Hybrid measurement-based WCET analysis at the source level using object-level traces

Adam Betts[1], Nicholas Merriam[1], and Guillem Bernat[1]

1 **Rapita Systems Ltd.**
IT Centre, York YO10 5DG, UK
{abetts,nmerriam,bernat}@rapitasystems.com

Abstract

Hybrid measurement-based approaches to worst-case execution time (WCET) analysis combine measured execution times of small program segments using static analysis of the larger software structure. In order to make the necessary measurements, instrumentation code is added to generate a timestamped trace from the running program. The intrusive presence of this instrumentation code incurs a timing penalty, widely referred to as the probe effect. However, recent years have seen the emergence of trace capability at the hardware level, effectively opening the door to probe-free analysis.

Relying on hardware support forces the WCET analysis to the object-code level, since that is all that is known by the hardware. A major disadvantage of this is that it is expensive for a typical software engineer to interpret the results, since most engineers are familiar with the source code but not the object code. Meaningful WCET analysis involves not just running a tool to obtain an overall WCET value but also understanding which sections of code consume most of the WCET in order that corrective actions, such as optimisation, can be applied if the WCET value is too large.

The main contribution of this paper is a mechanism by which hybrid WCET analysis can still be performed at the source level when the timestamped trace has been collected at the object level by state-of-the-art hardware. This allows existing, commercial tools, such as RapiTime, to operate without the need for intrusive instrumentation and thus without the probe effect.

Digital Object Identifier 10.4230/OASIcs.WCET.2010.54

1 Introduction

The task schedule in a real-time system is responsible for allocating the CPU's time to a number of single-threaded tasks. All scheduling algorithms, and the schedulability tests that check their feasibility, assume that the **Worst-Case Execution Time** (WCET) of each task is available as a fixed value.

Deriving the *actual* WCET completely automatically with a tool is impossible in the general case, not least because such a tool would also solve the halting problem, see [12]. However, practical experience has shown that a useful *estimate* of the WCET can be derived using a tool with a little guidance from the user.

Most often, industry uses the WCET estimate to validate compliance of tasks with their execution-time budgets, assigned early in the development process to allow construction of the task schedule. This validation is done by recording end-to-end execution times under quite extensive and demanding testing conditions; the longest time is generally referred to as the **High-Water Mark Time** (HWMT). The underlying premise of this approach is that the testing regime is representative of real system operation and that, with enough testing, the HWMT lies in close proximity to the actual WCET. However, there are a number of disadvantages of such a simplistic approach:

10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper; pp. 54–63

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- No understanding is given regarding the path through the software which leads to the HWMT and engineers therefore lack insightful knowledge of where time is spent. This information is crucial in optimisation efforts, particularly when the HWMT exceeds the budget of a task.
- Although pessimism is eliminated (either the test harness captures the actual WCET or an uncomputable degree of underestimation exists), a bound on the actual WCET cannot be guaranteed without full path coverage and, perhaps more significantly, *full state coverage* at the hardware level. Acknowledging this issue, industry adds some percentage to the HWMT in order to bypass any underestimation, and considers this as the WCET estimate. Even with this additional step there is still no guarantee that the actual WCET is bounded; it could even lead to underutilisation of system resources if, in fact, the actual WCET has been captured but the safety margin is too excessive.
- In safety-critical systems, MC/DC [5] is normally the coverage metric that needs to be satisfied before testing halts. In effect, properties affecting the execution time, such as hardware effects and loop iterations are neglected. This is a considerable anomaly given the increasing complexity of modern embedded hardware and the jitter it introduces into the execution time profile of a task.

An alternative to high-water marking, offered by **Hybrid Measurement-Based Analysis** (HMBA), is to measure execution times of small program segments and then combine these data in a final calculation stage, to deliver a WCET estimate. The necessary measurements are extracted from a timestamped trace of execution, which a running program emits when *instrumented*. Because instrumentation code is traditionally compiled into the executable, it has been *intrusive* by nature. On the one hand, this provides portability and flexibility. On the other hand, it manifests the **probe effect** where normal (i.e. without instrumentation) register and cache usages are displaced, a timing penalty is incurred, and overall code size increases.

Recent years, however, have seen an increasing number of embedded micro-controllers feature some level of trace capability (Nexus [1] adopted by Freescsale, ARM Embedded Trace Macrocell [8], and others) without the requirement for software instrumentation. A suitable debugger can then reconstruct exactly the sequence of instructions executed together with timestamps. This trace of instructions and timestamps provides an ideal foundation for WCET analysis at the object-code level, having great detail while eliminating the probe effect.

But object-level WCET analysis has two major drawbacks. First, many current HMBA techniques [6, 7, 3], and commercial HMBA tools such as RapiTime, derive their program model from hierarchical source code structures. They cannot, therefore, directly analyse the hardware-generated trace of object code. Second, even if that first obstacle were overcome, it remains expensive for engineers to interpret the results, such as which code is on the worst-case path, since their expertise concerning the application is usually at the source code level.

In this paper, we present a mechanism by which HMBA can still be performed at the source level when the timestamped trace has been collected at the object level by state-of-the-art hardware. This allows existing HMBA techniques to operate without the need for intrusive instrumentation and thus without the probe effect.

The remainder of this paper is structured as follows. Section 2 gives more background on HMBA, in particular discussing mechanisms available to obtain timestamped traces. Section 3 then shows how an object-code trace can be utilised at the source level and discusses future issues to be resolved. Next, Section 4 demonstrates the improvement gained by the proposed

method over existing intrusive instrumentation. We finally conclude in Section 5.

2 Background and Related Work

2.1 Instrumentation and Tracing

As noted above, HMBA relies on **instrumentation points** (ipoints) to collect measurements. Upon execution during testing, these ipoints emit (integer) identifiers and are time stamped accordingly, resulting in a timestamped trace of execution. The trace is subsequently parsed to extract both the WCETs of program segments and other path-related information such as loop bounds (if required).

There are a number of mechanisms available to generate and extract traces:

- Simulation: Cycle-accurate simulators, such as SimpleScalar [2], allow individual instructions to be traced (through the program counter) whereby the simulator provides the time stamp. However, because a simulator is a hardware model, this method encounters problems associated with producing a cycle-accurate processor model. This is often very difficult due to missing or incorrect information [9].
- Software: A tailored ipoint routine is inserted into the source code at particular locations. When an ipoint is hit during execution, it is time stamped on target; traces are stored internally in a memory buffer to be downloaded on test completion. The advantage of this approach is that porting to new architectures is relatively straightforward. However, this results into the probe effect.
- Software/Hardware: This is similar to software only instrumentation, except that the ipoint routine writes its identifier to an I/O port on target. The port is monitored by an external capture device, e.g. a logic analyser, which timestamps ipoints off target as they appear and also serves to store the timing traces. Penalties associated with the probe effect are minimised because the ipoint routine can be reduced to a few instructions. However, the target must have available and accessible pins to emit the data, which is not always practical with more advanced processors. It may also be necessary to disable the cache so that the data written by the ipoint routine is observed on the bus.
- Hardware: The probe effect can be prevented with the assistance of on-chip debug interfaces. In these cases, the trace data are either written to an on-chip trace buffer for subsequent download, or exported directly in real time through an external port. In order to limit the size of traces, only the program flow discontinuities are monitored, i.e. taken jumps. However, bandwidth remains the major technical obstacle because the port or debugger must keep pace with the rate at which trace data are produced; otherwise, blackouts arise in which parts of a timing trace are overwritten and essentially lost.

There is clearly a trade-off in using any of the above trace generation methods. On the one hand, source-level instrumentation provides greater flexibility and is often the most convenient, but it is handicapped by the probe effect. On the other hand, less intrusive instrumentation normally demands some type of hardware support.

2.2 Hybrid Measurement-Based Analysis

Although end-to-end testing and HMBA are closely linked, in that they both utilise test vectors to stress execution times, the latter has the following principal advantages:

- Piecing together measured execution times of small program segments with path-specific information in the WCET calculation alleviates the test harness in two ways. First, it eliminates the requirement to find a test vector that causes all loops to iterate through

their maximum bound *simultaneously*. By using a program model, HMBA instead allows loop bounds obtained from SA or end users to be included *a posteriori* to the measurement stage. Second, the test harness need not attempt to trigger the WCET of program segments in the *same* run, as HMBA multiplexes this information from all executions before carrying out the WCET computation. Obviously, coverage remains an issue since the units of computation must be stressed adequately enough to represent worst-case behaviour, although how this is realised is an open research question.

- Increasing the level of instrumentation beyond the start and end of the program allows valuable information to be extracted, including: the functions executed most frequently on the path leading to both the HWMT and the computed WCET estimate; the number of observed loop iterations; which sections of code have been covered by the test harness. As mentioned above, this information is essential to engineers as it provides insights into the non-functional properties of the code.

Once the timestamped trace has been generated, the calculation engine is tasked with producing a WCET estimate; how this is done depends on the type of program model. Much of the research on calculation engines has assumed a graph-based program model, such as the **Control Flow Graph** (CFG), chiefly because CFGs can easily model arbitrary control flow, including `goto`-littered code; comparatively, the **Abstract Syntax Tree** (AST), which is the *de facto* tree-based program model, fails or struggles to do so as the hierarchical relation between program segments is absent.

However, tree-based approaches based on the AST retain a number of advantages. First, in comparison to the linear programming [13] and path-based [14] calculations of graph-based program models, low computational complexity is incurred. Efficiency of the calculation is a specific concern when WCET analysis is integrated into an interactive environment [10] or when the system has a large code base. Second, in addition to integral values, execution time profiles derived from measurements can be combined to produce probabilistic WCET estimates [3, 4]. Third, it is straightforward to relay the longest path returned by the calculation engine onto the source code.

3 Object-Code Tracing

The normal work-flow of a source-level HMBA tool is to insert ipoints into the code while the program model, i.e. the AST, is being constructed. As ipoints are positioned, they are also assigned identifiers. When executed, these ipoints produce a trace of the form `(identifier, timestamp)`. However, the hardware tracing mechanisms mentioned in the previous section instead provide a trace of the form `(address, timestamp)`, where `address` is the destination of a branch; that is, the identifier of an ipoint is an address. Thus a source-level HMBA tool is usually unable to consume a trace in this format because there is no correspondence between the ipoint identifiers assigned and the addresses observed.

Thus the first issue to overcome is the ability for each inserted ipoint to use an address as its identifier. This can be done using global assembly labels[1]. That is, global assembly labels are generated for each inserted ipoint using a macro, an example of which appears in Figure 1. The resulting addresses of the labels are then passed back to the HMBA tool so that it can replace the original identifiers of ipoints. With this small step, the HMBA tool is

[1] Such a feature has already been implemented by RapiTime in the MERASA project — see *http://www.merasa.org*

```
#define RPT_Ipoint( I )                                    \
({ asm volatile (".globl __rpt_ipoint_" # I "\n"           \
                 "__rpt_ipoint_" # I ":" );                \
 })
```

Figure 1 C Macro to Generate Assembly Label Ipoint

able to parse a trace in `(address, timestamp)` format and therefore perform the WCET analysis.

It would appear that debug information generated by the compiler could instead be used to map traces to the source level, rather than global assembly labels. However, in industry, debug builds are typically only available very early in the development process; this solution is independent of debug information being available.

The second issue is that hardware debug interfaces only record a sequence of branch destinations, and not every address which corresponds to an ipoint label. The solution is to analyse the disassembly for branch instructions and then interpolate the missing instructions from the trace, recording only those that correspond to ipoint addresses. With both of these steps, therefore, a HMBA tool is able to consume an object-level trace as produced by a hardware debug interface.

3.1 Discussion

There are several outstanding issues with the proposed mechanism as summarised in the following discussion.

3.1.0.1 Processor limitations:

Evidently, our technique is limited to processors that provide tracing facilities. This implicitly excludes high-end PowerPC architectures which are widely used in the avionics industry and where there is a strong need for WCET analysis for certification reasons.

3.1.0.2 Loop unrolling:

Optimising compilers often unroll loops [11], in effect replicating the body of a loop a particular number of times and adjusting the control logic as necessary. This would generate multiple copies of a global label, which is illegal assembly, and consequently such optimisations have to be disabled. In any case, such optimisations are normally disabled in safety-critical systems.

3.1.0.3 Ipoint locations:

Knowledge of the object code also assists in improving the accuracy of source-level instrumentation. For instance, consider the source code in Figure 2. Here the intention is to observe the start execution time of `check_sum` with the ipoint `2147483643` (through the function `RPT_Ipoint`). However, in the compiled object code, shown in Figure 3, this ipoint appears at address `003f98b4`, which is six instructions after the start of `check_sum`. This can be resolved by instead assigning the start address of the basic block (which contains the ipoint label) as the ipoint identifier, rather than the raw address of the compiled label. In this example, therefore, the ipoint should be assigned identifier `003f989c`.

```
Uint8 check_sum( Uint8 * msg, Uint8 len )
{
  RPT_Ipoint( 2147483643) ;
  Uint8 c, i;
  c = 0xAA;
  ...
```

Figure 2 Example Instrumented C Source Code

```
003f989c <check_sum>:
  3f989c:       94 21 ff d0     stwu    r1,-48(r1)
  3f98a0:       93 e1 00 2c     stw     r31,44(r1)
  3f98a4:       7c 3f 0b 78     mr      r31,r1
  3f98a8:       90 7f 00 18     stw     r3,24(r31)
  3f98ac:       7c 80 23 78     mr      r0,r4
  3f98b0:       98 1f 00 1c     stb     r0,28(r31)

003f98b4 <__rpt_ipoint_2147483643>:
  3f98b4:       38 00 ff aa     li      r0,-86
  3f98b8:       98 1f 00 09     stb     r0,9(r31)
  3f98bc:       38 00 00 00     li      r0,0
  3f98c0:       98 1f 00 08     stb     r0,8(r31)
  ...
```

Figure 3 Example Instrumented Disassembly

Note that, when the intention of the ipoint is to record the time at which flow returns to the calling function, it is more appropriate to use the last address of the containing basic block as the identifier of an ipoint.

The next step in this work is to distinguish between ipoints corresponding to the beginning and end of function execution and all others, assigning each of the former its optimal identifier.

3.1.0.4 Trace size:

Tracing at the object-level typically results in very rapid generation of data. For instance, a processor running at 200MHz, with an average of one branch every 10 instructions and using 64 bits to record each branch timestamp, would generate 160MB in one second. A one-hour test would generate around half a terabyte of data. Despite the fact that storage costs continue to fall, object-level trace data remains difficult and expensive to archive and analyse. A key step forwards is to interpret the data in real time, removing the need to store all the trace data.

In 2010, Rapita Systems Ltd is starting a development project, assisted by the UK regional development agency Yorkshire Forward, to bring together the latest hardware and software techniques to allow real-time analysis of trace data.

4 Evaluation

To show the benefit of the proposed approach, the techniques described in the previous section were implemented in the RapiTime toolchain.

We analysed a synthetic in-house benchmark of 226 lines of C code, comprising six functions, whose functionality is to receive and respond to a number of different kinds of messages. The benchmark was instrumented using the C instrumenter of the RapiTime toolkit, which inserted a total of 28 ipoints. Compilation of the instrumented source was done through a GCC cross-compiler for the PowerPC architecture.

The binary was then wrapped in a test harness and executed on a Freescale MPC565 with a Wuerz evalutation board. The timestamped trace was obtained through a Lauterbach PowerTrace debugger via a Nexus debug port. Two such traces were obtained: one using an intrusive tracing mechanism and the other using the approach presented in this paper.

Each of the traces were parsed by RapiTime to produce a WCET estimate and other timing information for each function. A screenshot of the RapiTime report viewer showing results of the analysis appears in Figure 4. The most important information appears in the two stacked views in the centre:

Top view: This shows various execution time data, such as the longest execution time and the WCET estimate. The root function under analysis is `message_receive`: the highlighted times show its longest end-to-end execution time.

Bottom view: This displays the source code and the relative locations of ipoints. In Figure 4, there is an ipoint at the start of `message_receive` with identifier `4168328`, corresponding to the address `3f9a88`.

A more thorough presentation of the results appears in Table 1, which shows the WCET estimate for each function in the application obtained from object-level and source-level instrumentation. As can clearly be observed, every function benefits from the probe-free instrumentation. For the root function `message_receive`, the WCET is reduced from 20.339 microseconds to 11.863 microseconds, the former owing to instrumentation overheads.

This amounts to a ≈ 42% decrease in the overall WCET estimate when using object-code instrumentation, which is a substantial benefit for a relatively small application, thus underlining the benefit of adopting hardware debug interfaces in HMBA.

Function	WCET Object Instrumentation	WCET Source Instrumentation
`message_receive`	11.863	20.339
`check_sum`	7.232	8.732
`count_set_bits`	4.940	9.232
`process_in_buffer`	6.858	11.214
`send_message`	0.746	0.804
`simulate_save_to_flash`	0.457	0.839

Table 1 WCET Estimates (in Microseconds) for Each Function in Analysed Program

It should be noted, however, that the overheads of the intrusive instrumentation are unusually high in this application. This is because the code base itself is fairly trivial, containing several tight loops. Future work will assess the impact of the proposed technique on a range of more realistic and, in particular, larger examples.

5 Conclusions

State-of-the art HMBA tools such as RapiTime automatically insert software instrumentation points into the source code while constructing the program model on which they base the WCET calculation. The biggest drawback of this approach is that the instrumentation adds an overhead. The probe effect is a key weakness of HMBA tools and an objection to their uptake, with the overheads not only disturbing the accuracy of the results but even making the software too slow and/or large to run in the target test environment.

This can be circumvented by instead utilising the trace facilities provided by hardware debug interfaces. Using object-level tracing, however forces the analysis to that level. This is especially inconvenient for commercial tools, which normally present the calculated data at the source level, since this is where most engineers retain expertise. This paper presented a solution to the problem by showing how to match the addresses of instrumentation points in the program disassembly to source lines.

We showed some initial results using the presented approach: for a small application, there was a ≈ 42% reduction in the WCET estimate in comparison to using intrusive instrumentation. The ability to perform analysis without the probe effect is a significant advance in the maturity and applicability of HMBA tools and plays a vital role in their wider adoption in industry.

Acknowledgements

This work has been supported by the MERASA STREP-FP7 European Project under the grant agreement number 216415.

References

1 The Nexus 5001™Forum. http://www.nexus5001.org, April 2010.
2 T. Austin, E. Larson, and D. Ernst. SimpleScalar: an Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, February 2002.

3 G. Bernat, A. Colin, and S. M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proceedings of the 23rd Real-Time Systems Symposium (RTSS'02)*, December 2002.

4 G. Bernat, M.J. Newby, and A. Burns. Probabilistic Timing Analysis: an Approach using Copulas. *Journal of Embedded Computing*, 1(2):179–194, November 2005.

5 J. J. Chilenski and S. P. Miller. Applicability of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal*, 9(5):193–200, September 1994.

6 A. Colin and G. Bernat. Scope-tree: a Program Representation for Symbolic Worst-Case Execution Time Analysis. In *Proceedings of the 14th Euromicro Conference of Real-Time Systems (ECRTS'02)*, July 2002.

7 A. Colin and S. M. Petters. Experimental Evaluation of Code Properties for WCET Analysis. In *Proceedings of the 24th Real-Time Systems Symposium (RTSS'03)*, December 2003.

8 ARM development tools. http://www.arm.com, April 2010.

9 J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, April 2002.

10 T. W. Harmon. *Interactive Worst-case Execution Time Analysis of Hard Real-time Systems*. PhD thesis, University of California, Irvine, 2009.

11 S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

12 P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1(2):159–176, September 1989.

13 P. Puschner and A. V. Schedl. Computing Maximum Task Execution Times - A Graph-Based Approach. *Real-Time Systems*, 13(1):67–91, July 1997.

14 F. Stappert, A. Ermedahl, and J. Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *Proceedings of the international conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'01)*, November 2001.

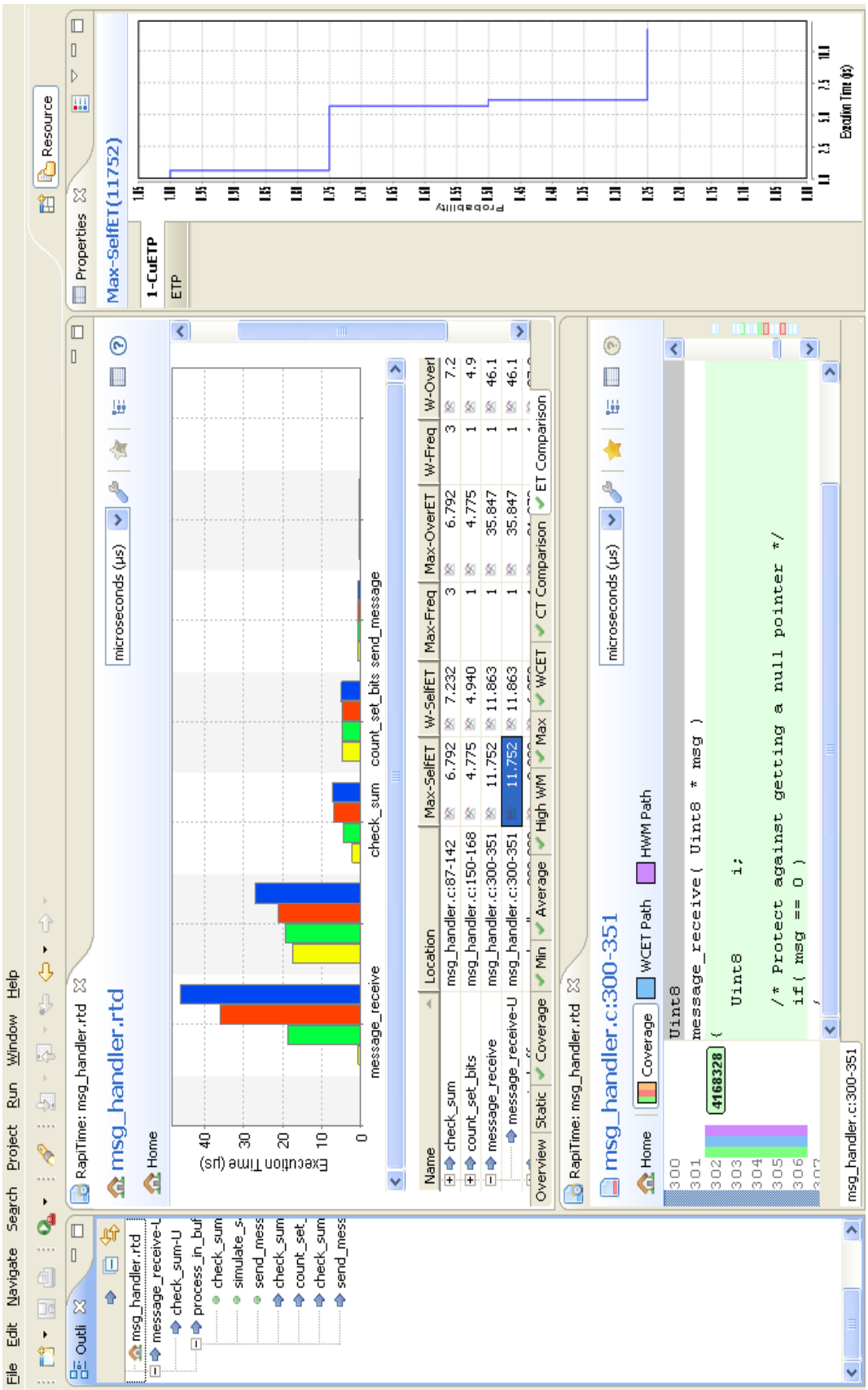

Figure 4 Screenshot of RapiTime Report.