

RapiTime White Paper

"The best model of a system is the system itself"

Worst-Case Execution Time Analysis

This white paper outlines the problems of timing faults in sophisticated real-time embedded software. It presents the RapiTime toolset, an essential building block in any credible strategy aimed at eliminating timing problems.

Disclaimer

The information, text and graphics contained in this document are provided for information purposes only by Rapita Systems Ltd. Rapita Systems Ltd. does not warrant the accuracy, or completeness of the information, text, and other items contained in this document.

Copyright and Trade Marks

All rights reserved. Information and images contained within this document are copyright and the property of Rapita Systems Ltd. All trademarks are hereby acknowledged to be the properties of their respective owners.

Contact

Rapita Systems Ltd.
IT Centre
York Science Park
York
YO10 5DG

Tel: +44 1904 567747
<http://www.rapitasystems.com/>
enquiries@rapitasystems.com

Document Control

RapiTime *White Paper*

Revision 2.02

12th June 2008

Part #: DOC-060118-1

1. Introduction

The last 10 years have seen significant advances in microprocessor technology, with a doubling of computing power roughly every two years. Whilst increased miniaturisation is responsible for some of this gain, the current trend is to further increase processor performance via the adoption of advanced hardware features such as data and instruction cache, pipelines, branch prediction units and out-of-order execution. The effect of these hardware acceleration features is considerable, typically improving processor performance by more than an order of magnitude.

In the avionics, automotive electronics and telecommunications industries the demand for advanced functionality places significant pressure on manufacturers to develop more complex systems in less time and at lower cost, whilst maintaining their reputations for high quality and reliability. As the functionality provided becomes more complex, so the job of testing for correct functional and in particular correct timing behaviour becomes ever more difficult. However, the damage that intermittent software glitches can do to profit margins is considerable, with production delays, no-fault-found replacements and damage to the company's reputation. As a result there is considerable interest in engineering approaches and tools that can be used to detect potential timing problems during the early stages of development.

Avionics systems, automotive electronic control units (ECUs) and telecommunications devices are all forms of embedded real-time system. By real-time, we mean that there are deadlines by which operations must be completed otherwise the system may fail. For example, the ECU controlling an airbag must activate the airbag when it detects sensor readings above a certain threshold (correct functionality) and it must do so within a specific deadline measured in milliseconds (correct timing behaviour).

Worst-Case Execution Times

The fundamental problem in building any embedded real-time system is ensuring that the system will always perform its required functionality within the specified time constraints. For example an engine management system must process sensor data and determine the correct fuelling requirements within a few milliseconds. If this computation is not completed in time, then the engine will not run smoothly and may not meet stringent emission requirements.

Obtaining accurate information about the longest time a piece of software can take to run, termed the *worst-case execution time*, is key to ensuring that time constraints are met and that a real-time system operates correctly.

Many low-end 8 and 16-bit microprocessors have relatively simple architectures. These processors take a fixed number of clock cycles to execute a given machine code instruction. This means that it is possible, although by no means easy, to use extensive testing to reveal close to worst-case execution times for the various software components. Engineers can and do use these values plus some margin for error, along with a model of the operating system or cyclic executive, to predict the timing behaviour of the overall system.

With increases in software complexity and the adoption of more advanced and powerful 32-bit processors such as Freescale Semiconductor's (Motorola's) MPC, Infineon's TriCore and NEC's V850E, this approach proves insufficient. There are two reasons for this:

Firstly, as the size and complexity of software increases there is an explosion in the number of possible paths through the code, this makes it very difficult to find and exercise the worst-case path during testing.

Secondly, the hardware itself contains features such as instruction and data caches, pipelines and branch prediction units. These features significantly improve average-case performance vastly increasing throughput, however they make the worst-case execution time much more difficult to predict. Instructions no longer take a fixed number of clock cycles, resulting in a single path through the code exhibiting wide variation in its execution times.

On advanced microprocessors, it is simply not possible during testing to ensure that all the functions and loops on a single path have exhibited their worst-case times simultaneously. This means that testing alone cannot provide the level of confidence in the timing behaviour of the system necessary for reliable deployment.

An alternative approach to determining the worst-case execution time is to use *Static Analysis*. The basic approach of static analysis involves computing segments of paths through the code. Using information about these sub-paths and how they can combine, along with a precisely accurate model of the behaviour of the hardware, it is in theory possible to predict the worst-case execution time. In practice however, whilst static analysis is possible for simple processors with instructions that take a fixed time, there are few tools that provide these facilities – those that do tend to be in the safety critical domain. For many advanced 32-bit processors, the only really accurate model of the hardware's timing behaviour is the processor itself, making static analysis infeasible.

The outlook in terms of tool vendors developing commercially viable static analysis tools is uncertain. Studies indicate that the long lead-time and high costs of supporting advanced new processor features combined with the pace at which new and improved microprocessors are introduced makes the return on investment poor.

This situation leaves manufacturers in the avionics, automotive electronics and telecommunications markets with a potentially costly dilemma. To remain competitive, they must add desirable features to their high-end products, requiring ever more complex software and more advanced microprocessors and yet software glitches caused by timing faults threaten to undermine product quality and reliability.

2. RapiTime Overview

RapiTime is a software toolset that provides a *unique* solution to the problem of determining worst-case execution time information. A solution that works for complex software running on advanced microprocessors.

RapiTime uses an innovative combination of three techniques:

1. The designers of RapiTime recognise that the best possible model of an advanced microprocessor is the microprocessor itself. RapiTime therefore uses *online* testing to measure the execution time of sub-paths between decision points in the code.
2. By contrast, *offline* static analysis is the best way to determine the overall structure of the code and the paths through it. RapiTime therefore uses path analysis techniques to build up a precise model of the overall code structure and determine which combinations of sub-paths form complete and feasible paths through the code.
3. Finally RapiTime combines the measurement and path analysis information using state-of-the-art statistical methods, known as the theory of Copulas, to compute worst-case execution times in a way that captures accurately the execution time variation on individual paths due to hardware effects.

Obtaining accurate worst-case execution time bounds is the fundamental building block from which confidence in a system's timing correctness can be built. This makes RapiTime a key asset in engineering mission critical embedded real-time systems.

In addition to worst-case execution time information, RapiTime provides a wide range of performance profiling information, outlined in section 3.

Toolset

The RapiTime toolset comprises the set of command line tools listed below, and an Eclipse based **Report Viewer**:

Tool	Purpose
cins	Automatic instrumentation of C / Ada source code.
adains	Extraction of structural information from instrumented source files.
xstutils	Overall structural analysis.
traceutils	Pre-processing of timing traces.
traceparser	Generation of measured execution time data from structural and trace information.
wcalc	Worst-case execution time calculation, generation of computed worst-case execution time data.

The final output from the RapiTime command line tools is a database of execution time information that can be accessed using the RapiTime **Report Viewer**.

Analysis Process

The RapiTime analysis process, shown in Figure 1, comprises six stages:

1. Build
2. Structural analysis
3. Testing and trace generation
4. Trace processing
5. Worst-case execution time calculation
6. Viewing the report.

The overall analysis process is depicted in Figure 1 and described in detail below.

Note, the description below applies to C source code. RapiTime can also be used with Ada source code. In this case, the **adains** tool is used to automatically instrument the source code and to extract structural information from the disassembled executable.

1 Build

The first step is to create an instrumented build of the application software. This can be done by manually or automatically adding instrumentation points to the C source code (`hello.c`). For automatic instrumentation, the input code needs to be pure C code with all macros and `#includes` expanded by the C pre-processor. This pre-processed code (`hello.p`) is passed to the **cins** tool, which automatically adds instrumentation points; typically to all control flow decision points in the program. The instrumentation takes the form of lightweight timing measurement code similar to that used by conventional code profiling tools. The instrumented software (`hello.i`) and instrumentation library code (`rpt.c`) are compiled and linked using the standard compiler tool chain. The executable produced is then downloaded onto the target hardware.

As part of the build process, the **cins** tool also extracts structural information from each instrumented C source file. This information is stored in the form of extended syntax tree components (`hello.xsc`).

2- Structural Analysis

The **xstutils** tool is used to analyse the structural information contained in the `.xsc` files produced by **cins**. **xstutils** produces a description of the overall structure of the code called an extended syntax tree. The extended syntax tree is output in the form of a parsing engine (`hello.xse`) used later in both trace processing and worst-case execution time calculation.

3- Testing and Trace Generation

The executable (`hello.exe`) is run on the target microprocessor or on a cycle accurate CPU simulator and subjected to a variety of tests. The aim of these tests is to exercise all the small sub-paths through the code under a variety of conditions.

With the instrumentation-based approach, whenever an instrumentation point is reached during testing, trace information is recorded in the form of an instrumentation point identifier and a timestamp. This trace information is extracted at the end of each test and written to a file (`trace.txt`) for later analysis.

Alternatively, if hardware support is available, then the instrumentation point identifier is simply written to an output port which is monitored by the RapiTime TraceBox or other hardware trace capture device, such as a Logic Analyser. The TraceBox records and timestamps the instrumentation point identifiers, storing the resultant timing trace in a file (`trace.txt`) for later processing.

4- Trace Processing

The **traceutils** tool pre-processes the files containing timing traces generated during testing. It filters the data and corrects for situations such as timer wrap-around. **traceutils** produces compressed trace files in binary format (`trace.rpz`). The **traceparser** tool parses these binary traces using the parsing engine created by **xstutils**. The resulting output is a distribution of observed execution times for a hierarchy of scopes or *elements*. These elements include high level functions, sub-functions, loops, blocks of code, and finally, each unique sub-path between decision points in the code. The execution time data is stored in a RapiTime database (`.rtd`) file.

5- WCET Calculation

Once the traces have been processed, the next stage of the analysis process is the calculation of worst-case execution times. The **wcalc** tool does this using information about the observed execution times of sub-paths (in the `.rtd` database file) and the overall structure of the code (`.xse` file). **wcalc** produces worst-case execution time information for each element. This is stored in the RapiTime database (`.rtd`).

6- Viewing the Report

The final stage involves using the RapiTime **Report Viewer** to view worst-case execution time information, and performance profiling data from the database.

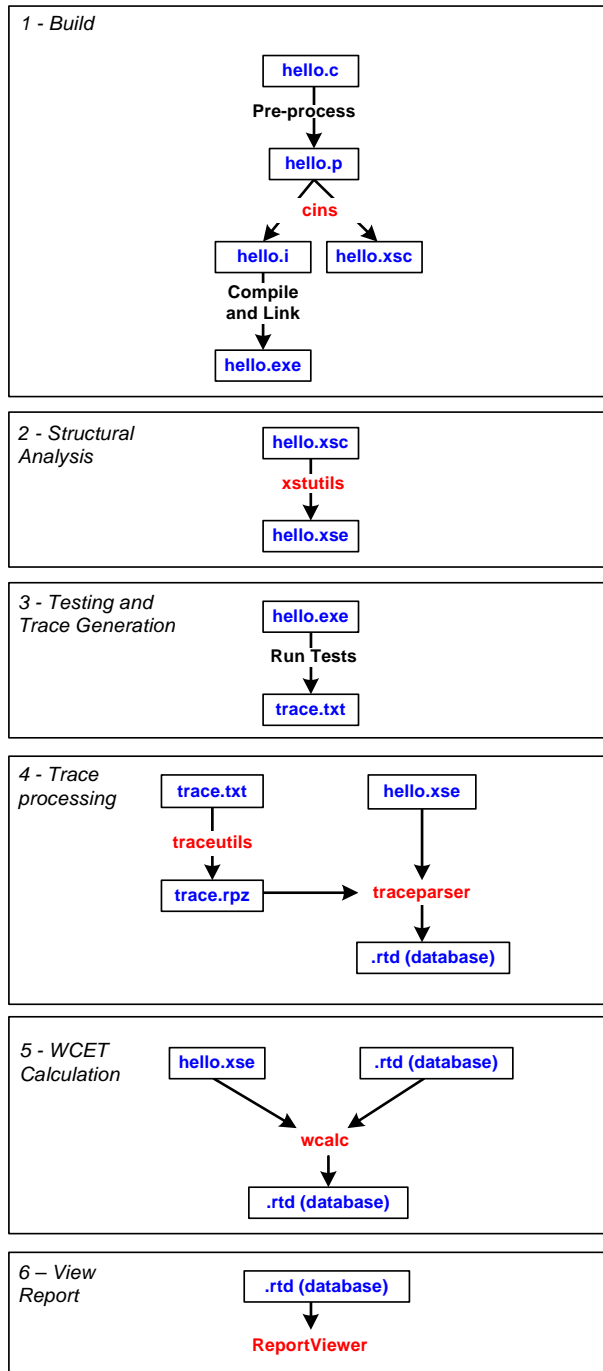


Figure 1: Stages of the analysis process

3. RapiTime Report

RapiTime provides a wealth of worst-case execution time, and performance profiling information that can be accessed via the **Report Viewer**, an Eclipse-based graphical user interface.

RapiTime provides eleven categories of performance profiling and worst-case execution time information:

1. Static data.
2. Code coverage.
3. Worst-case execution time.
4. High water mark.
5. Maximum
6. Minimum
7. Average case.
8. Code metrics.
9. Optimisation.
10. Budgets.
11. Comparisons.

RapiTime provides this information for each of the *elements* analysed. The *elements* analysed include sub-programs or functions, their specific calling contexts, loops, and blocks of code corresponding to sub-paths between instrumentation points.

Some of the analysis information, such as worst-case execution time contributions, is relative to the *root function*, the main sub-program or function selected for analysis.

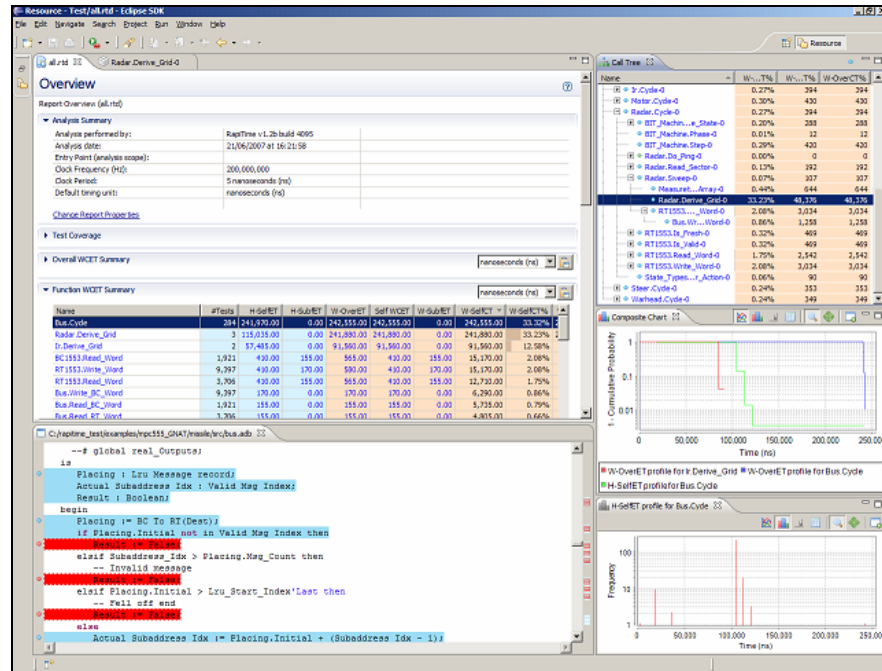


Figure 2: RapiTime Report Viewer

Report Data

Static data

The static information includes data such as the number of lines of code, loops, and sub-routine calls in each element, and the location of the code.

Code coverage

The code coverage information includes the number of times that each element was exercised during testing, and the number of instrumentation points in the element that were triggered during testing.

Worst-case execution time

The worst-case execution time information includes:

- The worst-case execution time of each element, broken down into the execution time of the element's own code, and that of any sub-function calls.
- The contribution of each element to the worst-case execution time of the root function, broken down into the contribution from the element's own code and its sub-function calls, enabling easy identification of worst-case hotspots.
- The worst-case frequency of each element, indicating how many times the element is executed on the worst-case path.

High water mark

The high water mark information refers only to data from the longest observed end-to-end run of the root function.

- The longest execution time of each element, found in the data for the longest observed execution of the root function, broken down into the execution time of the element's own code, and that of any sub-function calls.
- The contribution of each element to the longest observed execution time of the root function, broken down into the contribution from the element's own code and its sub-function calls.
- The frequency of each element, indicating how many times the element is executed when the root function takes its longest observed execution time.

Maximum (Minimum)

The maximum (minimum) execution time information includes:

- The maximum (minimum) observed execution time of each element, broken down into the execution time of the element's own code, and that of any sub-function calls.
- The frequency of each element, indicating the maximum (minimum) number of times that it was executed in any single run of root function.

Average case information

The average case execution time information includes:

- The average execution time of each element, broken down into the execution time of the element's own code, and that of any sub-function calls.

- The contribution of each element to the average observed execution time of the root function, broken down into the contribution from the element's own code and its sub-function calls.
- The frequency of each element, indicating how many times, on average, the element is executed per invocation of the root function.

Code metrics

RapiTime provides a number of execution time density metrics, for example worst-case execution time contribution divided by number of lines of code. These metrics are useful in selecting the best opportunities for optimisation – short sections of code that contribute disproportionately to the overall worst-case execution time.

Optimisation

Optimisation data provides information about:

1. The amount by which the execution time of each element would need to be increased/decreased for it to just be on the worst-case path. Here a positive value indicates the headroom to add additional functionality without increasing the overall worst-case execution time of the root function. By contrast, a negative value indicates the maximum reduction in the execution time of the element, whereby any further reduction would not reduce the overall worst-case execution time of the root function.
2. The amount by which the worst-case execution time of the root function would be decreased if the execution time of the element were optimised to achieve the execution time indicated in 1. above.

Budgets

RapiTime users can specify execution time budgets for elements, and whether or not each budget value should be used in the worst-case execution time calculations. RapiTime flags up if the maximum observed or worst-case execution time for the element exceeds its budget.

Comparisons

Comparisons between worst-case and high water mark values, highlighting where either testing or analysis needs to be improved.

Searching and sorting

The Report Viewer enables users to sort the RapiTime data by any of the given fields and to search for functions by name. This makes it easy to find those functions that contribute most to the worst-case execution time of the root function, or to find those functions that contribute most to the average case, or those functions that have the highest execution time density. Whatever the metric, it is possible to organise the RapiTime data appropriately.

Graphical information

In addition to the wide range of execution time data, RapiTime also provides the following information:

1. A call tree representation indicating which functions are on the worst-case path and those that are partially or fully untested.
2. Colour coded source, differentiating code that is on the worst-case path, code that has been tested but is known to not be on the worst-case path, and code that has not been tested.

3. Graphs of the execution time frequency distribution, and cumulative execution time probability distribution for each element.
4. Graphs of measured execution times against invocations of the element, and against invocations of the root function.

Worst-case execution times

RapiTime provides worst-case execution time information about each sub-program or function and all the contexts in which it is called. This makes it easy to identify those functions (and contexts) that contribute most to the overall worst-case execution time.

The function summary tables can be ordered by contribution to the overall worst-case execution time, highlighting those functions responsible for the majority of the overall execution time. Typically these functions represent only a small percentage of the total lines of code.

Links on the function names in the summary sections of the report navigate to detailed information about the contributions of individual blocks and loops within each function allowing engineers to drill down into worst-case hotspots.

Name	#Tests	H-SelfET	H-SubET	W-OverET	Self WCET	W-SubET	W-SelfCT	W-SelfCT%
Bus.Cycle	284	241,970.00	0.00	242,555.00	242,555.00	0.00	242,555.00	33.32%
Radar.Derive_Grid	3	115,035.00	0.00	241,880.00	241,880.00	0.00	241,880.00	33.23%
Ir.Derive_Grid	2	57,485.00	0.00	91,560.00	91,560.00	0.00	91,560.00	12.58%
BC1553.Read_Word	1,921	410.00	155.00	565.00	410.00	155.00	15,170.00	2.08%
RT1553.Write_Word	9,397	410.00	170.00	580.00	410.00	170.00	15,170.00	2.08%
RT1553.Read_Word	3,706	410.00	155.00	565.00	410.00	155.00	12,710.00	1.75%
Bus.Write_BC_Word	9,397	170.00	0.00	170.00	170.00	0.00	6,290.00	0.86%
Bus.Read_BC_Word	1,921	155.00	0.00	155.00	155.00	0.00	5,735.00	0.79%
Bus.Read_RT_Word	3,706	155.00	0.00	155.00	155.00	0.00	4,805.00	0.66%
BC1553.Is_Fresh	491	335.00	140.00	475.00	335.00	140.00	4,020.00	0.55%
BC1553.Is_Valid	3,408	335.00	140.00	475.00	335.00	140.00	4,020.00	0.55%
Measuretypes.encode.Bit4_Array	5	1,355.00	0.00	1,610.00	1,610.00	0.00	3,220.00	0.44%
Measuretypes.decode.Bit4_Array	3	1,040.00	0.00	1,510.00	1,510.00	0.00	3,020.00	0.41%
Steer.Extrapolate_Angle	1,204	370.00	100.00	480.00	375.00	105.00	3,000.00	0.41%
Clock.Read	3,333	100.00	0.00	105.00	105.00	0.00	2,835.00	0.39%
RT1553.Is_Fresh	331	335.00	140.00	475.00	335.00	140.00	2,345.00	0.32%

Figure 3: Example worst-case execution time summary

Worst-case hotspots

Unlike conventional code profiling techniques, the RapiTime report identifies the worst-case hotspots in a program from the point of view of execution time. That is the sections of code that contribute the most to the overall worst-case execution time. Conventional profiling techniques identify the sections of code that execute the most on average, which is very different. RapiTime also provides average case information, so users can see for themselves that the code that executes the most on average does not necessarily contribute the most to the worst-case; indeed it may not even be on the worst-case path.

RapiTime provides analysis of all worst-case hotspots. For each sub-program or function, the report provides detailed information of each loop and block of code, including their contribution to the overall worst-case execution time.

By inspecting the contribution each block makes to the overall worst-case, engineers can focus optimisation efforts where they will have the greatest impact. Hyperlinks on the block identifiers (filename and line numbers) navigate directly to the colour coded source for each function, highlighting the selected code.

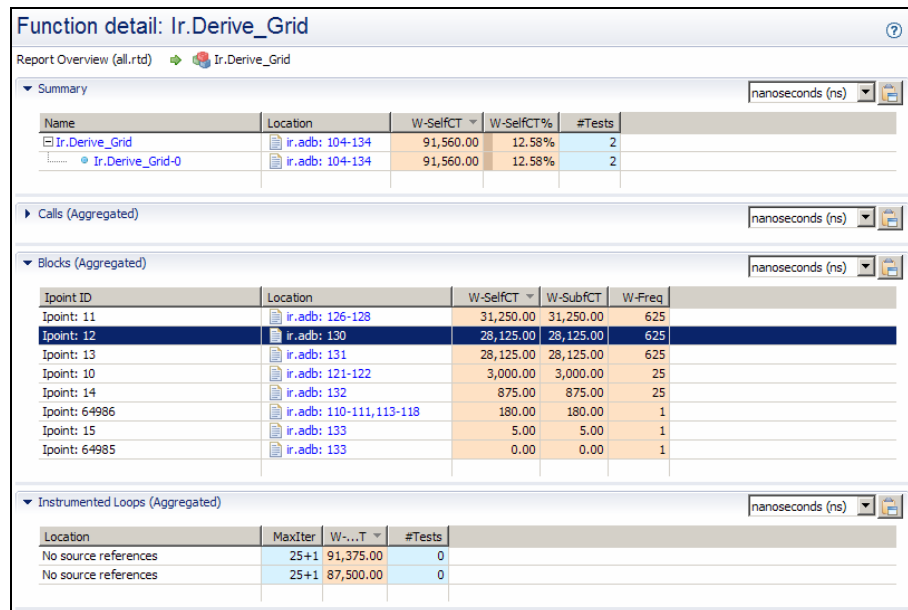


Figure 4: Contributions of individual blocks to the worst case execution time

Worst-case path

The RapiTime report helps engineers understand their systems better by visualising the critical paths in a program. This is of great benefit when the timing problems are so severe that the structure of the program needs to be changed to achieve an acceptable worst-case execution time.

RapiTime can highlight designs that, although having a good average-case performance, lead to an excessively long worst-case execution time and hence have the potential to cause intermittent timing problems. This knowledge helps engineers design more robust software.

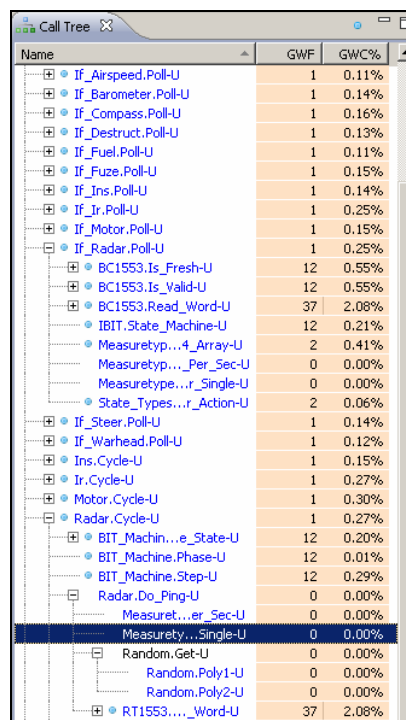


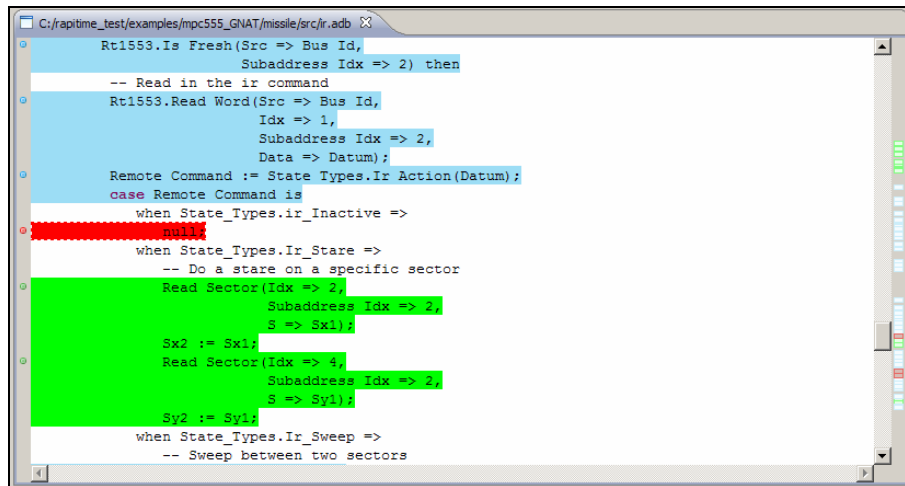
Figure 5: Call tree showing functions on the worst-case path (blue dots)

Colour coded source

The RapiTime report includes colour coded source, making it easy for engineers to identify:

- Code that has not been covered by the tests (Red).
- Code that is on the worst-case path and so contributes to the worst-case execution time (Blue).
- Code that does not contribute to the overall worst-case execution time (Green).

This information ensures that testing and optimisation efforts are highly effective.



```

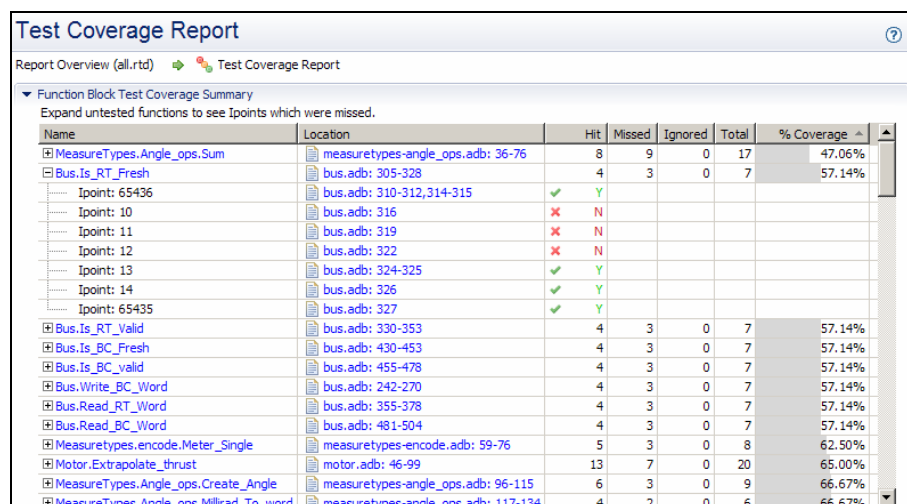
Rt1553.Is Fresh(Src => Bus Id,
                Subaddress Idx => 2) then
-- Read in the ir command
Rt1553.Read Word(Src => Bus Id,
                 Idx => 1,
                 Subaddress Idx => 2,
                 Data => Datum);
Remote Command := State Types.Ir Action(Datum);
case Remote Command is
when State_Types.ir_Inactive =>
null;
when State_Types.Ir_State =>
-- Do a stare on a specific sector
Read Sector(Idk => 2,
            Subaddress Idx => 2,
            S => Sx1);
Sx2 := Sx1;
Read Sector(Idk => 4,
            Subaddress Idx => 2,
            S => Sy1);
Sy2 := Sy1;
when State_Types.Ir_Sweep =>
-- Sweep between two sectors
    
```

Figure 6: Colour-coded source code

Code coverage

The RapiTime report identifies code that has not been exercised by the tests.

Untested blocks of code are identified by their file name and line numbers. Hyperlinks in the report navigate from this information directly to the colour coded source where untested code is marked in red.



Name	Location	Hit	Missed	Ignored	Total	% Coverage
MeasureTypes.Angle_ops.Sum	measuretypes-angle_ops.adb: 36-76	8	9	0	17	47.06%
Bus.Is_RT_Fresh	bus.adb: 305-328	4	3	0	7	57.14%
Ipoint: 65436	bus.adb: 310-312,314-315	✓	Y			
Ipoint: 10	bus.adb: 316	✗	N			
Ipoint: 11	bus.adb: 319	✗	N			
Ipoint: 12	bus.adb: 322	✗	N			
Ipoint: 13	bus.adb: 324-325	✓	Y			
Ipoint: 14	bus.adb: 326	✓	Y			
Ipoint: 65435	bus.adb: 327	✓	Y			
Bus.Is_RT_Valid	bus.adb: 330-353	4	3	0	7	57.14%
Bus.Is_BC_Fresh	bus.adb: 430-453	4	3	0	7	57.14%
Bus.Is_BC_Valid	bus.adb: 455-478	4	3	0	7	57.14%
Bus.Write_BC_Word	bus.adb: 242-270	4	3	0	7	57.14%
Bus.Read_RT_Word	bus.adb: 355-378	4	3	0	7	57.14%
Bus.Read_BC_Word	bus.adb: 481-504	4	3	0	7	57.14%
Measuretypes.encode.Meter_Single	measuretypes-encode.adb: 59-76	5	3	0	8	62.50%
Motor.Extrapolate_thrust	motor.adb: 46-99	13	7	0	20	65.00%
MeasureTypes.Angle_ops.Create_Angle	measuretypes-angle_ops.adb: 96-115	6	3	0	9	66.67%
MeasureTypes.Angle_ops.Millrad_To_word	measuretypes-angle_ops.adb: 117-134	4	2	0	6	66.67%

Figure 7: List of parts of the code that are untested

Execution Time Profiles

Each execution time or computed worst-case execution time value in the report is linked to an **Execution Profile**. Probability graphs and '1 – cumulative probability' graphs, otherwise referred to as *Exceedance Functions*, are used to illustrate the execution time distributions recorded in each Execution Time Profile.

The blue line on Figure 8 illustrates the **distribution of observed execution times** for a specific function. Given an execution time budget chosen from the x-axis, the corresponding value on the y-axis is the probability that a randomly chosen sample from the distribution of end-to-end execution times observed during testing exceeds that budget.

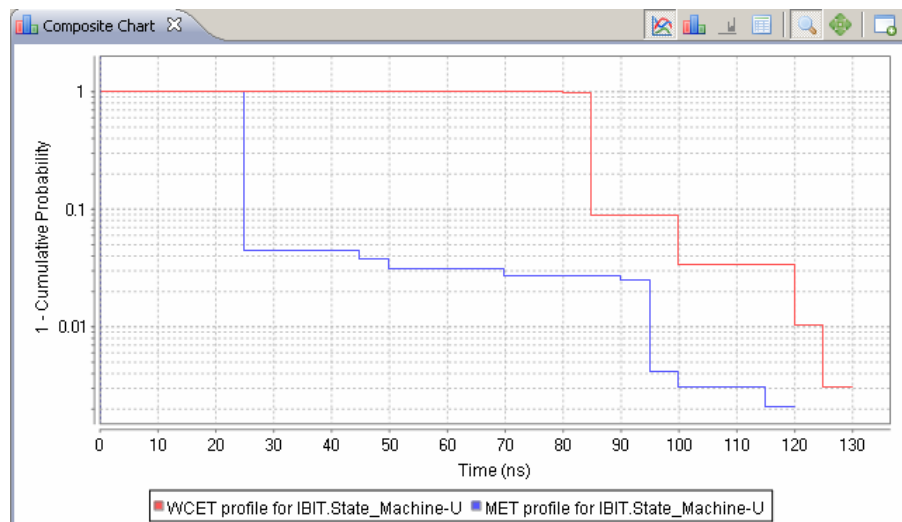


Figure 8: Distribution of observed execution times

Data can be read from the above graph in two ways.

- (i) Following a vertical line up from an execution time budget on the x-axis to the line on the graph and then horizontally to the y-axis means that you can read off the proportion of observations made during testing that exceeded the chosen budget. For example if the budget was 100ns, then the proportion of observations exceeding this budget was 0.003 (or 0.3%).
- (ii) Following a horizontal line across from a probability value (y-axis) to the line on the graph and then vertically down to the x-axis allows you to read off the execution time that a given proportion of observations exceeded. For example 0.03 (3%) of observations exceeded an execution time of 50ns.

The red line on Figure 8 illustrates the **variation in the execution time of the worst-case path** due to hardware effects such as cache, pipelines and branch prediction units. The graph shows for any given execution time budget chosen from the x-axis, the computed upper bound on the probability (read off the y-axis) that the execution time will exceed that budget when the worst-case path is taken.

Again data can be read from the graph in two ways.

- (i) Following a vertical line up from an execution time budget on the x-axis to the line on the graph and then horizontally to the y-axis means that you can read off the probability that the execution time will exceed the budget when execution follows the worst-case path.

For example if the budget was 120ns, then the probability that the execution time will exceed this budget when the worst-case path is followed is 0.01.

- (ii) Following a horizontal line across from a probability value (y-axis) to the line on the graph and then vertically down to the x-axis allows you to read off the execution time budget that the execution time of the worst-case path is likely to exceed with that probability. For example, there is a probability of approx. 0.09 that when the worst-case path executes, its execution time will exceed 85ns.

In addition to the '1 – cumulative probability' graphs, simple probability distributions are also provided for both measured and computed worst-case Execution Time Profiles.

Probability Distributions

An example probability distribution is shown in Figure 9 below. The probability distribution shows for a given execution time chosen from the x-axis, the computed probability that when the worst-case path is taken, it will have that execution time. Again this illustrates the variation in the execution time of the worst-case path due to hardware effects such as cache, pipelines, branch prediction units etc.

In the example shown below, the most probable execution time of the worst-case path is 85ns, which is expected to occur on approximately 90% of the occasions that the path is taken. The worst-case time for the worst-case path is 130ns, which is expected to occur on approximately 0.2% of the occasions that the path is taken.

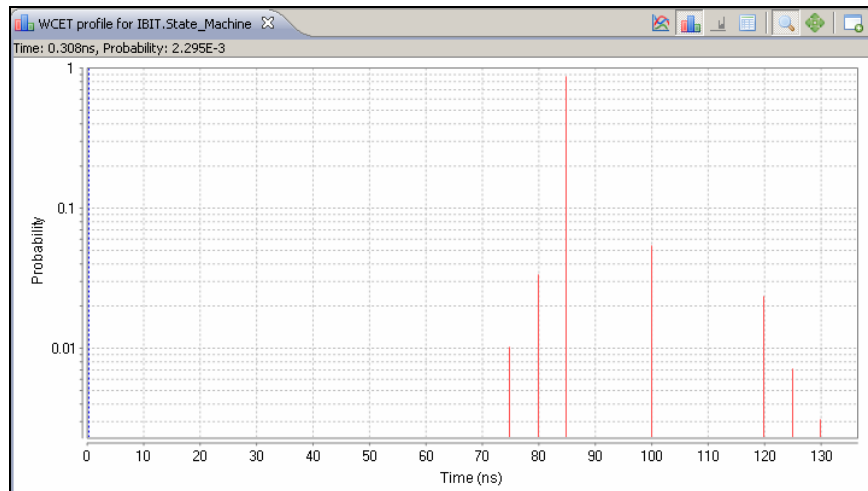


Figure 9: Probability distribution of the computed execution time for the worst-case path

Frequency Distributions

An example frequency distribution is shown in Figure 10 below. This frequency distribution is for the measured execution times of a function. The graph shows that the modal (most frequent) execution time was 25ns, occurring on approximately 900 invocations of the function. By comparison, the longest observed execution time was 120ns, which occurred on two invocations.

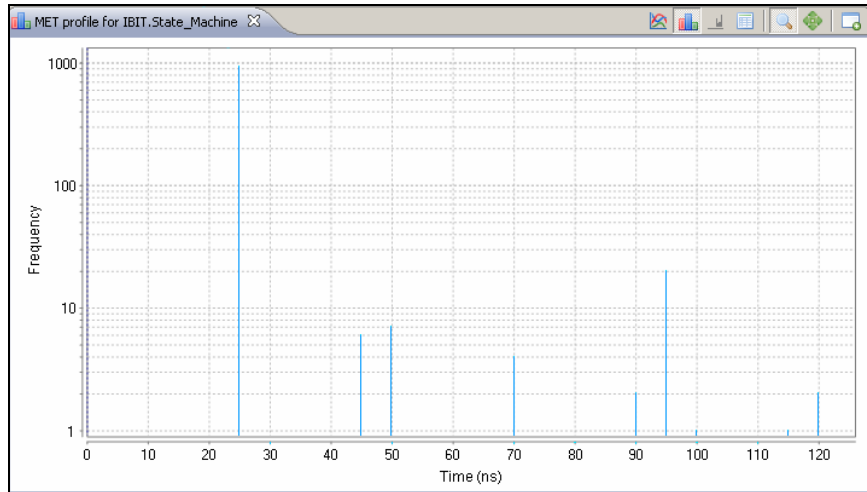


Figure 10: Frequency distribution of measured execution times

End-to-end execution time graphs

The RapiTime report provides graphs of the measured end-to-end execution time against invocation count for each function context.

An example graph of end-to-end execution times is shown in Figure 11 below. This graph illustrates that the code has a modal behaviour (in this case associated with a state machine) with longer execution times occurring on certain invocations. Code with this type of end-to-end execution time signature can often be re-arranged, moving code from the longer infrequently executed path to the shorter path and so reducing the worst-case execution time.

This type of graph shows end-to-end execution times split into two components; time attributed to code within the function itself (labelled 'Self') and time attributed to sub-function calls (labelled 'Sub-function'). (Note in the example below, the function does not make any sub-function calls).

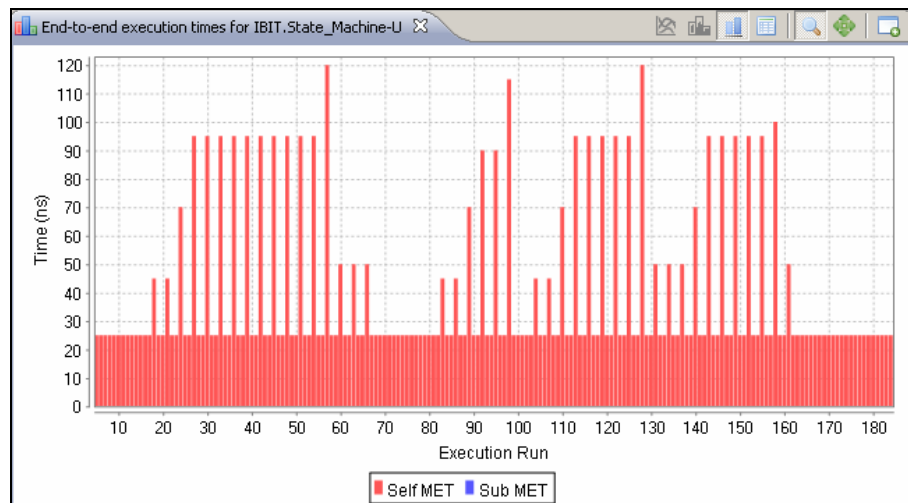


Figure 11: End-to-End Execution Time

4. Benefits of RapiTime

RapiTime is a toolset for performing worst-case execution time analysis, and performance profiling of real-time embedded systems. It uses information gathered during the testing process, along with path analysis, to produce execution time information for sub-programs, function calls, loops and basic blocks of code. The worst-case execution times computed take account of the worst-case input parameters and the worst-case hardware effects.

Using RapiTime engineers can determine **a wealth of execution time data**, including maximum, minimum and average execution times, high water marks, and worst-case execution times during unit testing. Potential timing glitches that would require days or weeks to track down during integration testing can be efficiently identified. This allows remedial action to be taken early preventing costly delays. The benefits are reduced time-to-market and a significant reduction in the number of timing problems going undetected.

RapiTime can identify code that is on the **worst-case path**. This is particularly useful when the worst-case execution time needs to be reduced. Engineers can look at the code that forms the worst-case path and direct their optimisation efforts where it is known to have a benefit. This reduces the effort required to resolve timing issues.

Unlike conventional code profiling techniques, RapiTime identifies the **worst-case hotspots** in a program from the point of view of execution time. That is the sections of code that contribute the most to the worst-case execution time. Conventional profiling techniques identify only the sections of code that execute the most *on average*, which is very different.

RapiTime's hotspot analysis ensures that attention is focused on improving those parts of the code that will have the maximum effect in reducing the worst-case execution time. This reduces the time spent optimising code to resolve timing issues, whilst eliminating unnecessary optimisation with its attendant maintenance headaches.

Targeted worst-case hotspot analysis makes it possible to extract the maximum performance from the most cost-effective processor variants, minimising unit costs in production.

As well as identifying targets for optimisation, RapiTime enables engineers to **answer what-if? questions**, quantifying the maximum performance gains obtainable by optimising selected software components. In addition, the headroom associated with each function can also be assessed, determining where additional code can be added without increasing the overall execution time.

RapiTime provides **code coverage** analysis identifying which parts of the code have been exercised during testing. This allows engineers to tell if the set of unit tests is incomplete and sometimes to identify code that simply should not be present! This has the benefit of increasing the quality of unit testing, reducing the number of functional and timing problems that make it through to the integration phase.

RapiTime helps engineers understand their systems better by **visualising the critical paths** in a program. This is of great benefit when the timing problems are so severe that the structure of the program needs to be changed to achieve an acceptable worst-case execution time. RapiTime can highlight designs that, although having a good average-case performance, lead to an excessively long worst-case execution time and hence have the potential to cause intermittent timing problems. This knowledge helps engineers design more robust software.

As well as calculating worst-case execution times, RapiTime also computes Execution Time Profiles. These profiles record the variation in the execution time of the worst-case path **due to hardware effects**. By examining the Execution Time Profiles engineers can see the likelihood that the worst-case path will result in an execution time greater than the specified constraint. If this probability is very small (e.g. 10^{-9}) then the software may be deemed to be acceptable. Knowing the difference between the predicted worst-case and that seen during testing allows engineers to be confident that the analysis is correct.

As software development is an iterative process and small changes in the source code can have a major impact on timing behaviour, RapiTime has been constructed as a automated tool chain that can be run in batch-mode and hence integrate into industry standard build processes. This simplifies the use of RapiTime during unit test reducing adoption costs.

RapiTime can also be used with a processor simulator rather than target hardware. When new processors are developed, it is sometimes the case that accurate simulations of the processor are available prior to the advent of fully functional silicon. If a simulator is available then using RapiTime engineers can examine the timing behaviour of the software before fully functional target hardware exists. This is of particular benefit to projects developing software and hardware in parallel.

RapiTime can automatically instrument and analyse industrial scale C and Ada programs comprising many hundreds of thousands of lines of code. RapiTime supports hardware trace capture via a Logic Analyser / trace board, with minimal instrumentation. The RapiTime roadmap includes support for probe-effect free tracing via Nexus and ARM RTM interfaces.

5. Conclusions

RapiTime addresses *the* fundamental requirement in developing reliable real-time embedded software on advanced microprocessors: the need to understand the execution time performance of the various software components. By providing this information, RapiTime enables engineers to use a systematic and scientific approach to ensuring that time constraints are met. In effect allowing them to engineer timing correctness into the system rather than spending a great deal of time and effort trying to get timing bugs out. This shortens time to market and reduces development costs making RapiTime a key asset in engineering embedded real-time systems.

For embedded technology manufacturers, the potential return on investment from utilising the technology that RapiTime brings is huge. This toolset offers the prospect of far fewer timing bugs going undetected through unit test and integration phases. Identifying timing issues early in development has the proven impact of reducing development cost, reducing time to market and enhancing the company's reputation through higher quality and more reliable products.

RapiTime is unique in utilising state-of-the-art mathematical techniques developed by leading researchers from the Real-Time Systems Group at the University of York. It combines the best features of the two major approaches to finding worst-case execution times: measurement and static analysis. This innovative approach means that RapiTime is the only toolset available that can provide worst-case execution time information for complex software on the latest generation of advanced processors.

Appendix: Comparison of RapiTime and other WCET methods

There are two traditional approaches to worst-case execution time analysis

1. Measurement
2. Static Analysis

This appendix reviews the strengths and weaknesses of these approaches and compares them to RapiTime.

Measurement

Measurement techniques insert profiling code into the software and measure the execution time of blocks of code during testing. This is the most common mechanism in commercial use today. It has the following advantages and disadvantages.

Measurement Advantages	Disadvantages
<ul style="list-style-type: none">• Provides a good indication of the range of execution times of the program.• A workable solution for simple processors running small programs.	<ul style="list-style-type: none">• Requires execution time measurements to be taken during testing.• Impact of adding instrumentation code.• Exhaustive testing is needed to ensure that the worst-case path is exercised. This becomes extremely time consuming and expensive as program complexity increases.• Provides no information on whether the worst-case path has been tested• Not sufficient for programs running on advanced processor architectures.

Static Analysis

Static analysis was developed as an alternative to measurement. Static analysis techniques address the problem of determining the worst-case execution time at two levels:

- Low-level analysis: a timing model of the processor is constructed taking account of the behaviour of pipelines, cache and other hardware features and their interactions. This model aims to provide a precise cycle accurate description of how long each machine instruction takes to execute, given the internal state of the hardware.
- High-level analysis: The worst-case path is computed by analysing the program, determining loop bounds and based on the information contained in the timing model the duration of the worst-case path is computed. This is done without running the code.

Static Analysis Advantages	Disadvantages
<ul style="list-style-type: none">• Can accurately determine upper bounds on the worst-case execution time for small programs running on simple processors.• Can identify the worst-case path.• Can determine the set of possible states of the processor at each machine cycle.• No measurements required.	<ul style="list-style-type: none">• The model used may not accurately reflect the real behaviour of the processor, undermining accuracy.• The complexity and effort required to build the timing model for each processor means long lead times in developing static analysis tools.• Custom hardware variants and different versions of silicon may have different timing behaviours rendering the models used by static analysis incorrect.• Using simplified models of complex features such as caches, pipelines, DMA and so on renders worst-case execution time bounds extremely pessimistic. (Orders of magnitude greater than the actual worst-case time).• Full details of modern processors are often not available making static analysis impossible.• No indication is given of how pessimistic the computed worst-case execution time is.

RapiTime

RapiTime is unique in that it integrates the **best features of both measurement and static analysis**. Measurement is the right tool to derive the actual execution time of small components of the software. Static analysis on the other hand is the right tool to construct a picture of the worst-case path through a piece of software. RapiTime combines the advantages of both previous approaches whilst avoiding their pitfalls.

RapiTime Advantages	Disadvantages
<ul style="list-style-type: none">• Computes the worst-case execution time bounds based on <i>actual</i> testing conditions not on extremely pessimistic assumptions.• Identifies code on the worst-case path.• Identifies hotspots within the worst-case path.• Enables visualisation of the worst-case and other paths.• A workable solution for advanced processors as an accurate timing model of the processor is not required.• Allows comparison between measured and computed worst-case execution time values• Determines the variability of program execution time due to hardware effects.• Provide worst-case execution time coverage information, used to determine if further testing and measurement is required.	<ul style="list-style-type: none">• Requires execution time measurements to be taken during testing.• Impact of adding instrumentation code.
