architecture. Some very elaborated methodology for WCET estimation, such as, Shaw [1] were developed in the past, but none of them takes into account the effects of cache and pipeline.

Theoretically, the estimation of WCET must skip over all the profits provided by modern processors, such as caches, and pipeline (i.e., each instruction execution suffers from all kind of pipeline hazards and each memory access would cause a cache miss) as they are the main

source of uncertainty. Experimentally, a very pessimistic result would be obtained, and so, making useless those processor's resources. Some WCET estimation schemes oriented to modern hardware features, were presented in the last years, and among them we refer to: Nilsen [2], Steven Li [3], Whalley [4], and Sung-Soo Lim [5]. However, these WCET estimators do not address some specificity of our target processors (microcontrollers and DSPs), since they are oriented to general-purpose processor. Therefore, we propose a new machine independent estimator, implemented as a little language for architecture description. Such a machine independent scheme, based on a little language was used before by Tremblay[6] to generate machine independent code, Proebsting and Fraser [7] to describe pipeline architectures and Nilsen [5] to implement a compiler, simulator and WCET estimator for pipeline processors.

2. LITTLE LANGUAGE

The purpose of any little language, typically, is to solve a specific problem and, in so doing, simplify the activities related to the solution of the problem. Our little language's statements are created based on the tasks that must be performed to describe processor's architectures in terms of structure and functional architecture of the interrupt controller, PTS (Peripheral Transaction Server), PWM (Pulse Width Modulation), WG (Waveform Generator), and HIS (High Speed Input), instruction set, instruction semantics, addressing modes, processor's registers, instruction coding, compiler's specificity, pipeline and cache resources, and so on. For our little language, we adopt a procedural and modular paradigm (language paradigm defines how the language processor must process the built-in statements), such that modules are independent from each other, the sequence of modules execution does not matter, but the register module must always be the first to be executed and within each module an exact sequence of instructions is specified and the computer executes them in the specified order. A processor's architecture program is written by modules, each one describing a specific processor's feature such as instruction set, interrupt structure and mechanism, registers structure, memory organization, pipeline, data cache, instruction cache, PTS, and so on. A module can be defined more than once, and it is a processor language job to verify the information consistency among them and concatenate all them into a single module.

The disassembler process implemented into four phases, has as input an executable file contains the code segment that one wants to measure and the compiled version of the processor's architecture program. The disassembler process starts at the start-up code address (startup code is the bootstrap code executed immediately after the reset or power-on of the processor) and follows the execution flow of the program:

1. starting at the start-up code address follows all possible execution paths till reaching the end address of the "main" function. At this stage, all function calls are examined and their entry code addresses are pushed into an auxiliary stack,
2. from the entry address of the "main" function, checks the main function code for interrupt activation,
3. for each active interrupt, gets its entry code address and pushed it into the auxiliary stack,
4. pops each entry address from the auxiliary stack and disassemble it, following the function's execution paths.

The execution of the simulation module is optional and the associated process is described by a set of operation introduced using the function "SetAction". For instance, the simulation process, including the flag register affectation, associated to an instruction are described by a

set of operation specified using “SetAction” calls. Running the simulation process before the estimation process, will produce a more optimistic worst case timing analysis since it can:

1. rectify the execution time of instructions that depend on data locations, such as stack, internal or external memory,
2. solve the indirect address problem by checking if it is a jump or a function call (function call by address),
3. estimate the iteration number of a loop.

The WCET estimator module requires a direct interaction with the user as some parameters are not directly measurable through the program code. Note that, the number of an interrupt occurrence and the preview of a possible maximum iterations number associated to an infinite loop are quite impossible to be evaluate using only the program code. The WCET estimation process was divided into two phases:

1. first, the code segment to be measured is decomposed into basic blocks,
2. for each basic block, it will be estimated the lower and upper execution time, using the shortest path method and a timing scheme [1].

The shortest path algorithm with the basic block graph as input is used to estimate the lower and upper bound on the execution time of the code segment. For the estimation of the upper bound, it is used the multiplicative inverse of the upper execution time of each basic block. A basic block is a sequence of assembler’s instructions, such as, only the first instruction can be prefixed by a label and only the last one can be a control transfer instruction. The decomposition phase is carried out following the steps below:

1. rearrangement of code segment to guarantee the visual cohesion of a basic block. Note that, the ordering of instructions by address make more difficult the visualization of the inter basic block control flow, due to long jump instructions that can occur between basic blocks. To guarantee that visual cohesion, all sequence of instructions are rearranged by memory address, excluding those one located from long jump labels which are inserted from the last buffer index,
2. characterization of the conditional structure through the identification of the instructions sequence that compose the “if” and “else” body,
3. characterization of the loop structure through the identification of the instructions sequence that composes the loop body, control and transfer control. It is essential to discern between “while/for” and “do while” loop since the timing schemes are different,
4. After the identification and characterization of the control and loop structures, it will be built a basic block graph, showing all the execution paths between basic blocks
5. and for each basic block, find the lower and upper execution time.

2.1. Pipeline Modelling

The WCET estimator presented so far, considers that an instruction’s execution is fixed over the program execution, i.e., it ignores the contribution of modern processors. Note that, the dependence among instructions can cause pipeline hazards, introducing a delay in the instructions execution. This dependence emerges as several instructions are simultaneously executed and as the result of this parallelism execution among instructions, the execution time of an instruction fluctuates depending on the set of its neighbouring instructions. Our little language analyses the pipeline using the pipeline hazard detection technique suggested by

Proebsting and Fraser [7] and models the pipeline as a set of resources and each instruction as a process that acquires and consumes a subset of resources for its execution. Special purpose functions, such as, “setPipeStage(Mn)” and “SetPipeFunctionalUnit(Mn,num)” are used to define the pipeline stages and functional units, respectively. For each instruction, there is a set of functions to specify the pipeline stage each source operand must be available, the pipeline stage the output of the destination operand becomes available, each pipeline stage required to execute an instruction and the execution time associated to that stage, and the control hazard cost associated to a branch instruction.

The pipeline analysis of a given basic block must always take into account the influences of the predecessor basic blocks (note that, the dependence among instructions can cause pipeline hazards, introducing a delay in the instructions execution), otherwise, it leads to an underestimation of the execution time. Therefore, at the hazard detection stage of a given basic block, it will be always incorporate the pipeline’s state associated to the predecessor basic blocks over the execution paths. The resources vector that describes the pipeline’s state it will be iteratively updated by inserting pipeline stalls to correct the data and/or structural hazards when the next instruction is issued. If these two hazards happen simultaneously, the correction process start at the hazard that occurred first and after it will be checked if the second one still remains. The issuing of the new instruction will be always preceded by the updating of the previous pipeline’s state, achieved by shifting the actual pipeline resource vector one cycle forward.

The pipeline architectures, usually, present special techniques to correct the execution flow when a control hazard happens. For instance, the delay transfer control technique offers the hardware an extra machine cycle to decide the branch. Also, special hardware is used to determine the branch label and value condition at the end of the instruction’s decode. As one can conclude, the execution of delay instructions does not depend on the branch decision and it is always carried out. So, we model the control hazard, as being caused by all kind of branch instruction and by adding the sum of execution time of all instruction in the slot delay to the basic block execution time.

2.2. Cache Modelling

Cache is a high speed and small size memory, typically, a SRAM that contains parts of the most recent accesses to the main memory. Nowadays, the time necessary to load an instruction or data to the processor is much longer than the instruction execution time. The main rule of a cache memory is to reduce the time needed to move the information from and to the processor. An explanation for this improvement, comes from the locality of reference theory – at any time, the processor will access a very small and localized region of the main memory and the cache load this region, allowing faster memory accesses to the processor.

In spite of the memory performance enhancement, the cache makes the execution time estimation harder, as the execution time of any instruction will vary and depends on the presence of the instruction and data into the caches. Furthermore, to exactly know if the execution of a given instruction causes a cache miss/hit, it will be necessary to carry out a global analysis of the program. Note that an instruction’s behaviour can be affected by memory references that happened long time before. Adversely, the estimation of WCET becomes harder for the modern processors, as the behaviour of cache and pipeline depend on each other. Therefore, we propose the following changes to the algorithm that takes into account the pipeline effects:

1. Classify the cache behaviour [4] for any data and instruction as cache hit or cache miss before the analysis of the pipeline behaviour,

2. Before the issuing of an instruction, verify if there is any cache miss related to the instruction, and if any, apply the miss penalty beforehand and then the detection and correction of pipeline hazards.

3. EXPERIMENTAL RESULTS

By the moment, we will present some results using the 8xC196 Intel microcontrollers as they are the only ones present all needed execution time information in the user's guide. But we hope soon to present results of experiments with modern processor such as, some Texas Instruments DSPs, Intel 8xC296, PICs and so on. At a first stage, the WCET estimator built the call graph given at the lower right quadrant of fig.2 and then, func() identified by the label C_2192 will be processed and providing a similar screen. At the upper right quadrant, information such as execution time of individual basic blocks, basic block control flow and function execution time are presented. At the lower right quadrant, can be presented the assembler code translated by the disassembler from the executable code, the call graph and the simulator state. The upper left quadrant presents parts our little language program describing the microcontroller architecture.

4. CONCLUSIONS

A very friendly tool for the WCET estimation was developed and the results obtained over some Intel microcontroller were very satisfactory. To a complete evaluation of our tool we will realize more test using other classes of processors such as DSPs, PICs and some Motorola microcontrollers. A plenty use of this tool requires some processors informations, such as, the execution time of each instructions composing the processor instruction set, sometimes not provided in the processor user's guide. In such case, to time an individual instruction, we recommended the use of the logic analyzer to trigger on the opcode at the target instruction location and on the opcode and location of the next instruction.

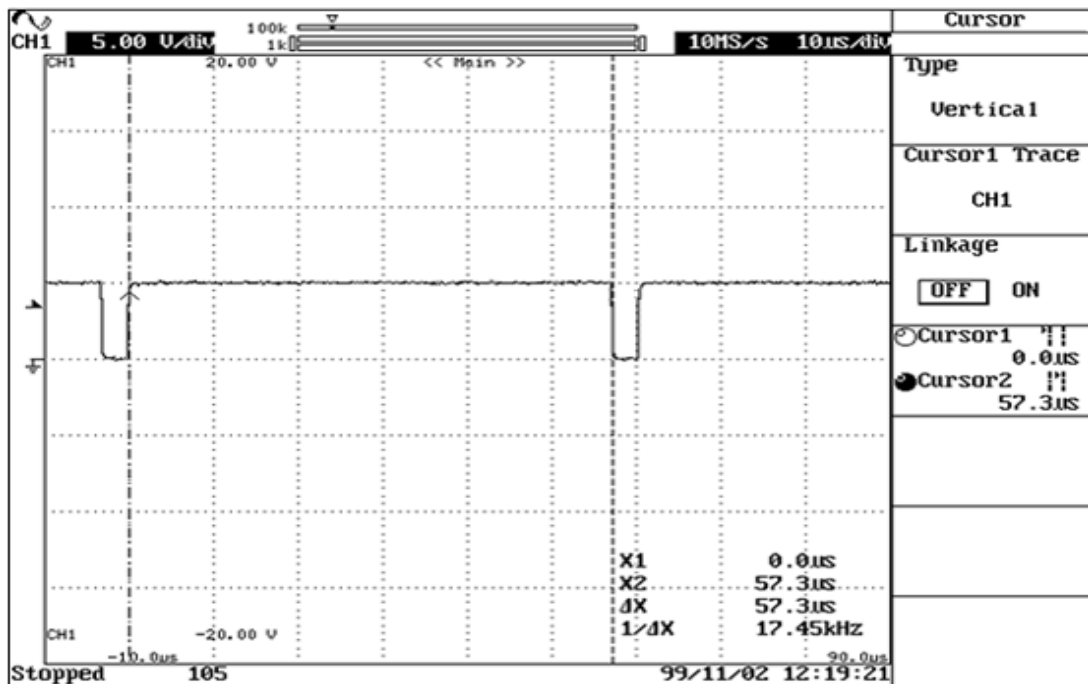


Fig. 1. Digital Oscilloscope triggers on the opcode of the first and last instructions of a code segment

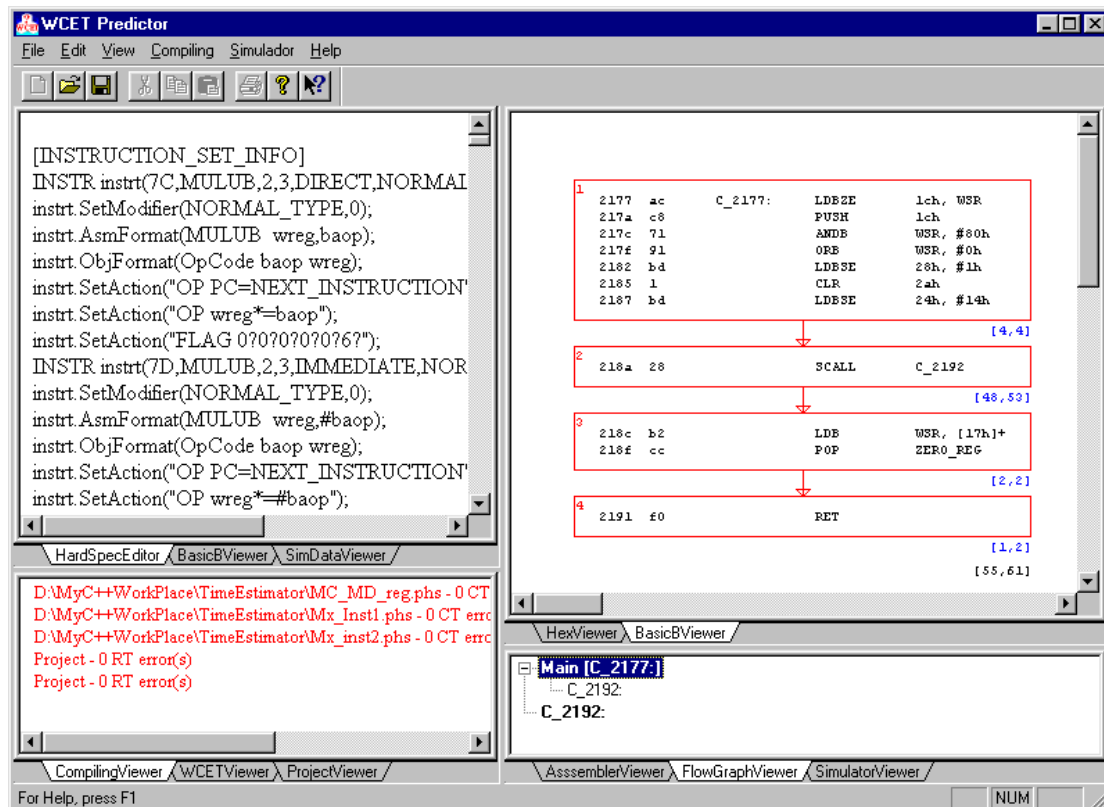


Fig. 2. WCET = 61 μ s was statically estimated using the developed tool over the same code segment

REFERENCES

- [1] Alan C. Shaw, Deterministic Timing Schema for Parallel Programs, *Technical Report 90-05-06*, Department of Computer Science and Engineering, University of Washington, Seattle, 1990
- [2] K. Nilsen & B. Rygg, Worst-Case Execution Time Analysis on Modern Processor, *ACM SIGPLAN Notices*, Vol. 30, No. 11, pp. 20-30, November 1995
- [3] Y. Steven Li et al., Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software, *Technical Report*, Department of Electrical Engineering, Princeton University
- [4] C. Healy, M. Harmon & D. Whalley, Integrating the Timing Analysis of Pipelining and Instruction Caching, *Technical Report*, Computer Science Department, Florida State University
- [5] Sung-Soo Lim, C. Yun Park et al., An Accurate Worst Case Timing Analysis for RISC Processors, *IEEE Transactions on Software Engineering*, Vol. 21, No. 7, pp. 593 – 604, July 1995
- [6] P. Sorenson & J. Tremblay, *The Theory and Practice of Compiler Writing*, McGraw-Hill, ISBN 0-07-065161-2, 1987
- [7] C. W. Fraser & T. Proebsting, Detecting Pipeline Structural Hazards Quickly, in *Proc. of the 21th Annual ACM SIGPLAN_SIGACT Symposium on Principles of Programming Languages*, pp. 280-286, January 1994