Identifying Opportunities for Worst-Case Execution Time Reduction in an Avionics System

Guillem Bernat, Robert Davis, Nick Merriam, John Tuffen
Rapita Systems Ltd. IT Centre, York Science Park. YO10 5DG UK.
Tel: +44 1904 567747; Email: bernat@rapitasystems.com

Andrew Gardner, Michael Bennett, Dean Armstrong
Hawk Mission Systems, BAE Systems. Brough. HU15 1EQ.

Abstract

This paper describes the results of a study that identified opportunities for worst-case execution time reduction in the Operational Flight Program (OFP) software of BAE Systems' Hawk Mission Computer. The RapiTime toolset was used to provide the execution time analysis information required to target optimizations where they would be most effective. Potential optimizations were identified for worst-case hotspots at three levels: design level, sub-program level and low-level. These hotspots accounted for only 1.2% of the lines of code but contributed 29% of the overall execution time. Focused optimizations on these hotspots resulted in a 23% reduction of the overall execution time for the analysed code.

Keywords: Worst-case execution time analysis, WCET, real-time, Ada, Avionics, performance measurement.

1 Introduction

In a real-time system, it is important to guarantee both functional and non-functional requirements, in particular timing correctness. Functional verification is a well understood process that includes requirements capture, design, implementation, review and testing. However, the process for timing verification is less well understood. Current trends towards more complex software and more advanced hardware have resulted in the need to spend significant time and effort in understanding, verifying and improving the timing behaviour of systems.

One such complex system is the Operational Flight Program for the BAE Systems' Hawk Mission Computer[1]. The Operational Flight Program software is written in Ada and consists of hundreds of thousands of lines of code divided into 25 partitions, themselves divided into tasks, executed in a cyclic schedule. In 2006, the current system was running close to capacity, in terms of available execution time. In order to provide capacity for new functionality, an internal activity was commenced to identify optimization opportunities that would reduce the worst-case execution time of the system by at least 10%; thus avoiding the need for an expensive hardware upgrade.

Figure 1 Hawk fast jet trainer

Manual identification of optimization opportunities in such a large system is a daunting task. In this case, the system was developed over a number of years by a large team of engineers. Its sheer size and complexity makes it difficult if not impossible for a single engineer to gain an in depth understanding of the entire system. Further, there was no clear information on which components actually contributed to the overall worst-case execution time.

Initial efforts at understanding the timing behaviour of the system were based on determining the execution time of each partition via *high water marks* measured on the target microprocessor.

A typical situation was that painstaking optimization of a sub-program would result in unit tests showing a significant reduction in execution time whilst making little or no impact on the overall high water mark. In contrast, simpler more minor optimizations could sometimes have a significant impact, reducing the high water mark readings. This occurred when, in the first case, the code was not actually on the worst-case path, and in the second case, when the sub-program was both on the worst-case path and called a large number of times on that path.

Initially, there were no mechanisms in place to identify which sections of code were on the worst-case path, thus

[1] Hawk is a fast jet trainer, famously flown by the Red Arrows display team [1]

the selection of which sub-programs to optimize was effectively an educated guess.

Note that conventional profiling mechanisms are not particularly useful in the solution of this problem as they identify code that contributes most to the average execution time, which may be completely different from the code that is on the worst-case path and contributes most to the worst-case execution time.

Rapita Systems, together with the Hawk Integration Team, investigated how the problem of identifying the correct targets for optimization could be solved using the RapiTime worst-case execution time and performance analysis toolset. The study also aimed at evaluating the capabilities of the RapiTime toolset to cope with very large Ada programs, and its ability to summarise execution time data so that optimization opportunities could be easily identified and prioritised.

Using RapiTime, the joint project team made up of Hawk Integration Team and Rapita Systems engineers was able to successfully analyse the selected subset of the system. RapiTime was used to identify the small number of sub-programs that contributed heavily to the worst-case execution time. This code was inspected and, using the worst-case hotspot information provided by RapiTime, key code constructs targeted for optimization. These optimizations were classified as: low-level, sub-program level and design level. The best candidates for optimization were prototyped and the new system analysed to verify the effectiveness of the changes. The results of these optimizations are reported in Section 5.

The remainder of the paper is organised as follows: Section 2 describes in more detail the system under study; the Operational Flight Program of the Hawk Mission Computer. Section 3 provides a brief overview of the RapiTime toolset. Section 4 gives a classification of optimization opportunities, with examples of constructs found at each level. Section 5 reports the main results of the study and finally, Section 6 provides a summary and conclusions.

2 Hawk Operational Flight Program

The system under analysis is a subset of the Mission Computer of the HAWK aircraft. The Mission Computer provides the graphics for all six cockpit display panels and head-up displays amongst other functionality.

2.1 Architecture

The software for the Operational Flight Program (OFP) is written in Spark Ada and comprises several hundred thousand lines of source code. The OFP is divided up into 25 partitions executed as part of a cyclic schedule.

The system runs on a microprocessor board based on the PowerPC MPC7410 running at 500 MHz. This processor has a significant number of complex hardware features. It includes a two level cache: separate data and instruction level-1 caches of 32 Kbytes each, with pseudo random replacement policy, and a 2 Mbytes integrated level-2 cache. It also has a branch prediction unit, a 9-stage pipeline, multiple instruction fetches per cycle, a floating-point unit, two integer units and a performance counter unit. The board has 512 Mbytes of RAM.

2.2 Previous approach to timing analysis

Prior to the study, the method used to obtain timing information about the Hawk OFP software involved taking average and ‘high water mark’ measurements of the time each partition or task took to execute during testing or normal operation. High water marking was implemented by simply recording the time at the start and end of the partition (or any arbitrary section of code) at each execution and subtracting these two values to determine the execution time. This value was then compared to the largest value found so far and if greater, the new value was kept. At the end of execution these high water mark values could be examined.

With this process, no on-target code coverage information was available, so it was not possible to determine how much of the code was actually exercised by the tests. One potential risk was that the real worst-case execution time could be much longer than the high water mark value due to code that had not been exercised.

The high water mark approach had the further disadvantage that the large number of software components involved were unlikely to take their worst-case execution times together. Hence the high water mark times ran a significant risk of being optimistic i.e. less than the real worst-case time, even for the set of sub-programs that were fully exercised by the tests.

Unfortunately, often the first indication of a problem with the timing behaviour of the system was when it overran its timing budget during the latter stages of testing. At this point, manual intervention to discover which components were the main contributors to the overrun required additional effort and resources, resulting in potentially expensive and time-consuming delays.

The RapiTime toolset enabled a systematic, efficient and effective approach to be taken in investigating the timing behaviour of the system and identifying the worst-case hot-spots that were the major contributors to the overall execution time. The processor of analysing the system using RapiTime is described in the next section, follow a brief overview of the RapiTime toolset.

3 RapiTime

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.

RapiTime [3] is an analysis toolset that provides a *unique* solution to the problem of determining worst-case execution times for complex software running on advanced microprocessors.

RapiTime uses an innovative combination of three techniques:

1. 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 in a way that accurately captures the execution time variation on individual paths due to hardware effects.

The RapiTime toolset can be used to:

- Determine worst-case execution times for each software component, from complex programs down to basic blocks of code.
- Identify code that is on the worst-case path.
- Provide detailed analysis of *worst-case hotspots* and their contribution to the overall worst-case execution time.
- Provide code-coverage metrics ensuring confidence in the analysis results.
- Generate Execution Time Profiles illustrating the variability in execution times due to hardware effects.

RapiTime not only computes maximum values of execution times, but also their full distribution (in a statistical sense), thus capturing the variability of execution times due to hardware effects. This analysis is based on state-of-the-art statistical methods for modelling statistical dependencies known as the theory of Copulas [2].

3.1 RapiTime analysis process

The RapiTime toolset integrates into the standard software build process. As part of the study, makefile scripts were modified to include the following extra steps required for RapiTime to analyse the system:

1. Build executables for analysis. A special build was produced that had the subsystem under analysis automatically instrumented as well as including a lightweight tracing library for the MPC7410.
2. Structural analysis. The make process was modified to include an extra step, allowing the RapiTime tools to extract the structure of the code. The structure was derived from analysis of the disassembled executable, capturing the transformations that the compiler introduced into the code.
3. Testing and trace generation. This stage involved running the application on the target microprocessor under a set of test scenarios, collecting the trace data and downloading it from the target. Several options exist to extract the timing data from the target. In this case, the standard debugger was used to download a memory dump of the area in memory where the trace data was stored.
4. Trace processing. RapiTime trace manipulation tools were used to extract timing traces from the memory dump, to filter out events of no interest, to compress the data, to fix timer wraparounds, and finally, to derive a set of measured execution time profiles for each sub-program, loop and basic block.
5. Worst-case execution time calculation and report generation. The final stage was the WCET calculation using the measured data from individual sub-paths and structural information about the code. Additional annotations were used at this stage to guide the calculation process. The results were formatted as a set of easy to navigate reports.

The information in the RapiTime reports was used to identify those sub-programs that contributed most to the worst-case execution time and thus selection the most promising opportunities for optimization. Opportunities for optimization were considered at three levels: design-level, sub-program level and low-level. These three categories are described in more detail in the next section.

4 WCET optimizations

Optimization is a compromise of several factors, in particular: time, space, readability, maintainability and effort. For example, some optimizations may lead to code structures that are very hard to maintain but result in a significant reduction in execution time. The key to any optimization strategy is to prioritise those optimizations where the minimum effort (and the minimum amount of compromise in other factors) is required to gain the maximum benefit in execution time reduction.

Profiling is not worst-case. 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 lines of code that contribute the most to the worst-case execution time. Conventional profiling techniques identify the lines of code that execute the most *on average*, which is very different. For example, in the following code:

```
if rare_condition_of_error then
  long_computation_to_fix_the_error;
else
  short_normal_operations;
end if;
```

A profiler would indicate that most of the time is spent performing the short_normal_operation, missing the fact that in the worst-case, the path to follow is through long_computation_to_fix_the_error. Any optimization performed on the else branch would have no impact at all on the overall worst-case execution time.

For example, the following code is optimized for the average case:

```
if most_of_the_times then
  short_execution_time;
elsif less_regularly then
  medium_execution_time;
elsif very_infrequently then
  long_execution_time;
end if;
```

However, in the worst case the code needs to do the three tests, an optimization for the worst-case would instead be:

```
if very_infrequently then
  long_execution_time;
elsif most_of_the_times then
  short_execution_time;
elseif less_regularly then
  medium_execution_time;
end if;
```

In this case, only one test is done on the worst-case path.

On a similar note, deciding with bit of code to lock in the cache may also be different for worst-case optimization than for average case optimization. For example in the previous example if long_execution_time took a very long time but actually used few cache lines, it would be a good candidate to be locked in cache.

4.1 Level of focus

A key focus of the optimization process is to identify the level at which to perform the optimization. Optimizations can be classified at three levels: design-level, sub-program-level and low-level.

Design-level optimizations

Optimizations at the design-level, as the name suggests, refer to changes in the overall design of the system. These optimizations may involve changes in the way in which software components communicate, changes in APIs, and changes in how components are structured and subdivided. For example, use of Ada generics may lead to longer execution times as some compilers fully inline the code, therefore missing significant benefits of instruction cache. Changing the architecture of the program to use less generic components and re-usable APIs has other consequences related to ability of the compiler to do constraint checking.

Analysis at this level is usually difficult and expensive as it may require changes to the overall system design, which can have significant impact on implementation and testing. However, very significant improvements in execution time can be achieved by changes at the design-level.

Sub-program-level optimizations

Optimizations at the sub-program level focus on changes within a single sub-program (or a set of tightly coupled sub-programs) without changing the specification of those components. Examples of these optimizations are changing the complexity of an algorithm, for example from an $O(n^2)$ to an $O(n \text{ Log } n)$ sort routine, changing an iterative process to one using lookup tables, loop unrolling, and avoiding making extra copies of data, therefore reducing memory footprint and the potential for cache misses.

Low-level optimizations

Low-level optimizations focus on the generated machine code. These optimizations aim to use the most efficient available machine instructions for performing particular tasks. For example, in some DSP processors, specific machine instructions exist to find the first-set bit or count the number of set bits in a word. These instructions are significantly faster than typical software implementations of the same functionality. Another example of machine dependencies is using non-native word sizes. This may result in significantly larger and slower code. A programmer who is not aware of this fact may miss an important opportunity for optimization.

Another important aspect is the nature of the generated code, especially relevant for Ada is the fact that a few lines of code can result in a very long execution time. For example:

```
type T is new integer;
type U is array ( 0 .. 10000) of Big_Record;
  …
  a,b : T;
  c,d : U;
  …
  a:=b; -- single copy of integer
  c:=d; -- can take a very long time to run!
```

The two last statements, although very similar at the source code level result in very different object code.

A particular aspect of importance at this level is the identification of the impact that compiler optimizations have on the code. For example, on modern processors with large caches and small memories, using compiler optimization for size can, counter-intuitively, result in better execution time performance than using compiler optimization for speed. This occurs when the bottleneck is actually fetching code from main memory, rather than the actual processing of those instructions.

4.2 RapiTime optimization process

RapiTime provides information on the percentage contribution of each sub-program to the overall execution time. This information is used to identify candidate sub-programs for optimization. The best candidates for optimization are then inspected. This involves studying both the Ada source code and in some cases the object code generated by the compiler.

Next, RapiTime is used to answer what-if questions about the effects of potential reductions in the execution time of these sub-programs. This shows that optimizing some

candidate sub-programs would result in a commensurate reduction in the overall worst-case execution time; whilst for other sub-programs, optimization would bring little benefit as the worst-case path shifted to other code.

Of particular importance is the fact that even though a sub-program can be a worst-case hot-spot, its optimization may not necessarily lead to a significant reduction in the overall worst-case execution time if by optimizing that code, the worst case path switches to another path. For example

```
If some_condition then
  A; -- in worst-case path. Takes 10 ms
else
  B; -- not in worst-case path. Takes 9ms
end if;
```

In this example, reducing A by more than 1 ms, switches the worst-case path to the branch B, therefore both A and B need to be optimized together to reduce the worst-case execution time.

Quantification of the improvement

An important aspect of the optimization process is a final review examining the impact and consequences of the optimization process. This review quantifies the reduction in execution time, and also assesses the impact of code changes on portability, maintainability, code size, etc. Some optimizations may be rejected at this stage if they do not bring sufficient benefits to warrant for example non-portable or difficult to maintain code.

5 Main results of the study

This section describes the main results of the study. The target for phase 1 of the study (reported here) was to deliver a saving in the overall schedule, corresponding to 100 execution time units (ETUs)[2]. Achieving this reduction would put the project well on track to achieve the overall reduction required to accommodate additional functionality.

In all, 5 out of 25 software partitions were analysed. These 5 partitions are referred to below as Partitions A to E. The software for these partitions amounted to over 100,000 lines of Ada code. Three of the partitions, A, B and C were comprehensively analysed, with improvements and targets for optimization selected on the basis of the information provided by RapiTime. Optimizations were prototyped for these partitions and the RapiTime performance analysis re-run to quantify the improvements obtained. For the final two partitions, D and E, several optimizations were identified, however prototyping and further analysis awaits the next phase of the study.

The analysis process sought to achieve savings in the overall execution time schedule, in the following categories:

1. **Budget reductions:** reductions in execution time budgets and hence schedule slots made possible by more accurate analysis of partition worst-case execution times.
2. **Optimizations** at design level, sub-program level and low-level.

Major savings in each of these categories are discussed in the following sections.

5.1 Budget reductions

During initial investigation of Partition A, it was found that the schedule slot (execution time budget) was significantly greater than actually required in the context of its use in the Operational Flight Program. The schedule slot had previously been increased to accommodate use of the partition in different context where it had a much longer execution time. Accurate context dependent analysis of the execution time allowed the budget to be safely reduced by 58 ETUs.

5.2 Design level optimization

Detailed analysis of Partition A revealed that over 80% of its execution time was spent copying data to a large intermediate buffer. Further investigation showed that in the context of how the software was used in the Operational Flight Program, only one response at a time was possible from any given client and thus the intermediate buffer copy was unnecessary. Removing this copy reduced the execution time of Partition A by over 62%, an overall saving of 17 ETUs.

This optimization opportunity is representative of the value of prioritising optimization opportunities. Determining that the usage of this component did not need an intermediate buffer was not obvious and required detailed discussion with various engineers responsible for the overall design of the system. This investigation would have not been performed if there had not been strong evidence of potentially large savings in execution time.

5.3 Sub-program optimization

Analysis of Partition C revealed that over 25% of the execution time of the partition was spent copying data in a loop that iterated over 2000 times. Close inspection of the code that performed this copy showed numerous redundant constraint checks. This code was replaced by a call to memcpy enabling the compiler to use more efficient code for the copy, without the large number of constraint checks. This reduced the execution time of the sub-program by over 80%, resulting in an overall reduction in the execution time of the partition of 23%, corresponding to a saving of 48 ETUs.

This optimization shows the trade-off between maintainability and code readability versus execution time. Widespread use of memcpy routines for copying data is not recommended as it makes the program less readable and less maintainable; however, in this context the change was more than justified by the significant gain in performance.

[2] ETUs are an arbitrary execution time unit used in this paper. The actual values are ‘commercial in confidence’ and are therefore not reported here.

5.4 Low-level optimizations

In Partition B, RapiTime showed that a small bit-unpacking sub-program was called over 700 times on the worst-case path. Further investigation showed that the compiler generated code was not particularly efficient. Writing the Ada code in a different way allowed the compiler to produce more efficient code, reducing the execution time of the sub-program by 57%, corresponding to an overall reduction in the execution time of Partition B of 7%, and a saving of 11 ETUs.

In general, it is not practical to do object code investigation on even a medium size program. However, using RapiTime, it is possible to identify code fragments that contribute significantly to the overall worst-case execution time. Blocks of code that are called very frequently on the worst-case path (over 700 times in this case) are often a good target for low-level optimization, as proved to be the case here.

5.4 Summary of the results

Overall, the following savings in the schedule were achieved:

- **76 ETUs** due to prototyped optimizations, including:
  - 17 ETUs from design level changes.
  - 48 ETUs from sub-program modifications.
  - 11 ETUs from low-level optimizations.
- **58 ETUs** due to identifying a reduced execution time budget for Partition A.

Total reduction in execution time **134 ETUs,** exceeding the targeted reduction of 100 ETUs.

Using RapiTime to identify candidates for optimization, it was possible to achieve reductions, amounting to **23.6%** of the execution time of the analysed partitions, whilst needing to manually examine just **1.2%** of the total lines of source code in these partitions. These 1250 lines of code were initially responsible for 29% of the overall execution time of the partitions. Design-level, sub-program-level and low-level optimizations reduced this contribution by a factor of almost 5, creating headroom for new functionality to be added without the need for expensive hardware upgrades.

	Execution Time		Improvement
Partition	**Before**	**After**	**%**
Partition A	28.2	10.6	62.4%
Partition B	140	129	7.9%
Partition C (1)	95.5	72.9	23.7%
Partition C (2)	58.1	33.2	42.9%
Total	321.8	245.7	**23.6%**

Table 1 Reduction in worst-case execution times achieved using RapiTime

The reductions in partition execution times achieved are summarised in Table 1 and illustrated in Figure 2. Partition C appears twice as it is executed twice within the major cycle of the schedule. The contexts of these two executions are however different and consequently two different context dependent execution times were derived for Partition C.

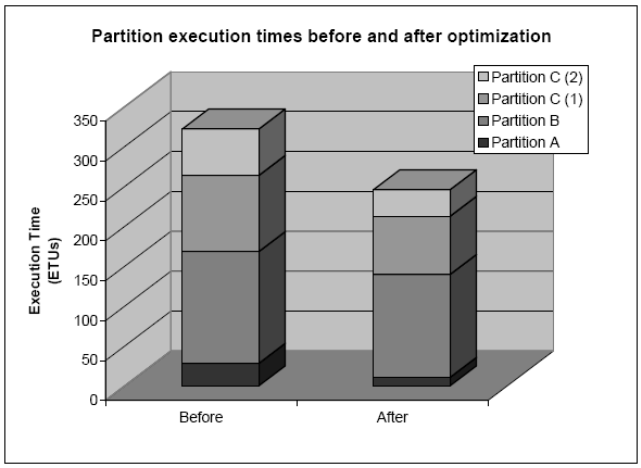

Figure 2 Reduction in worst-case execution times achieved using RapiTime

6 Summary and conclusions

During the study described in this paper, the process of using RapiTime for the Hawk AJT project was refined. A number of partitions within the Operational Flight Program of the Hawk Mission Computer were successfully analysed and significant reductions in execution time made. Overall, the improvements made put the project on track to provide the headroom necessary to incorporate additional functionality without recourse to an expensive hardware upgrade.

As part of the study, RapiTime identified that only 1.2% of the code contributed more than 29% of the overall worst-case execution time. These blocks of code were obvious targets for optimization. A detailed study of some 1250 lines of code identified specific targets for optimization and hence opportunities for execution time reduction. These optimizations were classified as: low-level, sub-program-level and design-level. The best candidates were prototyped and implemented and the new system analysed to verify the effectiveness of the changes. The optimized partitions had an execution time that was 23% smaller than before.

References

[1] BAE Systems. Hawk Jet Trainer. http://en.wikipedia.org/wiki/BAE_Hawk

[2] G. Bernat, A. Burns and M. Newby (2005), *Probabilistic Timing Analysis. An approach using Copulas*, Journal of Embedded Computing, Vol 1 no 2, pp 179-194

[3] Rapita Systems Ltd. *RapiTime White Paper*. (2005.) http://www.rapitasystems.com