Computation of Worst Case Execution Time

H. Cassé and P. Sainrat

{casse, sainrat} @irit.fr
Université de Toulouse, Institut de Recherche en Informatique de Toulouse (IRIT)

Critical real-time embedded real-time systems needs verification and therefore, WCET computation. In this paper, we present (1) our implementation of WCET computation by static analysis and (2) how it has been adapted to support the Leon 3 architecture, involved in the SOCKET project. Key features of this processor are supported (delayed branches and cache blocks with valid bits).

Introduction

Verification of embedded real-time systems allows to avoid disastrous effects (environmental, economical, etc) caused by a failure. In order to verify that all tasks of real-time application meet their deadline, Worst Case Execution Time (WCET) is needed. This paper shows the static analysis techniques implemented in our tool, OTAWA [2] in order to obtain a safe WCET and how they are applied to the Leon 3, a microprocessor used in the SOCKET[1] project.

Motivation

There are several approaches to compute WCET but only static analysis may be totally safe. Indeed, current approaches based on measurement (full measurement or hybrid methods), even if they are directed by coverage evaluation, cannot cope with long-time effect devices found in modern processors such as cache effects, branch prediction and so on.

On the opposite, the static analysis approach relies on a whole model of the hardware in order to prove that the obtained time is a safe overestimation of the WCET. Yet, this approach exhibits also some issues:

- the hardware model and the actual hardware must be proven to match,
- still, some features found in current hardware are hard or even impossible to analyse precisely,
- some forms of programs are hard to support, or drive to an important overestimation.

Nevertheless, as the measurement approaches are not safe enough for critical real-time applications, static analysis remains the single totally safe approach.

At this time, the more used and the more fruitful static analysis method is called Implicit Path Enumeration Technique (IPET). In the next sections, we present this method and its adaptation to the Leon 3 processor.

IPET Basics

IPET represents the WCET computation as the maximization of an Integer Linear Programming (ILP) problem. The program is represented as a Control Flow Graph (CFG) and the function to optimize represents the WCET value: $WCET = \sum t_i x_i$ where t_i and x_i, are, respectively, time and execution count of Basic Block (BB) *i* in the WCET path.

x_i variables are subject to a bunch of constraints ensuring a bound for the WCET. Basic constraints include flow constraints that describe the control flow inside the CFG. They state that (1) the entry BB is executed once, $x_{entry} = 1$ and (2) the number of times a BB is entered is equal to the number of times it is left:

$$x_i = \sum_{j \in pred(i)} e_{j,i} = \sum_{j \in succ(i)} e_{i,j}$$

Finally, as the CFG may contain loops, we need to bound the number of times a loop iterates. So, we have to exhibit *N*, the maximum number of times a loop (of header BB *h*) iterates[2] each time it is started:

$$\sum_{(i,h) \in back(h)} e_{i,h} \leq N \times \sum_{(h,i) \notin back(h)} e_{i,h}$$

To achieve this goal, IPET is divided into 3 steps. Firstly, the path analysis analyses the control flow from the binary form of the executable to obtain the CFG. Secondly, it uses the hardware model to compute the BB timings while the last step builds the ILP system and solves it.

Path analysis

The path analysis extracts the CFG from the binary executable. This step requires the understanding of the instructions of the program and is therefore architecture dependent. The analyser may be written by hand or automatically generated as in OTAWA.

We used the simulator generator developed at IRIT, (GLISS2) to generate an instruction decoder from the Leon 3 ISA described using the Architecture Description Language SimNML. In addition, other information items about instructions such as their kind, branch targets and used registers may be generated almost automatically.

Building the CFG needed also to take into account a particularity of the Leon 3, the delayed branches. According to a flag in the instruction word, the instruction following a branch may be executed always or only when the branch is taken. Moreover, although the delayed instruction is not executed, it still causes a cache access and an empty slot allocation in the pipeline. To conform to this behaviour, a transformation is performed on the

1 SOCKET project is supported by FUI.

2 *back(h)* is the set of back edges of loop header *h*.

CFG to group back the instruction following a branch. In some CFG configurations, it may be a little bit harder (for example if the delayed instruction is the target of another branch.

The final phase in path analysis is the computation of the loop bounds. This may be automatically performed on most of loops with the oRange [3] tool while unresolved loop bounds may also be provided by hand.

Global Effects Analysis

The timing analysis is usually divided into two steps. First, static analyses are performed to predict, as precisely as possible, the behaviour of global effect devices such as caches, branch predictor, buses, etc. Then the actual BB timing is computed.

In the case of Leon 3, we have to take into account the instruction cache, the data cache and different timings of the memory. The latter feature was already supported as different memories (static RAM, dynamic RAM or ROM with different timings) are mapped at different address ranges.

The instruction cache has 32-byte blocks and is 4-way associative. Its Least Recently Used (RLU) replacement is usually well supported by static analysis (and thus by Otawa) and causes very few overestimation. LRU assigns ages to blocks and augments them when older blocks are accessed. The last accessed block gets always the youngest age.

The computation method [4] we are using is based on abstract interpretation and requires two analyses. The first one, called MUST, estimates the oldest possible age of each block while the second one, MAY, estimates the youngest one. The analysis evaluates the block ages along the CFG path and the LRU age update is applied at each BB as is but, when two execution paths are joined, the MUST analysis takes the maximum of ages while the MAY analysis takes the minimum.

In fact, the Leon 3 instruction cache implements the LRU approach but with an extension: when a new block is loaded, not the whole block but only the first word and a few following ones are got from memory and they are marked with a valid bit. Hopefully, this particularity is easily supported in the abstract interpretation. With the age of a block, we store also a mask representing the valid bits. Valid bits are updated according to the loaded words and the only difference is in the join: for the MUST, we combine masks with an AND (the worst situation arises when less valid bits are activated) while, for the MAY, an OR operation is used.

In the end, the results of both analyses are used to categorize the behaviour of the accessed instructions. Before its execution, if a block is in the cache and is not older than the associativity, the MUST analysis considers it as always in the cache and it is denoted as Always Hit. If the MAY analysis shows it is out of the cache, it is marked as Always Miss. Otherwise, its category is Not Classified: we cannot do any hypothesis on its behaviour. In the average, the rate of Not Classified is under 10% of cache accesses and thus induces very few overestimation.

There is another category, called Persistence [1], obtained from a Persistent analysis: a melt of MAY and MUST analyses. This analysis is used to differentiate cache behaviour between the first iteration and other iterations. This analysis has been also extended to the valid mask. All these categories are finally used together to count the number of misses by adding constraints to the ILP system.

Pipeline Analysis

In this last step, the execution time t_i of BBi is determined. The execution graph [5] approach is used: a graph is built and models the traversal of instructions in the pipeline stages by nodes. The edges model the sequence, the dependencies and the resource usage of instructions. Each node is assigned a date and the latest date gives the execution time of the BB.

Yet, in our analysis, we may have nodes with different possible latencies. Persistent or Not Classified memory induce two times. One solution consists in obtaining a unique time by getting the maximum of the different execution times according to the node latency possibilities. This causes a big overestimation.

Another solution consists in exhibiting as many times as the number of combinations of different latencies. It requires a lot of different variables for the different ways a block executes. It gets the maximum precision but at the cost of an important increase of computation time.

In the middle way, we chose to be accurate for some important effects (Persistent category) and to maximize the time for other ones (Not Classified category). In practice, this is a good trade-off between speed and precision. Finally, the maximized function is fixed according to the calculated times and added variables and is solved to obtain the WCET.

Conclusion

We have presented how WCET is computed in our tool and how it has been adapted to the Leon 3 architecture. Particularities such as delayed branches and valid bits in the instruction cache are successfully supported.

In the future, we plan to support the data cache. Current literature provides either incomplete, or poor overestimating methods. The challenge is double: we have to model with precision the data accesses and to analyse data cache with support of valid bits.

References

[1] C. Ballabriga, H.Cassé. *Improving the First-Miss Computation in Set-associative Instruction Caches*. ECRTS 2008.

[2] C. Ballabriga, H. Cassé, C. Rochange, P. Sainrat. *OTAWA: an Open Toolbox for Adaptive WCET Analysis*. SEUS 2010.

[3] M. de Michiel, A. Bonenfant, C. Ballabriga, H. Cassé. *Partial Flow Analysis with oRange*. ISoLA 2010.

[4] C. Ferdinand, F. Martin, R. Wilhelm. *Applying Compiler Techniques to Cache Behavior Prediction*. ACM SIGPLAN 1997.

[5] C. Rochange, P. Sainrat. *A Context-Parameterized Model for Static Analysis of Execution Times*. Transactions on HIPEAC, Springer, 2007.