Impact of automatic gain time identification on tree-based static WCET analysis

Mathieu Avila, Maxime Glaizot, Isabelle Puaut
IRISA, Campus de Beaulieu, 35042 Rennes Cédex, FRANCE
e-mail: puaut@irisa.fr

Abstract

WCET estimates obtained using static analysis methods are getting increasingly pessimistic as the complexity of hardware and software increases. The difference between the WCET of one task (estimated off-line) and its actual execution time (only known on-line) is known as gain time. *Identifying gain time as soon as possible is important because it increases the number of tasks that can be accepted dynamically. While some research has already been undertaken for the identification of gain time, few work has considered the impact of gain time identification and reclaiming on static WCET analysis methods. This is the objective of this paper, in which we introduce three classes of methods for gain time identification, and discuss their impact on tree-based static WCET analysis methods.*

1 Motivations for automatic gain time identification

Most scheduling algorithms for hard real-time tasks assume that the WCET estimation of each task is known. A number of dynamic scheduling algorithms have also been proposed to dynamically accept soft real-time tasks when spare capacity is left by hard-real-time tasks. Spare capacity is either *extra time* (time known to be left by the hard real-time tasks during the design phase) or *gain time* (spare time appearing at run-time when hard real-time tasks execute in less than their WCET).

Static WCET analysis techniques return an upper bound on the execution time of a task on a given hardware, based on its source code. Having an upper bound on all possible execution times (safety) is of prime importance in hard real-time systems to have confidence in the schedulability analysis methods. But despite the important progress made in static analysis methods, safety comes at the cost of pessimistic WCET estimations. Two sources of pessimism can be identified: (i) analysis of the execution paths, or *high level analysis* (when it is not known statically which path will be executed, the longest path is selected), (ii) *low-level analysis* (when the execution time of an instruction is not known *a priori* due to the use of complex processors with performance enhancing features such as caching or branch prediction, the most pessimistic execution time is selected). As the complexity of software and hardware increases, the degree of pessimism of WCET estimates also increases. In such situations, identifying and reclaiming gain time is getting increasingly important. In this paper we concentrate on the estimation of gain time, and not on its reclaiming.

In our opinion, the methods for gain time identification should have the following properties:

- *Early detection*. The presence of gain time should be detected before the tasks finish their execution. The sooner the gain time is detected, the earlier new tasks can be dynamically accepted.
- *Predictable cost*. Early identifying gain time requires to monitor the progression of the tasks, which has a cost in terms of execution time. This cost has to be predictable and the designer should have means to control the cost of gain time identification.
- *High efficiency.* All gain time should be detected, should it come from the low-level or the high-level sources of pessimism of static WCET analysis.
- *Transparency.* No support (or very low support) from the designer should be required.
- *Predictable and low memory requirements.*

Several techniques have been proposed for gain time identification. [3] consists in measuring the execution time of tasks between so-called *gain points* using specific hardware, the gain points being placed by the programmer. A software evolution of [3] is presented in [2]. In this proposal, the gain points are automatically determined, but the target language is very simple. Both [3] and [2] identify all gain times because they use measurements to monitor the tasks progress. Other methods, that only identify gain time coming from the pessimistic identification of worst-case execution paths, have been proposed in [1] and [4] (the latter tackles object-oriented hard real-time programs). The principle of these two methods is to know statically the WCET of the different paths in the program. Then, each time a path decision is taken, the gain time can be estimated, but

because there are no measures of the actual execution times, the *low-level* analysis pessimism will not be identified.

In the following, we briefly propose three classes of techniques aiming at reaching all the above-identified desirable properties and study the support that static WCET analysis should provide in order for these techniques to be implemented.

2 Three methods of gain time identification and their impact on WCET analysis

All three methods use some general principles. Instrumentation code is inserted in the tasks at specific points called gain points (GP) in which the time actually consumed by the task is measured. Measurements are used to identify *all* sources of pessimism of WCET analysis. All three methods identify gain time on-line by subtracting the measured execution time of *segments* of the task code from the WCET of the same segments. The methods differ by the rules governing GP placement and the definition of a segment. Our discussion hereafter concentrates on the impact of gain time identification on tree-based WCET analysis tools.

A simple example is used hereafter to illustrate the pros and cons of each method for GP placement, as well as their impact on WCET analysis. The source code of the example is presented in figure 1. Two important things can be noticed about this code: (i) the maximum number of iterations of the loop (three, as indicated in the annotation [3] in the source code) may be overestimated (the loop may execute once or twice only); (ii) the most time-consuming execution path within the loop is the "else" path, although the "then" path can be actually executed too.

```
int i;
int j;
for (i=0;i<N;i++) { [3]
  /* Known to iterate at most 3 times */
  if (j==1) {
    j=3;
  }
  else {
    j=j+i;
  }
 }
```

Figure 1. Source code sample

We can extract two data structures from this source code: the program control flow graph (left part of figure 2) and its syntax tree (right part of figure 2). The latter data structure is used in so-called *tree-based* WCET analysis tools to compute the WCET of a piece of code through a bottom-up traversal of its syntax tree.

Figure 3 depicts for this sample program the differences that may exist between the off-line and a given on-line timeline. It shows that the actual execution time is lower than the

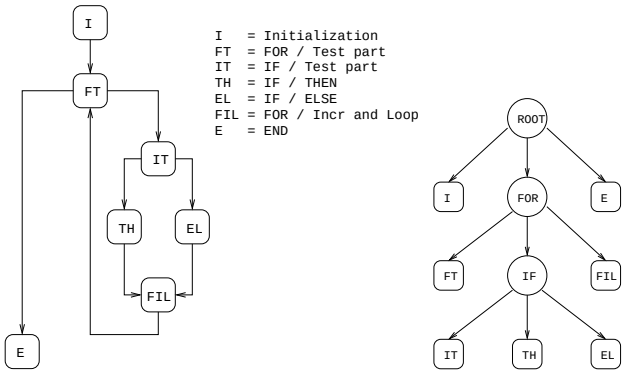

Figure 2. Control-flow graph (left) and syntax tree (right)

WCET. More precisely, it identifies the different sources of gain time: (i) gain time coming from the pessimism of low-level analysis (in the figure, the actual execution time of basic block I is lower than its worst-case counterpart identified off-line); (ii) gain time due to the pessimistic evaluation of the worst-case execution path (e.g. the loop iterates two times instead of three at worst; within the loop the "else" branch – basic block EL – may be executed whereas it is not the longest branch).

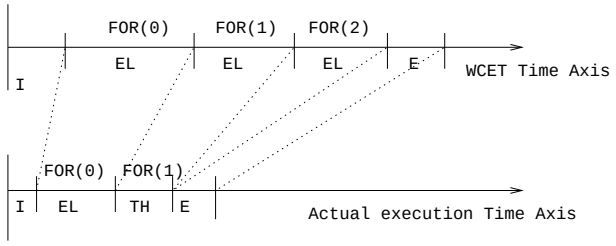

Figure 3. Off-line and on-line timelines

2.1 Segment-based method

This method puts almost no constraint on the locations of GPs. It reasons on *segments* defined as intervals between successive GPs in the task control flow. This method is flexible since the length and location of segments can be tailored so as to find an appropriate tradeoff between cost and earliness of gain time identification. However, the off-line overheads of the method are high. Indeed, the static WCET analyzer has to generate partial WCETs for any pair of points that can be consecutive in the task control flow, leading to a potentially high number of partial WCETs to be computed.

For instance, if three GPs are placed in our sample code as shown in figure 4, six partial WCETs must be computed to cover all possible paths between successive GPs in the control flow graph. Furthermore, the integration of the computation of WCETs of segments is not natural in tree-based WCET analyzers because of the mismatch between the location of GPs and the data structures used by such analyzers.

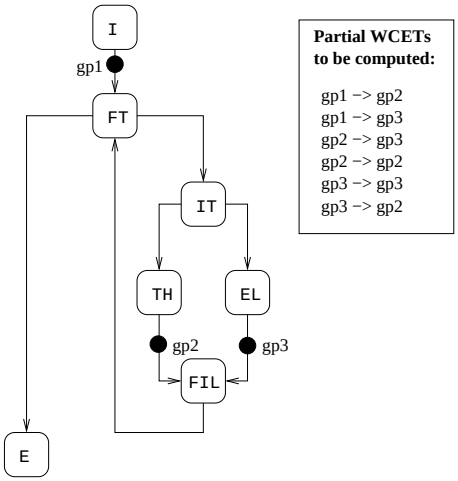

Figure 4. Example of GP locations in the segment-based GP placement method

The large number of partial WCETs, in addition to increasing the complexity of the computation of partial WCETs, also increases the complexity of the on-line part of gain time identification, since all partial WCETs have to be accessible on-line. Another problem of this method is that, when a GP is placed in a loop body, it only allows to identify gain time within the loop but is unable to identify the gain time arising when the loop iterates less than expected.

2.2 Structural method

This method restricts the locations of GPs to the control structures in the syntax tree like loops, conditional constructs (segments do not cross control structures boundaries as in the first method). A control structure in which it is interesting to reclaim gain time is enclosed by a pair of GPs (immediately before and after the control structure). Partial WCETs are then needed for all "instrumented" control structures. Figure 5 shows on our example all possible pairs of GPs (e.g. GPs ga_b and ga_e to define a segment corresponding to the loop).

One can note that the number of partial WCETs to be computed off-line tend to be less numerous than with the

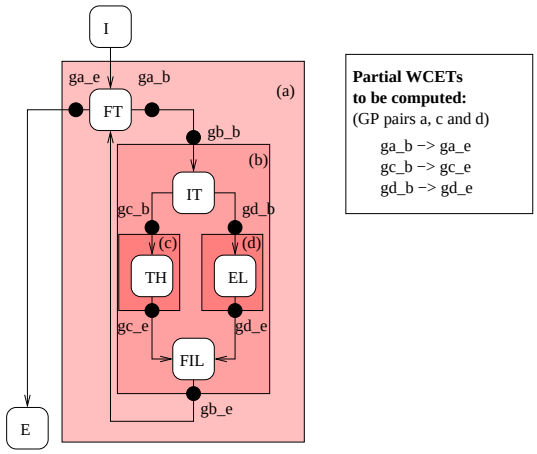

Figure 5. Possible GP locations in the structural GP placement method

first method. On our example, if the TH, EL and FOR control structures are instrumented (pairs of GPs a, c and d, which is roughly equivalent to placing GPs $gp1$, $gp2$ and $gp3$ in the first method), only three partial WCETs have to be computed, compared to six in the first method. The off-line computation of the partial WCET of segments is rather straightforward, as tree-based tools actually compute a WCET for each level of the syntax-tree. However, the method is less flexible than the first one because of the imposed restrictions on the locations of GPs.

2.3 Path based method

This last method is an hybrid one which is halfway between the first two ones. As in the segment-based method, no restriction is put on the locations of GPs, thus ensuring the flexibility of the method (ability to find an appropriate trade-off between cost and earliness of gain time identification). But instead of defining segments as intervals between successive GPs in the task control flow, it defines segments as intervals between the *beginning* of execution of the task and the different GPs (see figure 6).

Since a GP can be encountered several times if it is enclosed in a loop (for instance $gp2$ in the figure), several partial WCETs have to be generated depending on the loop counters. Instead of generating all possible WCETs of segments, we propose to represent the WCET as parametric values depending on the loop counters (functions with the loops counters as parameters). These functions should be simple enough to be evaluated on-line, but expressive enough to represent the WCET time elapsed since the beginning of the program.

On the on-line part, the task must keep track of the values

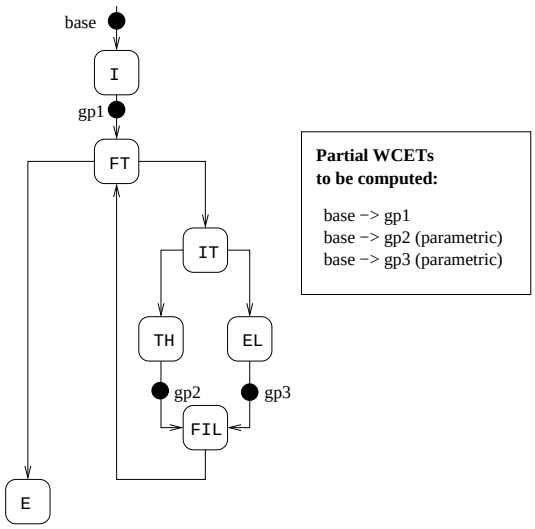

Figure 6. Example of GP locations in the path-based GP placement method

needed by the evaluation functions (essentially loop counters). Each time a GP is encountered, these values, the evaluation function and the elapsed time are stored. The actual computation of gain time (especially the calls to the functions that evaluate the segments' WCETs) can be deferred until gain time is requested by the dynamic scheduler. The impact of this method on tree-based WCET analysis is bigger when GPs are placed inside loops than outside, because then parametric WCET representations must be generated.

2.4 Degree of pessimism of partial WCET estimates

A common issue to be addressed is the degree of safety of WCET estimations of segments (partial WCETs) in order for the estimation of gain time not to be overly optimistic.

This issue is illustrated in figure 7, which depicts gain time identification during the execution of a task made of a sequence of two blocks A and B. Due to the consideration of pipelining effects, the sum of the partial WCETs of A and B (6 time units each in the figure) may exceed the WCET of the sequence A;B (10 time units). Assume that the actual execution of both A and B is 4 time units (the gain time is 2 time units). If the partial WCETs of 6 are used for gain time computation, the gain time is overestimated (4 time units instead of 2), which can cause new tasks to be accepted dynamically whereas too few spare time is available.

More generally, the requirement is that the partial WCETs be *consistent* with the global WCET of the task. Thereby we mean that the value of the WCET of a segment of code is lower or equal to its equivalent in the WCET of

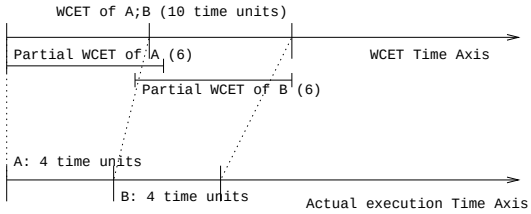

Figure 7. Pessimism of partial WCET estimates

the whole task. Such consistency problems could occur because of low-level analysis in architectures with pipelines (as shown in the example) and in symbolic WCET estimation methods. Possible directions to address this issue would be to provide "optimistic" partial WCETs or to change the WCET computation method for the whole task so that partial WCETs are consistent with the global one.

3 Concluding remarks

Early identification of gain time requires to obtain WCETs of *segments* of the task code instead of considering the code as a whole. In this paper, we have proposed three classes of methods for identifying gain time, differing by their definition of segments. We have further examined their impact on tree-based WCET analysis methods.

Except for the second proposed method, which can be integrated naturally in tree-based WCET analyzers, we are convinced that the need to compute partial WCETs has a non negligible impact on the structure of the WCET analysis tools. Earliness of gain time identification comes at the price of a further increase in complexity of WCET analysis techniques.

References

[1] N. C. Audsley, R. I. Davis, and A. Burns. Mechanisms for enhancing the flexibility and utility of hard real-time systems. In *IEEE Real-time systems symposium*, pages 12–21, December 1994.

[2] P. Gopinath and R. Gupta. Applying compiler techniques to scheduling in real-time systems, 1990. Philips Laboratories.

[3] D. Haban and K. Shin. Application of real-time monitoring to scheduling tasks whith random execution times. In *IEEE Transaction on software enginneering*, December 1990.

[4] E. Y.-S. Hu, A. Wellings, and G. Bernat. A novel gain time reclaiming framework integrating wcet abalysis for object-oriented real-time systems. In *Second workshop on WCET analysis*, June 2002.