INSTITUT DE RECHERCHE EN INFORMATIQUE DE TOULOUSE

Coding guidelines for WCET analysis using measurement-based and static analysis techniques

Armelle Bonenfant, Ian Broster, Clément Ballabriga,
Guillem Bernat, Hugues Cassé, Michael Houston, Nicholas Merriam,
Marianne de Michiel, Christine Rochange, Pascal Sainrat

March 2010

Technical report IRIT/RR—2010-8—FR

Abstract

This document has been written in the context of the MERASA project (2007-2010) within the Seventh Framework Programme of the European Union. In this project, a multi-core architecture has been designed hand in hand with timing analysis techniques and tools to guarantee its analysability and predictability regarding timing. Both static WCET analysis tools (the OTAWA toolset, developed in the IRIT research laboratory, Toulouse, France) as well as hybrid measurement-based tools (RapiTime, developed by Rapita Systems Ltd., York, UK) are considered in the project.

This report provides some recommendations for the use of the WCET tools as a set of coding guidelines. It is aimed at software engineers developing or integrating software for the MERASA architecture. The coding guidelines aim to allow analysable software to be written for the MERASA architecture. They consist of number of rules which should be followed when writing code, and present a number of techniques which may be used to improve analysis of existing code.
If the guidelines are followed when writing software, effort needed to adapt it to WCET analysis tools is reduced. Moreover, the need for analysis refinement to achieve a tight WCET estimates is lower. As a result, the cost is greatly reduced.

These guidelines will also help to ensure that software has predictable timing behaviour, which is essential to the reliability and safety of real-time embedded systems.

Introduction

The MERASA project (Seventh Framework Programme of the European Union) aims to design a multi-core microprocessor architecture and system-level software with predictable worst-case timing behaviour.
The tools considered for Worst-Case Execution Time (WCET) analysis of programs running on the MERASA multi-core architecture are:

- RapiTime, an hybrid measurement-based tool developed by Rapita Systems Ltd., York, UK
- OTAWA, based on static analysis techniques, designed at the IRIT research laboratory, Toulouse, France

Goal

This document is aimed at software engineers developing or integrating software for the MERASA architecture. It presents a set of coding guidelines which aim to allow analysable software to be written for the MERASA architecture. They consist of number of rules which should be followed when writing code, and present a number of techniques which may be used to improve analysis of existing code.

By following the guidelines, the cost of analysing software will be greatly reduced due to the reduction of effort needed to adapt it to WCET analysis tools, and to perform any necessary analysis refinements to achieve a tight WCET estimate.

These guidelines will also help to ensure that software has predictable timing behaviour, which is essential to the reliability and safety of real-time embedded systems.

Tools

The guidelines are written specifically for use with the project tools. A brief summary of the purpose of each of these is presented in this section.

oRange

oRange is a tool for automatic calculation of loop bounds from C source code. It is used by OTAWA to provide high-level constraints which would otherwise need to be manually provided by the user in the form of source code annotations.

OTAWA

OTAWA is a tool suite dedicated to the estimation of Worst-Case Execution Times using static analysis techniques. It includes a number of facilities to handle binary code and has been designed to support different architectures.

It provides an efficient framework to develop analysis modules that, when applied in sequence to binary code, help to tightly estimate the WCET.

It supports (or will support) everything needed for the MERASA Core architecture. OTAWA receives loop bounds produced by oRange and, if oRange cannot provide bounds, it is possible to give loop annotations in order to tighten the analysis.

Static analysis techniques that are part of OTAWA can provide an estimate of the WCET in most cases. Those where it cannot are described by the rules in this document.

RapiTime

RapiTime is a hybrid measurement-based WCET and timing analysis tool. It uses a trace of execution from the hardware itself to compute a worst-case estimate, and also provides detailed timing and profiling information.

Platform

Further system-level requirements are present which are a consequence of the design of the MERASA architecture and the system software.

MERASA HW

It is the hardware MERASA architecture.

MERASA OS

It is the system software which runs on MERASA HW. It provides a Posix-compatible interface to the hardware scheduler and other system-level components.

Applying the guidelines

Most of these guidelines concern C source code and some of them relate to assembly code.

In this document, the terms 'guideline' and 'rule' are used interchangeably.

This report is not a report on general good rules of coding as, for example, [MISRA04]. Note that several of the rules are also in [MISRA04], so if you are used to respecting the MISRA rules then it is highly probable that your code will be correctly analysed with oRange, OTAWA and RapiTime.

oRange

Coding rules for oRange may be neglected if annotations are used to provide flow information to OTAWA. However, it is better to leave oRange find the loop bounds as it is less error-prone.

All oRange coding rules concern the control flow of the program. Some rules must be respected because oRange will otherwise fail to analyse the source code. In the worst case, oRange will be unable to find some loop bounds and in most cases, it will produce overestimated loop bounds. This is indicated for each rule.

OTAWA

Most OTAWA rules should be respected, except where specified when an annotation can be used instead.

RapiTime

RapiTime rules marked as 'Recommended' are not necessary to complete an analysis of the software, but will improve the accuracy of the WCET calculation.

Rules marked as 'Required' will prevent RapiTime analysing the software if they are not followed. In some cases annotations can be provided for cases which RapiTime cannot automatically handle, and these are listed in the exceptions section for each rule.

Terms

Several terms are used throughout this document. They are described here in more detail.

Worst-Case Excecution Time (WCET)

The WCET of a function or application is the theoretical maximum time that it could take to complete its execution. This is important for scheduling software in a real-time system so that it meets its deadlines.

Root Function

The Root Function is the scope of analysis in RapiTime – the unit of code for which a WCET is reported.

Root Exit

A Root Exit is a statement which is considered to be the end of the Root Function, usually a return statement, or an annotated goto.

Annotation

An annotation is a source code comment or compiler directive which provides an instruction to an analysis tool which provides more information about how the source code should be analysed. For full details of the annotations available for each tool, please see the relevant product manuals.

Template

Each rule follows the same standard template. The template is based on the one used by MISRA.

The template begins with the rule title, which is in the form of an instruction to the developer.

Immediately below the title is a row of boxes, one for each part of the MERASA project. Each box indicates how the rule applies to that component. The possible applications are:

Required – This rule must be obeyed for the correct functioning of the component

Recommended – This rule will improve the functioning of the component, but is not required.

Caution – Caution should be taken with the situation described by the rule, but it will not have a serious functional impact if it is ignored.

Not Recommended – Application of this rule will have a negative effect on the component.

Blank – This rule does not apply to the component.

Below is a representation of how the template will appear:

Rule x: The programmer should follow this instruction

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS

Explanation

A more detailed explanation of how to follow the rule.

Rationale

The rationale describes the technical reasons why the rule should be followed, and describes the consequences if it is not.

Exceptions

Any cases where the rule does not apply, or a work around can be used. For example, many rules have associated annotations which can be added to the source code to communicate additional information to the analysis tools.

Example

A concise code example should be provided for each rule, illustrating its application.

Control Statement Expressions

Do not use assignment operators in expressions that yield a boolean value

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
Recommended				

Explanation

An assignment statement should not be used as part of a compound expression which returns a boolean value.

Rationale

oRange cannot compute the correct loop bounds if this construction is used.

Exceptions

Example

Avoid	Prefer

`if ( ( x = y ) != 0 )`
`{`
`  foo();`
`}`
	`x = y;`
	`if ( x != 0 )`
	`{`
	`  foo();`
	`}`

Reserve expressions in for statements for loop control

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
Required				

Explanation

The three expressions of a `for` statement should be simple expressions concerning only the loop counter.

Rationale

oRange cannot compute the correct loop bounds if complex expressions are used in `for` statements.

Exceptions

oRange can sometimes produce a loop bound in the case of an && operator but will usually produce the highest: 5 in the example below. Any other operator is not supported.

Example

Avoid	Prefer

`flag = 1 ;`
`for (i = 0; (i < 5) && (flag == 1);`
`i++)`
`{`
`  ... ;`
`  if (...)`
`  {`
`  flag = 0;`
`  }`
`}`
	`flag = 1 ;`
	`for ( i = 0; (i < 5 ); i++)`
	`{`
	`  ... ;`
	`  if (...)`
	`  {`
	`  break;`
	`  }`
	`}`

Do not modify the loop counter inside a conditional statement

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
Required				

Explanation

Modification of the loop counter inside a condition statement should be avoided.

Rationale

oRange cannot correctly calculate the loop bound if the loop counter is modified inside a conditional statement.

Exceptions

This rule can be relaxed if the loop counter is only modified monotonically in the same direction as the `for` statement increments it.

Example

The left example should be absolutely avoided. It's better, if possible to avoid the right example.

Avoid	Prefer

`for ( i = 0 ; (i < 5 ) ; i++)`
`{`
`  ... ;`
`  if (...)`
`  {`
`  i-- ;`
`  }`
`}`
	`for ( i = 0 ; (i < 5 ) ; i++)`
	`{`
	`  ... ;`
	`  if (...)`
	`  {`
	`  i++ ;`
	`  }`
	`}`

Avoid input-dependent loop bounds

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
Recommended		Recommended		

Explanation

Do not use loops which iterate over a data structure with dynamic size.

Rationale

The worst-case will be the size of the input. In many cases, a loop depending on an input can be replaced by a loop with a tighter bound. In some cases, for example when going through a sorted array, it is possible to proceed by binary search which gives a tighter upper-bound.

Exceptions

Example

In this example, the variable bound is replaced by a static bound. Also note that the iteration order has been reversed to preserve the semantics of finding the first match in the array, and the whole array is searched. This is not a common way to write a search routine, but is the case that would be reflected in the WCET.

The re-written search function assumes that any entries in the array beyond the 'valid' entries do not match **x**.

Avoid	Prefer

`int find(int x, int array_len, int* array)`
`{`
`  int i;`
`  int found = -1;`
`  for (i = 0; i < array_len; i++)`
`  {`
`  if (array[i] == x)`
`  {`
`  found = i;`
`  break;`
`  }`
`  }`
`  return found;`
`}`
	`#define MAX_ARRAY_LEN 255`
	
	`int find(int x, int* array)`
	`{`
	`  int i;`
	`  int found = -1;`
	`  for (i = MAX_ARRAY_LEN - 1;`
	`  i >= 0; i--)`
	`  {`
	`  if (array[i] == x)`
	`  found = i;`
	`  }`
	`  return found;`
	`}`

Do not use goto

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
Recommended		Recommended		

Explanation

`goto` statements should not be used.

Rationale

Statements which disrupt the flow of execution in arbitrary ways make the structure of the program difficult to determine.

This rule is not mandatory for oRange but avoiding `goto` eases analysis.

RapiTime will successfully parse the source code if a `goto` is present, but paths containing the `goto` will not be considered for the worst-case path. This may result in incorrect WCET calculations.

Exceptions

Example

Do not use setjmp or longjmp

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
Recommended	Required	Recommended		

Explanation

The `setjmp` and `longjmp` assembly instructions should not be used.

Rationale

Statements which disrupt the flow of execution in arbitrary ways make the structure of the program difficult to determine.

`setjmp` and `longjmp` are not supported by OTAWA.

`longjmp` is allowed in RapiTime if it is a root exit and is annotated with exit_function.

Exceptions

Example

Do not use break, continue or return statements to exit a loop early

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
Recommended				

Explanation

Loop structures should be as simple as possible, with no early exits.

Rationale

In some cases, the use of `break`, `continue` or `return` in the middle of a control flow structure may prevent oRange calculating a loop bound.

Exceptions

Example

switch statements should be well-structured

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
Recommended		Recommended		

Explanation

Keep `cases` and `breaks` at the same nesting level.

Rationale

Placing `break` statements inside nested conditionals introduces mutually exclusive paths in the code which cannot be readily analysed by the tools.

Exceptions

Example

Avoid	Prefer

`switch (v)`
`{`
`  case 1 :`
`  case 2 :`
`  if (...)`
`  {`
`  ... ;`
`  break;`
`  }`
`  ...`
`  break ;`
`}`
	`switch (v)`
	`{`
	`  case 1 :`
	`  case 2 :`
	`  if(...) {`
	`  ... ;`
	`  }`
	`  else`
	`  {`
	`  ... ;`
	`  }`
	`  break ;`
	`}`

Do not use large switch statements

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
	Caution	Not Recommended *		

Explanation

Replace large switch statements with a series of `if … then…else` statements.

Rationale

Large switches are not supported by OTAWA because they can be translated by the compiler into branch tables.

Exceptions

Keeping a switch is possible but it will need manual annotations which is error prone.

Note that some compilers provide options to configure the way the switch is translated and to prevent the generation of indirect branch tables. OTAWA provides for some architectures and some compilers (but not the Tricore) switch analysis through pattern recognition.

*RapiTime analysis and the performance of the code are improved if large switch statements are compiled to lookup tables rather than cascading if,

then, else statements. The worst-case performance for cascading tests requires that all the conditions are evaluated.

Example

Do not interleave case statements with other control structures (cf. Duff's device)

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
Recommended		Required		

Explanation

The C language syntax allows some unusual constructions, including the interleaving of a loop and switch statement. You should not use them.

Rationale

Interleaved control-flow statements are not supported by the tools, and will prevent an analysis being made.

Exceptions

Example

Duff's device is an implementation of a serial copy (from http://en.wikipedia.org/wiki/Duff's_device):

```
send(to, from, count)
register short *to, *from;
register count;
{
        register n=(count+7)/8;
        switch(count%8){
        case 0: do{     *to = *from++;
        case 7:         *to = *from++;
        case 6:         *to = *from++;
        case 5:         *to = *from++;
        case 4:         *to = *from++;
        case 3:         *to = *from++;
        case 2:         *to = *from++;
        case 1:         *to = *from++;
                }while(--n>0);
        }
}
```

Do not use infinite loops

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
Recommended	Recommended	Recommended		

Explanation

You should not write code which contains loops with no termination condition.

Rationale

An infinite loop, by definition, has an infinite WCET.

Exceptions

It is possible to annotate an infinite loop with a maximum number of iterations in OTAWA.

RapiTime will use the observed number of iterations in its calculations, but if the main function never terminates in testing, RapiTime will not be able to compute an estimate since it will not have seen a 'complete run' of the main program. The tool includes an option to use an infinite loop inside the main function as the analysis root, which treats each iteration of the loop as a complete execution run of the program.

Example

A very simple example of an infinite loop is shown below. By rewriting the body of the loop as a function, an estimate for the execution time of one loop iteration can be determined by choosing the loop_body function as the analysis root.

Avoid	Prefer

`void main() {`
`  while (1)`
`  {`
`  statements;`
`  }`
`}`
	`void loop_body() {`
	`  statements;`
	`}`
	
	`void main() {`
	`  while (1)`
	`  {`
	`  loop_body();`
	`  }`
	`}`

The WCET of the whole program cannot be computed since the program runs forever. The user should provide an annotation on the while loop to indicate the maximum possible number of iterations if it is necessary to determine the overall execution time.

Functions

Do not use variable numbers of arguments (varargs)

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
Required	Recommended	Recommended		

Explanation

Variable arguments such as those of the libc `printf()` function should be avoided.

Rationale

In general, varargs are equivalent to variable size input data.

oRange does not support the use of va_arg, va_start, va_end.

For OTAWA, the use of function with a variable number of arguments is not advised since it requires annotations to bound the loop that retrieves the arguments.

Similarly, RapiTime will produce a more accurate analysis if the execution time is not dependent on the input data.

Exceptions

Example

Do not use recursion

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
Recommended	Recommended	Recommended		

Explanation

Do not use recursion in your application – either direct (a() → a()) or indirect (a() → b() → a())

Rationale

Generally speaking, recursion should be avoided because it carries with it the danger of exceeding available stack space.

For worst case analysis, recursive calls are difficult to model; recursion is often used to process dynamically structured input data, and as a result causes the same dependence of timing behaviour on input data.

Exceptions

oRange supports some simple forms of recursion (cases where the recursion can be transformed in a for loop). It requires annotations to bound the loop.

RapiTime supports direct recursion, but it must be annotated with a bounded recursion depth. Indirect recursion is not supported in the WCET analysis, the recursive function group is automatically 'black-boxed', which will encapsulate the entire scope of the recursion with a single end-to-end measurement.

Example

Use const pointers for unmodified pointer parameters

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
Recommended				

Explanation

If a pointer is a parameter of a function and the pointer is not modified in the function, it is better to use `const`.

Rationale

`const` pointers allow the analysis to assume that the target of the pointer will not be modified. This improves the loop bound analysis.

Exceptions

Example

The following statement is acceptable if **a** is modified in the function, and **b** is not.

```
foo(int *a, const int *b)
```

Pointers

Only use pointer arithmetic for array indexing

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
Required				

Explanation

Arithmetic on pointers should only be used for array indexing.

Rationale

When not used to index arrays, pointer arithmetic is usually used for memory management. In this case, loops bounds are hard to determine since memory operations potentially impact any word in the memory.

Exceptions

Example

Do not use multiple indirection

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
Required				

Explanation

Pointers to pointers, or further levels of indirection, should not be used.

Rationale

As the level of pointer indirection increases, difficulty of analysis increases. oRange only supports one level of indirection.

Exceptions

Example

Do not use function pointers

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
Required	Caution	Caution		

Explanation

Function pointers should not be used.

Rationale

Function pointers are not supported by oRange.

Exceptions

RapiTime can analyse function pointer calls so long as the pointer is marked with **call_to** annotations. These annotations are generated automatically from a trace of execution.

If you use function pointers, you must make sure that your tests exercise all possible function pointer targets, otherwise the analysis will be incorrect. You can manually add extra **call_to** annotations to indicate a possible code path which is not covered by your tests.

OTAWA requires user annotations to indicate possible targets of an indirect call.

Example

Data structures

Do not use union data types

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
Required				

Explanation

Do not use unions in your code.

Rationale

Unions are not supported by oRange.

Exceptions

Example

System-level Requirements

Do not modify the stack pointer

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
	Required	Required		

Explanation

It is forbidden to have a function which modifies the stack pointer so that the pointer is not the same after the function has been executed.

Rationale

The execution path cannot be determined if the normal call/return semantics of C are not obeyed.

Exceptions

Example

Avoid use of blocking system calls

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
	Required	Caution		

Explanation

Avoid making calls to system functions which are called using a software interrupt or other supervisor-mode operation.

Rationale

System calls are not supported as the target depends on the configuration of the underlying OS.

Exceptions

OTAWA: The target of the system call may be provided by hand and considered as a normal branch provided the system routine is in the binary code.

RapiTime: The system call is considered to be a 'black-box' function, and the maximum end-to-end time is recorded. This may not be the worst-case time for the system call, especially in the case on blocking code which is non-deterministic.

If possible, the system software should also be instrumented if the behaviour is likely to have large execution time variability.

Example

Avoid hardware control instructions

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
	Required			

Explanation

Do not use instructions which affect the state of the hardware, such as controlling the behaviour of cache, MMU, etc.

Rationale

Most of these instructions are not handled correctly by the WCET analysis in OTAWA. Do not expect their effect to be correctly handled unless it is explicitly stated in documentation. For example, if you use an instruction which inhibits the cache, the fact that the cache is inhibited for the rest of the program is not taken into account.

Exceptions

Example

Instrument context switches

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
		Recommended		

Explanation

When the operating system performs a context switch, this event should be recorded in the RapiTime trace. The event should indicate the ID of the thread that is now active.

Rationale

Context switches indicate that a new program is now executing. The switch may occur in the middle of a test run, and the trace event is needed to enable RapiTime to determine which run the execution time should be accounted to.

Exceptions

If your operating system does not support time slicing and the root is a 'one-shot' function which runs to completion after a release, you do not need to instrument context switches.

On the MERASA architecture, the Hard Real-Time threads (HRT) are considered to be non-pre-emptable, however due to the Symmetric Multi-Threaded nature of the hardware scheduler, several non-HRT threads may also issue instructions when the HRT thread is blocked. It is important for the purposes of tracing that it is possible to determine which thread is running and separate out the traces accordingly. This is accomplished in the MERASA architecture by recording the thread ID with each recorded ipoint.

Example

WCET Overestimation

Do not use dynamic memory allocation

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
	Recommended	Recommended		

Explanation

Do not use `malloc()` and related memory allocation functions.

Rationale

The management of dynamic memory involves complex algorithms that make harder the prediction of the used memory. It may even be impossible to predict it if the program contains complex allocation patterns. This has mainly a bad impact on the prediction of the data cache use (for example, all accesses will be considered as misses).

Instead of using standard allocation primitives of C, a good solution is to develop specialized allocators based on array whose memory area is well known because it is reserved statically. This is usually known as a Slab Allocator.

Exceptions

Example

Avoid

```
struct node_t {
  struct node_t *next;
  ...
} *p = NULL, *q;

for (...) {
  q = (struct node_t *)
    malloc(sizeof(struct node_t));
  q->next = p;
  p = q;
  ...
}
```

Prefer

```
struct node_t {
  struct node_t *next;
  ...
} *p = NULL, *q, tab[MAX];
int next_block = 0;

for (...) {
  q = &tab[next_block];
  next_block++;
  q->next = p;
  p = q;
  ...
}
```

Do not allocate dynamic arrays on the stack

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
	Recommended			

Explanation

Some compilers allow declaring arrays with dynamic size in the local variables of a function. Do not use this feature.

Rationale

This makes the address in the stack dependent on the flow of the data and makes the prediction of the data cache usage harder.

In addition, this kind of allocation makes the prediction of the stack size harder or impossible if the data flow determining the size is dependent on the program input.

Exceptions

Example

Avoid	Prefer

`void f(..., int n, ...) {`
`int t[n];`
`...`
`}`
	`int t[MAX_N];`
	`void f(..., int n, ...) {`
	`...`
	`}`

Do not use alloca()

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
	Recommended			

Explanation

`alloca()` allocates memory on the current stack frame. It should not be used.

Rationale

The use of this standard libc routine is not advised by the manual pages. In WCET computation, it produces the same effect as described in the previous rule.

Exceptions

Example

Avoid conditional code in loops

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
	Caution	Recommended		

Explanation

The body of a loop should contain as few conditional paths as possible.

Rationale

Any overestimation of the loop's execution time will be multiplied by the number of loop iterations.

When a condition is not simple enough to be evaluated by the WCET analyser, the most costly part of the loop body will be considered by the WCET analysis to execute on every iteration of the loop.

If this conditional part is executed only rarely, it will induce a large overestimation. This is especially pronounced when the selection has an empty 'else' part and the 'then' part is used only once in a loop as shown in the following examples.

Exceptions

RapiTime allows the frequency of conditional blocks inside a loop to be marked using the **wfreq** annotation. This only works for one level of loop nesting however.

Examples

It is a common practice to perform some special initialisation in the first loop iteration, which is not executed in subsequent iterations.

Avoid	Prefer

`for (i = 0; i < N; i++) {`
`  if (i == 0) {`
`  /* initialisation */`
`  }`
`  else {`
`  ...`
`  }`
`}`
	`i = 0;`
	`/* initialization */`
	`for(i = 1; i < N: i++) {`
	`  ...`
	`}`

In the following example, the algorithm is looking for a unique t[i] in order to perform processing on it. If the condition is too complex, the WCET analyser will conclude that the work is done for each element of t in the left version of the program and only once in the right one

The inserted boolean **found** does not break compatibility with oRange as it is not used in the loop condition.

Avoid	Prefer

`for(i = 0; i < N; i++) {`
`  if(... t[i] ...) {`
`  /* work on t[i] */`
`  break;`
`  }`
`  ...`
`}`
	`  int found = 0;`
	`  for(i = 0; i < N; i++) {`
	`  if(... t[i] ...) {`
	`  found = 1;`
	`(1)  break;`
	`  }`
	`  ...`
	`  }`
	`  if(found) {`
	`  /* work on t[i] */`
	`  }`

Note: the break statement at (1) conflicts with Rule 18. It is used here in order to reduce another type of analysis problem, but caution should be taken.

Avoid multiple execution paths

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
	Recommended	Recommended		

Explanation

Multiple paths through the code, such as conditional statements, should be avoided.

Rationale

The main cause of overestimation is due to the lack of determinism in the application. From a C source point of view, indeterminism is caused by the number of execution paths. In the analysis, every joining of paths may cause a loss of precision.

To reduce the number of paths, one has to avoid the use of:

conditional statements,

loops with a variable bound.

Compilers' optimisations provide ways to enforce these restrictions. Algorithms can often be designed to improve performance and exploit parallelism in the processor by aggregating adjacent code blocks. As a side effect, this reduces the number of paths.

The price is an increase in the size of the program due to code duplication. As we are targeting embedded real-time applications, a trade-off between determinism and code size should be found.

These optimisations should be performed manually on the source code because using those of the compiler may not lead to analysable code. The most common optimisations are:

- function inlining – a small function call is replaced by the function's code,
- loop fusion – adjacent loops with the same number of iterations are merged,
- loop unrolling – the loop is unrolled to exhibit more parallelism in each iteration,
- single-path code – use of guarded instructions to avoid branches.
- super-block – if a selection is followed by a simple block, this block may be duplicated in each branch.

In summary, multiple execution paths make the WCET difficult to determine, because the longest possible path must be considered, even if it is never taken during execution of the application.

Exceptions

Example

Prefer structural conditionals over data dependency

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
		Recommended		

Explanation

Code in which the control-flow is dictated by data often has mutually-exclusive blocks which are not visible from the source code. In such cases, a structural representation of the mutual exclusion is preferred. This has several trade-offs, and in some cases requires duplication or refactoring of the source code to preserve the correct behaviour.

Rationale

Data dependency should be avoided, as previously described in 15.

Exceptions

You can use **lwpath** annotations to avoid RapiTime assuming each block executes on every iteration of an enclosing loop.

Example

In this example common code has been cloned into both conditional blocks to enforce the mutual exclusion of the blocks in the example on the left.

Avoid	Prefer

`if (a) {`
`  ...`
`}`
 	
`/* common code */`
 	
`/* b = !a */`
`if (b) {`
`  ...`
`}`
	`if (a) {`
	`  ...`
	`  /* common code */`
	`}`
	`else {`
	`  /* common code */`
	`  ...`
	`}`

Improving WCET analysis

In some cases, it is not possible to rewrite the code in order to avoid overestimation caused by flow analysis. For some of them, one can provide manual annotations in order to help OTAWA and RapiTime reduce the overestimation.

Annotate loop bounds

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
	Recommended	Caution		

Explanation

In order to obtain a more precise loop behaviour description, one should give several annotations on inner nested loops:

- total: the maximum number of iterations over all the program execution,
- minimum: minimum number of iterations,
- context: different maxima, total or minima according to the call chain leading to the loop.

Rationale

Loops with complex or variable bounds (most often these are nested loops) are a source of overestimation because, for each entry in the loop (i.e. each time the outer loop is executed) the maximum number of iterations will always be considered.

Exceptions

RapiTime uses observed loop bounds in its calculations. If the observed loop bounds are not sufficient (because the tests do not exercise the full range of possible loop conditions), annotations are required to improve the WCET estimate.

Example

In this example, the inner loop iterates at most N times each time it is called and as loop (1) iterates N times, the analyser may conclude that loop (2) iterates N^2 in total. But, the programmer knows that the inner loop iterates N (N + 1) / 2.

Sample	Annotations

(1) for(i = 0; i < N; i++)
(2) for(j = 0; j < i; j++) {
/* body */
}
}
	Loop (1)
	maximum = N
	total = N
	
	Loop (2)
	minimum = 0
	maximum = N
	total = N * (N + 1) / 2

Below, one can see that the overall maximum iteration count is 105 while, in case (1), the local maximum is 5 and in case (2), it is 100. Both local maxima are very different and the overall maximum would induce a big overestimation. Considering only the total is also not enough as the computation may be free to assign 100 iterations to the first call to maximize other features cost like cache misses. In fact, it is more precise to consider maxima and total for each call context.

Sample	Annotations

void f(..., int n, ...) {
int i;
for(i = 0; i < n; i++) {
...
}
}
 	
int main() {
...
(1) f(..., 5, ...);
...
(2) f(..., 100, ...);
...
}
	Loop in f
	maximum = 100
	total = 105
	
	Call (1) / loop
	maximum = 5
	total = 5
	
	Call (2) / loop
	maximum = 100
	total = 100

Annotate infeasible paths

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
	Recommended	Recommended		

Explanation

An important source of overestimation is the inclusion in the WCET analysis of infeasible paths, that is, paths that are not in the set of possible execution paths due to the program semantics. Although WCET analysers can find automatically some of these paths, it remains some constructions that remain too hard to handle but the developer may help to avoid such a kind of overestimation.

Rationale

Exceptions

Example

In this very simple example, you can also provide an annotation on (1). Yet, it may be hard or error prone to find by hand all infeasible paths. A better approach is to consider only cases of a selection where the "then" and "else" parts have very different costs for the WCET computation and to put an annotation on the most costly branch.

Sample	Annotations

`for(i = 0; i < 100; i++){`
`  if(i % 2 == 0) {`
`(1)  /* light block */`
`  }`
`  else {`
`(2)  /* heavy block */`
`  }`
`}`
	`Annotation on (2)`
	`  execute at most 50 times`

Parallel Programming

Avoid sharing memory between threads

oRange	OTAWA	RapiTime	MERASA HW	MERASA OS
			Recommended	

Explanation

Consider copying result from one thread to memory of another as producer/consumer pattern.

Rationale

Exceptions

Example

References

[MISRA04] MISRA-C:2004 - Guidelines for the use of the C language in critical systems

[RPT09] RapiTime WCET coding guidelines, Rapita Systems Ltd., 2009.

[UPS09] Coding guidelines for static analysis, Technical Report, UPS, 2009.