TU/e technische universiteit eindhoven

Sharper WCET Upper Bounds using Automatically Detected Scenarios

Stefan Valentin Gheorghita, Sander Stuijk, Twan Basten, Henk Corporaal

ES Reports
ISSN 1574-9517

ESR-2005-04
21 March 2005

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems

Get the habit of WCET analysis - it will, in time, enable synthesis for real-time to become your habit of mind.

http://www.es.ele.tue.nl/esreports
esreports@es.ele.tue.nl

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems
PO Box 513
NL-5600 MB Eindhoven
The Netherlands

Sharper WCET Upper Bounds using Automatically Detected Scenarios

Stefan Valentin Gheorghita, Sander Stuijk, Twan Basten, Henk Corporaal
Eindhoven University of Technology, PO Box 513,
5600MB, Eindhoven, The Netherlands
{s.v.gheorghita, s.stuijk, a.a.basten, h.corporaal}@tue.nl

21 March 2005

Abstract

Modern embedded applications usually have real-time constraints and they are implemented using heterogeneous multiprocessor systems-on-chip. Dimensioning a system requires accurate estimations of the worst-case execution time (WCET). Overestimation leads to over-dimensioning. This paper introduces a method for automatic discovery of scenarios that incorporate correlation between different parts of applications. It is based on the application parameters with a large impact on the execution time. We show on a benchmark that using scenarios the estimated WCET may be reduced with 16%.

Keywords: WCET, Real-Time, Application Scenarios, Compilers, Static Timing Analysis

1 Introduction

Embedded systems usually consist of processors that execute domain-specific programs. Many of their functionalities (tasks) are implemented in software, which is running on one or multiple generic processors, leaving only the high performance functions implemented in hardware. Typical examples of embedded systems include TV sets, cellular phones and automotive engine controller units. As many of these systems have real-time constraints, to dimension them, accurate estimations of the worst-case and best-case execution time (WCET and BCET) of their tasks are required. More precisely, it is required to tightly bound the execution times of all feasible paths of the program. If the minimum and maximum duration of all these executions are denoted by T_{min} and T_{max}, the *actual bounds* of a program execution time are given by the interval $[T_{min}, T_{max}]$. The goal of the estimation is to find an interval $[t_{min}, t_{max}]$ that tightly encloses the actual bounds (see Fig. 1) [LM98]. This interval represents the estimated bounds of the program execution time, and respectively, t_{min} and t_{max} are the estimated BCET and WCET of the program. Since estimation of WCET and of BCET are very similar to each other and the techniques developed for one can be easily adapted for the other, we focus only on WCET.

To determine the estimated WCET of a program, all the factors that affect the program execution time must be considered: the feasible execution paths and the execution time of each instruction in each path. In this paper, we discuss the first factor, which is platform independent. The second one depends on architecture parameters, like number of cycles per instruction type, memory hierarchy and pipelining and it was extensively researched in the last years (e.g. [WM05, BR05, ZWHM04]). A micro-architecture model is needed to analyze it.

One of the problems in finding the estimated WCET of a program is that the longest execution path is unknown in many cases. If it can be determined, the problem is trivial to solve. Simulation of all execution paths is clearly impractical as their number is usually exponential in the program size. The

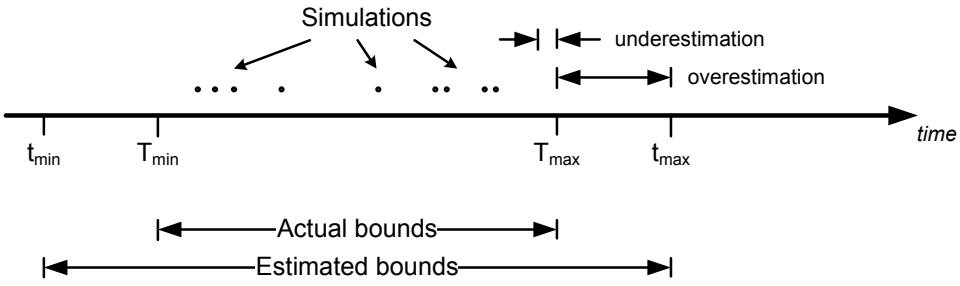


Figure 1: Estimated vs. actual bounds.

results from the simulation of a subset of feasible execution paths are very likely to fall strictly within the actual bounds of the program, even if the subset was very carefully selected ([BCP02, BCP03, CG04]). This leads to an underestimation of the bounds (see fig. 1). With some extensions, simulation-based analysis can be used for designing soft real-time systems, but it cannot be tolerated in analysis of hard real-time systems.

To avoid the explosion in the number of execution paths, many approaches use a timing schema as the basis for estimating the WCET. Such a timing schema is attributed to certain high-level language constructs, and it is essentially a set of formulas for computing an upper bound on their execution time [Sha89]. Nevertheless, the timing schema cannot be directly applied to programs because not all the needed information is contained in their source code. One of the reasons is that programs contain non-manifest loops. In many cases, the number of iterations of these loops cannot be determined automatically as they may depend on input parameters. With only a few exceptions (e.g. [RW94, Bli94]), all the existing techniques rely on the programmer to provide loop bounds.

Although by using a timing schema the explosion in the number of paths is avoided, often a large number of infeasible paths is considered in WCET estimation, potentially introducing a big overestimation (see fig. 1). This is because timing schema does not differentiate between infeasible and feasible paths, and the estimated WCET may appear because one of the infeasible paths. In [LM98, MACT89, PK89] authors attempt to solve this problem attaching an execution counter to each statement in the source code. It represents the maximum number of execution times for the statement. The counters are not enough in the case of large applications, where parts of them tend to relate to each other.

In this paper, we propose an automatic method for reducing the number of infeasible paths considered in a timing schema based WCET estimation. We use static analysis to discover the correlations between parts of an application. These correlations are used to partition the application in different, so called, *scenarios*. The application estimated WCET is computed as the maximum estimated WCET of these scenarios. We are not the first ones to consider the concept of scenarios but, to the best of our knowledge, there are no methods that can automatically determine scenarios. Our method for reducing WCET overestimation is platform independent and can be applied on top of all existing WCET estimation methods based on timing schema. To prove its efficiency, we implemented it in a tool that was tested on two benchmarks (MP3 and H.263 decoders). In one case, the estimated WCET was reduced with 16%. In the other case, the estimated application WCET could not be reduced, but a reduction of 39% percent of a typical scenario was obtained. Both results provide valuable information for dimensioning the final system implementation.

The paper is organized as follows. Section 2 compares our work with related approaches. Section 3 describes how a timing schema works. Section 4 shows how scenarios can be used to estimate more accurately the WCET for an application. In section 5, we introduce an algorithm suitable for scenario discovery. The evaluation of our approach on two test cases is presented in section 6. Our conclusions and future plans are presented in section 7.

2 Related work

In this section we compare our work with different approaches for WCET overestimation reduction. Other areas where scenarios are already used are also presented.

Many approaches to reduce the overestimation of WCET have been studied. Some of them use C [PK89] or assembly [MACT89] level user annotations to describe the maximum number of executions for different statements. On top of these approaches, in [Par92], a mechanism that allows a user to specify the correlations between parts of the application is added. However, all of these approaches require additional information added into the source code, which is what we avoid in our work.

Another way to control the WCET overestimation is parametric WCET. There are different methods to compute it, based on timing schema [CB02] and path enumeration [BB00]. Manual annotations for constraints on loop counters and infeasible paths are needed. As an extension, in [VHMW01], an iterative method to compute parametric WCET bounds for simple loops has also been suggested. However, even for a fully automatic approach, which can find both loop bounds and infeasible paths [Lis03], there is a huge explosion in the number of parameters. It is very hard to identify the most important parameters only by the variables' name. In our approach, we introduce a method which helps in identifying the parameters that influence the estimated WCET the most.

The scenario concept was first used [YMW+03] to capture the data-dependent dynamic behavior inside a thread, in order to better schedule a multi-thread application on a heterogenous multi-processor architecture. In [PBV+04], the authors try to apply different source-to-source transformations to each discovered scenario to improve the overall application performance. To the best of our knowledge, we are the first ones to present a technique for automatically detecting scenarios and the first ones to use scenarios to reduce WCET overestimation.

3 A Simple Timing Schema

Before getting into the depth of our method, we first present how a timing schema works. All existing timing schema are based on Shaw's schema [Sha89] and they are applied on the abstract syntax tree (AST) of the program. The AST leafs are the basic blocks[1] of the program and the inner node corresponds to syntactic composition of the blocks. Three types of composition exist: sequential composition, conditional composition and iterative composition.

A timing schema is a set of the rules that allow to determine the WCET of a program segment as a function of the execution time of its components. Each rule is associated with a type of node in the AST. B, B_1, B_2 are blocks of statements (not mandatory basic blocks) and n is the number of loop iterations:

$$\text{WCET}(B) = \text{an integer value, if B is a basic block} \tag{1}$$

$$\text{WCET}(B_1; B_2) = \text{WCET}(B_1) + \text{WCET}(B_2) \tag{2}$$

$$\text{WCET}(\textit{if } B \textit{ then } B_1 \textit{ else } B_2) = \text{WCET}(B) + \textit{max}(\text{WCET}(B_1), \text{WCET}(B_2)) \tag{3}$$

$$\text{WCET}(\textit{while } B \textit{ do } B_1) = (n+1) * \text{WCET}(B) + n * \text{WCET}(B_1) \tag{4}$$

Informally, the WCET of a sequence of two blocks of statements is the sum of their WCETs (eq. 2: *sequential composition*). For an `if-then-else` statement, the WCETs of `then` and `else` branches are compared and the maximum is added to the WCET of the `if` condition (eq. 3: *conditional composition*). For a `while` loop, the WCETs of the loop body and condition are multiplied by the number of iterations, and the condition WCET is added one more time because of the loop exit test (eq. 4: *iterative composition*).

[1] A basic block is a set of instructions that have no control flow instruction except posibly the last one

	source code	Influence coefficient for variable ct
1	`if (ct == 1)`	$IC(ct) = 2 \cdot [max(8 \cdot IC_f(ct), 8 \cdot IC_g(ct)) + abs(8 \cdot WCET(f) - 8 \cdot WCET(g))]$
2	`  for (y=0; y<8; y++)`	$IC(ct) = 8 \cdot IC_f(ct)$
3	`    f(&b[x*8+y]);`	$IC(ct) = IC_f(ct)$
4	`else /* ct!=1 */`	
5	`  for (y=7; y>-1; y--)`	$IC(ct) = 8 \cdot IC_g(ct)$
6	`    g(&b[x*8+y]);`	$IC(ct) = IC_g(ct)$
7	`if (ct != 1)`	$IC(ct) = max(8 \cdot IC_f(ct), 8 \cdot IC_g(ct)) + abs(8 \cdot WCET(f) - 8 \cdot WCET(g))$
8	`  for (y=0; y<8; y++)`	$IC(ct) = 8 \cdot IC_f(ct)$
9	`    f(&b[x*8+y]);`	$IC(ct) = IC_f(ct)$
10	`else /* ct=1 */`	
11	`  for (y=7; y>-1; y--)`	$IC(ct) = 8 \cdot IC_g(ct)$
12	`    g(&b[x*8+y]);`	$IC(ct) = IC_g(ct)$

Figure 2: Educational example

These equations cover the entire ANSI C grammar (which is the most used programming language for the embedded systems area), as all other control constructs can be rewritten to use them. Simple control flow statements, like `for`, `switch`, `goto`, can be directly transformed to `while` and `if` statements. A few constructs are hard to handle: recursive functions (unknown depth), back jumps (hidden loops) and dynamic function calls. The first two can be transformed in loops using different mechanisms [Bli01, DB73]. Even though the dynamic function call seems to be a fundamental problem, it is solvable in embedded software, as usually all possible called functions are known at design time.

4 Sharper upper bounds using scenarios

In order to refine the estimation of the WCET, we divide the application in a set of scenarios. *A scenario is defined as the application behavior for a specific type of input data*. The set of scenarios must cover all possible input data. An example of a scenario for the H.263 decoder [Rij95] is the application behavior for any frame of type *P*. Together with scenarios for frame types *I* and *B*, they cover all possible input data.

For each scenario, those parts of the application source that are never executed, are identified and removed, and the WCET is estimated using e.g., Shaw's schema. Preserving the conservativeness of estimation, the WCET for the entire application is then defined via the following equation:

$$WCET(app) = \max_{S \in Scenarios}(WCET(S)) \tag{5}$$

To emphasize the possible benefit of scenarios in WCET computation, fig. 2 presents an educational example. Notice that only the order in which the functions f and g are executed differs, based on the value of ct. Using only a timing schema, the estimated WCET is

$$2 \cdot 8 \cdot \max(WCET(f), WCET(g)) + const. \tag{6}$$

where $const$ represents the loop overhead. Considering two scenarios defined on different values of variable ct (the first scenario for $ct = 1$, and the second one for $ct \neq 1$), the WCET is

$$8 \cdot (WCET(f) + WCET(g)) + const \tag{7}$$

If the WCET of f and g are very different, then the use of scenarios seriously reduces the overestimation compared to the approach based only on timing schema.

Besides correlations between different parts of the code, as illustrated above, scenarios may also incorporate a different number of loop iterations. For example, in one scenario, a loop iterates for maximally 10 times, and in another scenario the same loop iterates for only maximally 5 times. If the WCET for this code is computed without considering scenarios, the maximum number of iterations must be considered 10.

```
1   if (ct != 0) ct = 1;
2   for (y=0; y < 8 * (ct+1); y++)
3     if (ct == 1)
4        f(&b[x*8+y]);
5     else
6        g(&b[x*8+y]);
7   for (y=0; y < 8 * (2-ct); y++)
8     if (ct != 1)
9        f(&b[x*8+y]);
10    else
11       g(&b[x*8+y]);
```

Figure 3: Extended example.

An extension of the previous example, presented in fig. 3, emphasizes the effect of different numbers of iterations in different scenarios. Notice that only the order in which the 16 calls to function f and the 8 calls to g are executed differs, based on the value of ct. Its estimated WCET based only on a timing schema is:

$$2 \cdot 16 \cdot \max(WCET(f), WCET(g)) + const. \tag{8}$$

The one computed based on the scenario approach is:

$$8 \cdot WCET(g) + 16 \cdot WCET(f) + const. \tag{9}$$

Both, correlations between different parts of the source code and the number of loop iterations, are considered in our algorithm for detecting scenarios.

5 Automatic scenario detection

Our approach is based on static analysis of the application source code and it consists of five steps:

1. Identify the parameters that could potentially have an impact on the application execution time.
2. Compute the maximum possible impact of these parameters on the WCET.
3. Partition the application in scenarios considering these parameters together with their impact.
4. Generate source code for each scenario and estimate their WCET using a timing schema.
5. Compute the application WCET using equation 5.

1: The first step is based on the fact that there are usually a few parameters that have a significant impact on the application execution time (e.g. in a video decoder: image size and type). Many of these parameters are read at the beginning of the execution and remain constant for the rest of it. Moreover, usually, there is only a small set of possible values for them (e.g. for an H.263 decoder, there is one variable which specifies the image type, with three possible values: *I*, *B* or *P*). In a C source code, these parameters usually appear as variables or fields of structures of integer or enumeration type. There is only one statement for each parameter in the program that changes its value (often it is set based on the program input data).

2: To identify from these parameters the ones that might influence the WCET the most, we first compute the application WCET using a timing schema. Then, the possible impact on the WCET of each variable or structure field (denoted by v) that respects the above observations, is computed in the form of its so-called *influence coefficient* ($IC(v)$). The $IC(v)$ represents the maximum possible variation (in cycles) caused by the different values of v on the WCET of the application.

To this end, the abstract syntax tree (AST) of the program is traversed in a post-order manner and $IC(v)$ is computed in each statement s, as a sum of its contribution and the maximum of $IC(v)$ computed for all its successors in the program (eq. 10).

$$IC_s(v) = contribution(s) + \max_{x \in successors(s)}(IC_x(v)). \tag{10}$$

A statement may have zero, one or multiple successors. The last statement of a function or a loop body has no successors (e.g. fig. 2, line 12). The control statements may have multiple possible next statements (e.g., for if, the first statement of the $then$ and the $else$ branches, as appear in the max factor of the $IC(ct)$ from fig. 2, line 7). The rest of the statements have only one successor. For each statement, its successors are always already processed, due to the post-order traversal of the AST.

The contribution of a statement to $IC(v)$ quantifies the maximum variation in execution time of the statement as caused by different values of v. Depending on the statement type, the contribution is equal to:

- **function call**: the $IC(v)$ computed in the first statement of the called function f (e.g. fig. 2, line 12). If v is a parameter of the function call, a renaming is done for computing the $IC(v)$ inside the function.
- **loop**: the $IC(v)$ computed in the first statement of the loop body multiplied by the maximum number of iterations (e.g. fig. 2, line 11).
- **if**: if v appears in the if condition, and it is compared to a constant,

  $$abs(WCET(then) - WCET(else)), \tag{11}$$

  or else 0 (e.g. abs factor of $IC(ct)$ from fig. 2, line 7).
- **switch**: if v appears in the $switch$ condition,

  $$\max_{B \in Branches}(WCET(B)) - \min_{B \in Branches}(WCET(B)). \tag{12}$$

  or else 0

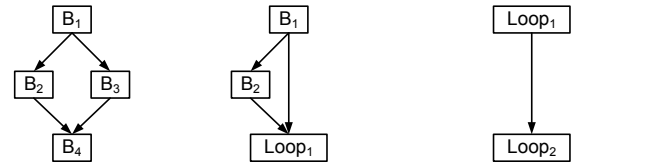
Equations 11 and 12 represent the only points where values different from zero are injected in our algorithm. Scenarios can be used for different purposes than WCET estimation, such as memory usage estimation. The same method can be applied but with different formulas for equations 11 and 12.

In the case of loops, in order not to overestimate the $IC(v)$ value, instead of multiplying by the maximum number of iterations, it might be better to use the difference between maximum and average number of iterations. This is not always possible, as to compute the average number of loop iterations, a large set of carefully selected input data must be simulated [MLCO04].

3: The first statement of the program will yield the ICs computed for each possible parameter. To avoid an explosion in the number of scenarios, different criteria to select which parameters are used to define scenarios might be used. The selection may incorporate knowledge about the application combined with heuristics based on the computed ICs. An example heuristic may be to select only those parameters with very big IC values. However, the algorithm used in the selection stage, depends on what the scenarios are used for and this can be an open point for a design-space exploration approach.

For each selected parameter, the constants it is compared to in the source code are collected. These constants, together with the comparison operators, are used to split the set of possible values of the parameter in subsets. A scenario is characterized in the end by possible values of the selected parameters.

Figure 2 shows how the (IC) for variable ct is computed for the first example presented in section 4. As it could already be seen in the source code, we can automatically detect that, based on ct, two scenarios are defined: $ct = 1$ and $ct \neq 1$.



$S_1 : B_1, B_2, B_4$ $S_2 : B_1, B_3, B_4$ (a)

$S_1 : B_1, B_2, Loop_1(x)$ $S_2 : B_1, Loop_1(y)$ (b)

$S_1 : Loop_1(s), Loop_2(t)$ $S_2 : Loop_1(u), Loop_2(v)$ (c)

Figure 4: Examples of good scenario selection ($x < y$, $s < u$, $t > v$ are the number of iterations for loops).

At this point, we can refine our notion of a scenario as a part of the application source code with a specified maximum number of loops iterations[2]. The scenario's set of execution paths consists of all possible execution paths through it. In order to potentially obtain a reduction for estimated WCET using scenarios, a scenario should not include all application execution paths. To avoid an explosion in the number of generated and evaluated scenarios in step 4 of our algorithm, all scenarios that have the set of execution paths included in another scenario's set must be ignored. To fulfill these two conditions, each pair of selected scenarios must fall in one of the following cases:

- there must be at least one part of the source code which is executed in the first one and not in the second one, and vice versa (e.g. scenarios S_1 and S_2 from fig. 4(a)).
- one of the scenarios includes a part of the code which is not included in the other one and it executes a loop for a smaller number of iterations (e.g. scenarios from fig. 4(b)).
- they have different maximum numbers of iterations for two loops and for one loop the first scenario must iterate more than the second scenario, and vice versa for the second loop (e.g. scenarios from fig. 4(c)).

4: For each scenario a modified version of the *unreachable code elimination compiler phase* is used to remove the code that is never executed because of specific parameters values. The estimated WCET is computed based on a timing schema, like Shaw's one.

5: In the end, equation 5 is used to obtain the application WCET.

6 Evaluation

We tested our method on two multimedia applications, an MP3 decoder [Lag01] and an H.263 decoder [Rij95], that supports only *I* and *P* frames. For the first case, we obtained an improvement of 16% over the estimated application WCET computed without taking scenarios into consideration. The second benchmark does not show a reduction in the overall WCET estimation, but an improvement of 39% for one of the scenarios was obtained.

For our experiments we used a microarchitecture model similar to ARM7TDMI [arm]. This processor does not have caches and the pipeline effects were considered only inside basic blocks. For computing scenarios WCET we use Shaw's timing schema [Sha89]. To determine the loop bounds we use Rustagi's approach [RW94]. In the case when they cannot be automatically derived, we provided them ourselves.

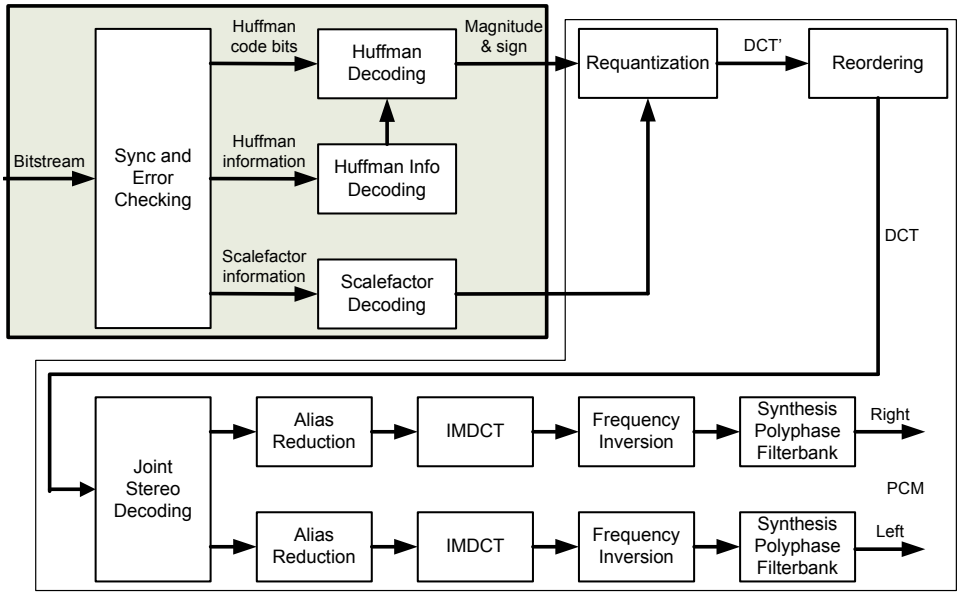


Figure 5: MPEG-1 Layer 3 audio decoder structure

```
for (granule = 1..2)
  for (ch = 1 .. no_channels)
    Requantization()
    Reordering()
  JointStereoDecoding()
  for (ch = 1 .. no_channels)
    AliasReduction()
    IMDCT()
    FrequencyInversion()
    Synthesis()
```

Figure 6: MP3 back-end decoder pseudocode

Table 1: Characterization of back-end kernels

Kernel	Behaviour
Requantization	Different algorithms for short and long blocks.
Reordering	Executes only on short blocks.
AliasReduction	Executes only on long blocks.
IMDCT	Different algorithms for short and long blocks.
FrequencyInversion	Doesn't make difference between long and short blocks.
Synthesis	Doesn't make difference between long and short blocks.

Table 2: Variables' influence coefficients for MP3 Decoder

Variable Name	IC
block_type	1.053.728
mixed_flag	66.128
mode_extension	104.418

6.1 MP3 Decoder

The MPEG-I Layer III [Shl94] decoder is a frame-based algorithm, which transforms the compressed bitstream in normal PCM coded data. A frame consists of 1152 mono or stereo frequency-domain samples, divided into two granules. Each granule consists of 576 frequency components divided into 32 subbands of 18 frequency lines each.

The structure of an MP3 decoder is shown in fig. 5. In the application front-end (the gray box of fig. 5), the Huffman decoder is applied on each received frame. It does irregular accesses to a list of lookup tables, depending on which ones were used for encoding the frame. The application back-end consists of several kernels which use blocks as basic processing units. There are two types of blocks: short blocks which contain 6 frequency lines and long blocks which contain a subband (18 frequency lines). The standard specifies that each channel from a granule can be encoded in one of three possible ways: only with short blocks (96), only with long blocks (32) or mixed (2 long blocks for the lowest frequency subbands and 90 short blocks for the rest).

Table 1 shows information about how the kernels behave on different types of blocks. It can be easily observed that the back-end of this application may represent a good candidate for our approach to reduce the estimated WCET. Besides the channel encoding, there are two other parameters which can influence the execution time of the application: the number of audio channels (1 or 2) and the audio mode (mono, dual mono, stereo or joint-stereo). The back-end pseudo-code (fig. 6) shows that the number of channels determines only how many times the same code is executed, and having different scenarios for different numbers of channels will not reduce the overall estimated WCET using our method. In case of the audio mode, it depends on the implementation, but as everything is implemented in only one function, it is very simple for the user to write an optimized code for WCET analysis.

For our experiments we used the implementation provided in [Lag01]. We chose it because it is very close to the standard implementation, it is totally written in C and it contains many algorithmic optimizations, like using Newton's method in the requantization kernel, and fast DCT in matrix operation of the synthesis polyphase filterbank. During our experiment, we assumed that only one type of channel encoding (short, long, mixed) is used for an entire frame. This does not specialize the analyze or change its results, it only reduces the number of possible scenarios in order to make the story easily presentable.

Our tool was run on the MP3 decoder back-end. We first estimated its WCET based only on the timing schema to 2.405.968 cycles and computed the influence coefficient (IC) for all possible parameters. The ones with relevant IC (bigger than 100 cycles) were selected to be used to define scenarios (see table

[2]These numbers may be smaller than the ones considered for the same loops in the WCET analysis based only on timing schema.

Table 3: MP3 Decoder scenarios ($WCET = 2.405.968$)

scenario number	block type	mixed flag	mode extension	WCET (cycles)	reduction
1	2 (short)	0	*	2.079.124	16%
2	2 (short)	1	*	1.984.816	18%
3	$\neq$ 2 (long)	*	*	1.666.804	31%

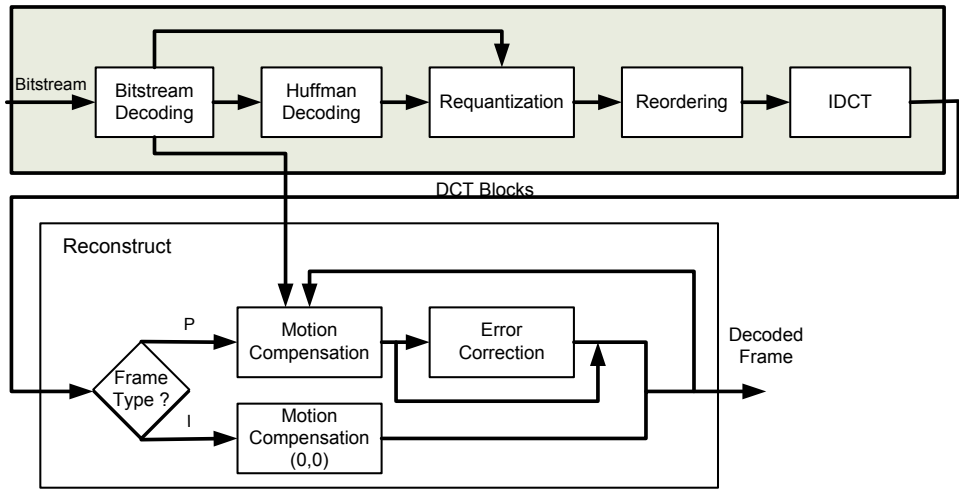


Figure 7: H.263 video decoder structure

2 for their names and influence coefficients). The first two represent together the channel encoding type, and the third one represents the audio mode.

Splitting the application, based on them, we obtain three possible scenarios (see table 3). The first two scenarios represent the short and the mixed encoding type. In the last one, the mixed_flag does not have any impact, as block_type is long. The different values for mode_extension do not introduce different scenarios in the application partitioning, as all of them are automatically eliminated based on the rules presented in section 5. The estimated WCET for each discovered scenario is smaller than the application WCET previously estimated. Using equation 5, the application WCET is reduced with 16%.

If the rules from section 5 are relaxed, our tool generates 12 scenarios based on the possible values of block_type (2 values), mixed_flag (2 values) and mode_extension (3 values) variables. As we have expected, the application estimated WCET is not reduced furthermore.

6.2 H.263 Decoder

H.263 [Rij95] is a standard video-conference codec, optimized for low data rates and relatively low motion. The codec was used as a starting point for the development of the MPEG-II codec which is optimized for higher data rates. The structure of an H.263 decoder is depicted in fig. 7. The bitstream decoder splits bitstream into dequantization tables, motion vectors and encoded picture data. The picture data contain all information needed to decode the frame(s) of the movie sequence. A frame consists of blocks, which form the basic data elements in the decoder. A block is passed subsequently from the bitstream decoder through the huffman decoder, requantization, reordering and IDCT. If sufficient blocks are decoded in this path, the frame can be reconstructed. The H.263 decoder we used supports two types of frames: I-frames and P-frames. To decode a P-frame, the reconstruct uses the previous decoded frame and the already decoded blocks. For an I-frame, only the decoded blocks are used. The reconstruct step (see Fig. 7) handles both frame types in different sub-steps. The I-frame reconstruction requires that each decoded block is put at the right position in the frame. The P-frame reconstruction first uses a

Table 4: Variables' influence coefficients for H.263 Decoder

Variable Name	IC
pict_type	536.268
block_count	374.670
fault	2.415
long_vectors	552
newgob	184

Table 5: H.263 Decoder ($WCET = 1.308.538$)

Scenario	**WCET (cycles)**	**Reduction**
$pict\_type = 1(Pframe)$	$1.308.538$	0%
$pict\_type = 0(Iframe)$	794.350	39%

motion vector to retrieve the correct block of pixel data from the previous frame. The resulted pixel data is corrected, if needed, in the error correction step with the pixel data contained in the decoded block (input of the reconstruct block).

The reconstruction of an I-frame and P-frame may seem to be different, which may lead to the idea that a sharper upper-bound can be obtained on the WCET. However, the processing performed for an I-frame is a true subset of the processing done for a P-frame (i.e. no error correction and motion compensation with all motion vectors set to zero). From this we conclude that no sharper upper-bound on the estimated WCET can be obtained using our method, as the decoding of a P-frame will be the slowest situation possible. The experimental results, presented in table 5, confirm this conclusion [3]. Table 4 shows the values of computed influence coefficients for H.263 decoder benchmark, based on a image size of $352x288$ pixels ($22x18$ blocks).

Even if based on scenarios the application estimated WCET is not reduced, the information that scenarios have different WCET may be useful in designing real-time systems (e.g. use the spare time for I-frames to save energy by frequency scaling).

7 Conclusions and future work

In this paper, we have presented an automatic method for detecting sharper upper bounds on the estimated WCET of an application. It can be applied on top of all WCET estimation approaches based on timing schema. It is based on scenarios, which incorporate both the correlations between different parts of the application source code and different numbers of iterations for the same loop. To discover scenarios, we propose an algorithm, based on static analysis of the source code.

Solutions for preventing an explosion in the number of scenarios were proposed. Our method was tested on two applications, an MP3 audio decoder and an H.263 video decoder. For the first benchmark the estimated WCET was reduced with 16%. For the second one, the estimated application WCET could not be reduced, but an improvement of 39% for a typical scenario was obtained. Both results provide valuable information in dimensioning real-time systems.

In the future, we plan to develop methods to dimension real-time systems based on scenarios. We will consider that there are different WCET per scenario together with knowledge about possible sequences of scenarios. We also want to extend our work in detecting scenarios for multi-task applications. We are further investigating ways of detecting scenarios based on profiling. Even if this method can not offer

[3] Besides to reduce the stream size, another reason for introducing P-frame type in the H.263 standard is that the average time to decode it is smaller than the time needed for an I-frame. Contrary, from our experiments we observe that the WCET for decoding a P-frame is larger than the one for decoding an I-frame. From this we can conclude that if we want a hard real-time system for H.263 decoder, maybe is better to not support P-frames in our input streams

the conservativeness needed for estimating WCET[4], it is good enough for soft real-time systems or other scenario exploiting approaches like [PBV+04, YMW+03].

References

[arm] ARM7TDMI datasheet. `http://www.arm.com/products/CPUs/ARM7TDMI.html`.

[BB00] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *Proc. of 25th Workshop on Real-Time Programming*, Palma, Spain, May 2000.

[BCP02] Guillem Bernat, Antoine Colin, and Stefan M. Petters. WCET analysis of probabilistic hard real-time systems. In *Proc. of 23rd IEEE Real-Time Systems Symposium*, pages 269–278, Austin, TX, December 2002.

[BCP03] Guillem Bernat, Antoine Colin, and Stefan M. Petters. pWCET, a tool for probabilistic WCET analysis of real-time systems. In *Proc. of Third International Workshop on Worst–Case Execution Time (WCET) Analysis*, pages 21–38, Porto, Portugal, July 2003.

[Bli94] J. Blieberger. Discrete loops and worst case performance. *Computer Languages*, 20(3):193–212, 1994.

[Bli01] J. Blieberger. Real-time properties of indirect recursive procedures. *Information and Computation*, 171(2):156–182, December 2001.

[BR05] C. Burguiere and C. Rochange. A contribution to branch prediction modeling in WCET analysis. In *Proc. of the Design, Automation and Test in Europe*, pages 612–617, Munich, Germany, March 2005.

[CB02] Antoine Colin and Guillem Bernat. Scope-tree: A program representation for symbolic worst-case execution time analysis. In *Proc. of 14th Euromicro Conference on Real-Time Systems ECRTS'02*, pages 50–63, Vienna, Austria, June 2002. IEEE.

[CG04] Matteo Corti and Thomas Gross. Approximation of the worst-case execution time using structural analysis. In *Proc. of the 4th ACM International Conference on Embedded Software*, Pisa, Italy, September 2004.

[DB73] J. Darlington and R.M. Burstall. A system which automatically improves programs. In *Proc. of the 3rd International Joint Conference on Artificial Intelligence*, pages 479–485, 1973.

[Lag01] K. Lagerström. Design and implementation of an MP3 decoder, May 2001. M.Sc. thesis, Chalmers University of Technology, Sweden.

[Lis03] Björn Lisper. Fully automatic, parametric worst-case execution time analysis. In *Proc. of Third International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 99–102, Porto, Portugal, July 2003.

[LM98] Yau-Tsun Steven Li and Sharad Malik. *Performance Analysis of Real-Time Embeded Software*. Kluwer Academic Publishers, 1998.

[4]The advantage of this method over the one based on profiling is that we are totally sure that the discovered scenarios cover all possible behaviors of the application. This is very important in the WCET analysis, as it must be conservative. For the approach based on profiling, the problem of covering all possible behaviors can be solved considering a backup scenario, which is equal with the entire application. However, in this case our method will not be able to reduce the application WCET.

[MACT89] A. K. Mok, P. Amerasinghe, M. Chen, and K. Tantisirivat. Evaluating tight execution time bounds of programs by annotations. In *Proc. of the 6th IEEE Workshop on Real-Time Operating Systems and Software*, pages 74–80, May 1989.

[MLCO04] Alexander Maxiaguine, Yanhong Liu, Samarjit Chakraborty, and Wei Tsang Ooi. Identifying ŕepresentativeẅorkloads in designing MpSoC platforms for media processing. In *Proc. of 2nd Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, Stockholm, Sweden, September 2004.

[Par92] Chang Yun Park. *Predicting Deterministic Execution Times of Real-Time Programs*. PhD thesis, University of Washington, Seatle, August 1992.

[PBV+04] Martin Palkovic, Erik Brockmeyer, Peter Vanbroekhoven, Henk Corporaal, and Francky Catthoor. Augmenting the exploration space for global loop transformations by systematic preprocessing of data dependent constructs. In *Proc. of Program Acceleration through Application and Architecture driven Code Transformations (PA3CT) Symposium*, Edegem, Belgium, 2004.

[PK89] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems*, 1(2):159–176, September 1989.

[Rij95] K. Rijkse. Video coding for narrow telecommunication channels at $<$64kbits/s. Technical report, Telenor R&D, 1995.

[RW94] Viresh Rustagi and David B. Whalley. Calculating minimum and maximum loop iterations. Technical report, Computer Science Department, Florida State University, May 1994.

[Sha89] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.

[Shl94] S. Shlien. Guide to MPEG-1 audio standard. *IEEE Transactions on Broadcasting*, 40(4):206–218, December 1994.

[VHMW01] Emilio Vivancos, Christopher Healy, Frank Mueller, and David Whalley. Parametric timing analysis. In *Proc. of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems (LCTES'2001)*, pages 88–93, Snow Bird, Utah, June 2001.

[WM05] L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Proc. of the Design, Automation and Test in Europe*, pages 600–605, Munich, Germany, March 2005.

[YMW+03] Peng Yang, Paul Marchal, Chun Wong, Stefaan Himpe, Francky Catthoor, Patrick David, Johan Vounckx, and Rudy Lauwereins. *Multi-Processor Systems on Chip*, chapter Cost-efficient mapping of dynamic concurrent tasks in embedded real-time multimedia systems. Morgan Kaufmann, 2003.

[ZWHM04] Wankang Zhao, David Whalley, Christopher Healy, and Frank Mueller. WCET code positioning. In *25th IEEE International Real-Time Systems Symposium*, pages 81–91, Lisbon, Portugal, December 2004.