TASKING

Tasking Embedded Profiler
Product Overview

Novel Smart Profiling Technology - Remove Bottlenecks in a Fraction of the Time Required for Traditional Measurement Profiling

TASKING®
An Altium Brand

TASKING EMBEDDED PROFILER — HIGH LEVEL OVERVIEW

Traditionally, profiling is associated with measuring execution times of functions. Knowing that a function takes a specific amount of clock cycles or milliseconds to execute is in itself not very helpful to identify if the function can be improved at all and which code or setting needs to be modified in what specific way to actually make the function run much faster on your specific target device.

Unlike existing profilers that measure only function run-times, the **TASKING Embedded Profiler** has *built-in expert-level knowledge about the AURIX inner-workings*, so that it can:

- Identify functions and code lines that unnecessarily waste large amounts of core time because they misuse hardware resources within the AURIX,
- Explain the root cause of these performance bottlenecks,
- Provide concrete mitigation suggestions to quickly resolve the bottleneck with minimal effort.

After a short, non-intrusive analysis of your code's runtime behavior using a simple USB-cable or inexpensive mini-wiggler connected to your AURIX board, the **TASKING Embedded Profiler** tells you exactly which source lines and configuration settings cause the biggest slow down, what the root cause of that slowdown is and what you need to do to fix the specific problem. After having fixed the problem, function runtimes measured by the tool before and after the fix can be compared to evaluate what speedup was achieved by applying the fix.
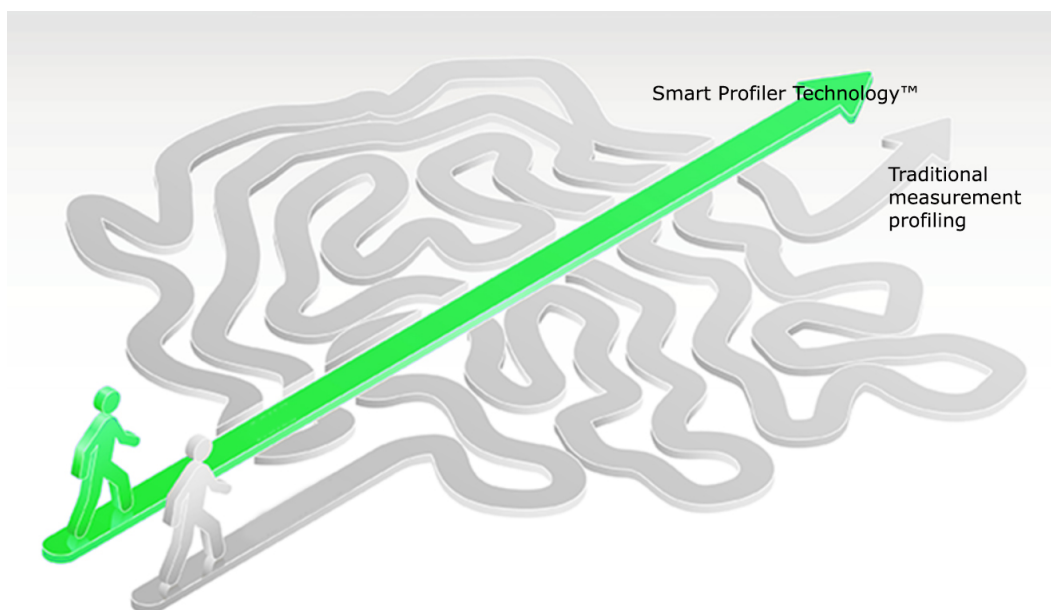
When applying this **smart profiling technology** to code running on a new microcontroller, often significant speed-ups can be achieved within a few hours of work because the small changes needed to improve the code (without altering what is computed) can be found immediately and the improved code utilizes the microcontroller specific hardware resources much more optimally.

Smart Profiling Technology

- Built-in, expert-level knowledge about the AURIX inner-workings

    - Memory System Delays

    - Cache Behavior

    - Branch Prediction

    - Clock Frequencies

    - And more

- Short, non-intrusive measurement of an application's misuse of specific hardware resources.

- Easy to understand, graphical representation of the root cause (source lines and reasons) that generate the biggest slowdowns.

- Concrete mitigation suggestions for different root causes to quickly fix the problem

- Compare function and application runtimes before and after applying a mitigation to document the real world performance improvement of the mitigation.

- Quick turnaround time, often improves application performance significantly within a few hours.

- Performance bottlenecks are analyzed for the entire application or for a user-selected subset of functions and the results are presented and in order of descending severity — spend your time where it matters most.

- Easy to use for non-experts: Start the TASKING Embedded Profiler, attach to your device, measure, follow mitigations to improve performance. **No expensive probes, special knowledge about the hardware or preparation of the application is required**.

- Stand alone tool, can be used with any application (irrespective of compiler tool set and programming language*) running on the supported target cores.

In the remainder of this product overview, we will discuss a typical workflow of the TASKING Embedded Profiler using concrete examples and give more details on how it functions.

*Source level information is limited to C-code compiled with DWARF 3 compatible tool chains. Otherwise, assembly level information is provided.

TASKING EMBEDDED PROFILER — DETAILED INTRODUCTION

Stalls - The Root Cause of Hardware Performance Issues

A stall occurs when the microcontroller is blocked from performing useful work (i.e. execute your program). Instead, the microcontroller consumes energy and time to resolve the stall.

For example, most multi-core microcontrollers do not have a constant memory access speed. Specific cores can access specific memories faster than others. If data that is placed in a slow-to-access memory is frequently accessed by a specific core, then that core will spend most of its time stalled, waiting for data rather than doing useful computations.

Depending on the specific microcontroller, there are many different kinds of stalls and these stalls are not visible to the user — the program delivers no error or warning, the same result is generated, irrespective of the number of stalls.

Since there are many different kinds of stalls that are invisible to the application, it is not uncommon that, without the knowledge of the developers, even mature applications may spend more than 50% of their time stalled.

Usually, minor changes to the compiler settings, program code or memory layout (e.g. by moving a variable from a slow-to-access to a faster-to-access memory) can eliminate a large portion of the stalls so that the program runs several-hundred-percent faster while computing the same results as before. On the other hand, changing compiler settings, program code or memory layout in the wrong way can introduce a large number of new stalls.

Traditional Profiling

Using a traditional profiler that merely exposes function runtimes, you are left to guess — maybe employing your gut feeling and many years of experience with the specific hardware — what changes might reduce the number of stalls. You can compare runtimes before and after a change that feels promising and by employing a lot of time through trial and error you might find a few improvements.

Smart Profiling

In addition to the function runtimes, the TASKING Embedded Profiler shows you a graphical summary about which functions generate most stalls so that the problems that cause the most significant slowdowns can be easily identified and attacked first, without any guess work. You can drill down into each function to find the source lines and assembly instructions that caused the problem.

It is not necessary for the user to have a deep understanding of the hardware or the different stall sources. The smart profiler compares your measurements results to knowledge about the stall behavior of applications that are known to use the hardware resources well. Using this reference, the tool evaluates which functions and source lines have abnormally high stall rates and offers mitigation suggestions to fix these specific problem instances. The tool infers the mitigation suggestions from the measured stalls in combination with in-depth knowledge about the target specific stall sources. This knowledge is encoded into the smart profiler itself.

The data collection of the smart profiler works by gathering statistics about program instructions (assembler instructions that are translated into source lines using debug information) which generate stalls within the target microcontroller using non-intrusive hardware tracing.

An Example Workflow in the TASKING Embedded Profiler

In the following, we will examine how the smart profiler works on a specific example. Generally, the workflow follows this pattern:

1. Compile and flash target application

2. Analyze problems and find new improvements using the smart profiler

3. Revise application

4. Recompile, reflash and reprofile to compare results and verify improvement

5. Repeat at Step 1 until you are satisfied with your application

1. Compile and Flash Target Application

We will investigate the following, quite simple application:

```
#define ARRAY_SIZE   (16 * 1024)
volatile int x[ARRAY_SIZE];

int main(void)
{
      clock_t clockstart = clock();
for (int i = 0; i < ARRAY_SIZE; ++i){ x[i] = 1; }
      int duration = (int) (clock() - clockstart);
      printf("duration %i ticks\n", duration);
}
```
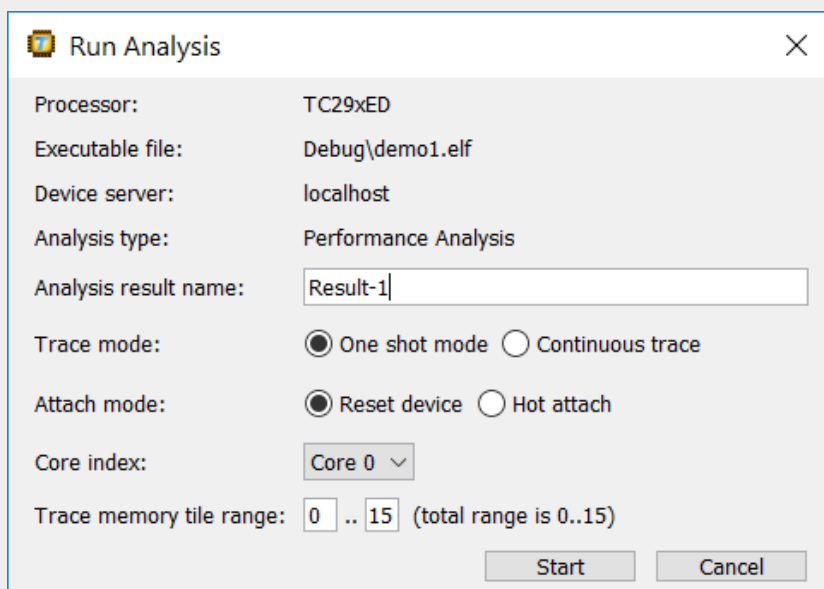
First, the application needs to be compiled with the highest optimization settings you are allowed to use — after all you want to measure performance — and flashed to your target device. Running the application will give you similar information to using a traditional profiler, that is, the execution time of the `for` loop computed using `clock()` is e.g. 73754 ticks on a specific device. It seems, there is not much that could go wrong with this application.

2. Smart Profiling

The TASKING Embedded profiler is a stand alone application. Thus, a project that will store your application data (links to folders containing the target application and sources) as well as smart profiling results needs to be created. This will just take a minute and only needs to be done once before you can start profiling an application.

After the project has been created, you can select what you want to optimize for using the corresponding analysis. The smart profiler support several analysis kinds, depending on the kind of problem you are investigating. Usually, you will start with a "performance analysis" which will give you a high level overview over the biggest issues. Simply click the "plus" button, follow the wizard to create a new "performance analysis" and run the analysis once by selecting it in the project tree on the left and pressing the "play" button (see Figure1). The different analysis configuration options available after pressing "play" are discussed in the box on the side, for now we will just start the analysis as is by pressing "start".

Analysis Configuration

Trace Mode: Trace until trace memory is full or upload from trace memory while tracing (may cause short stops in the application that is being traced while trace memory is read).

Attach Mode: Reset device before measurement or keep current system state.

Core Index: Which core to measure on (currently only the code running on one core can be analyzed at a time).

Tile Range: Which part of the trace memory is the profiler allowed to use for tracing (allows you to allocate parts of the trace memory for other applications).

Figure 1: A new performance analysis can be easily created using the Wizard.

After a few seconds where the analysis data is collected from the hardware and the results are computed, the analysis summary is displayed as shown on the right hand side of Figure 2.
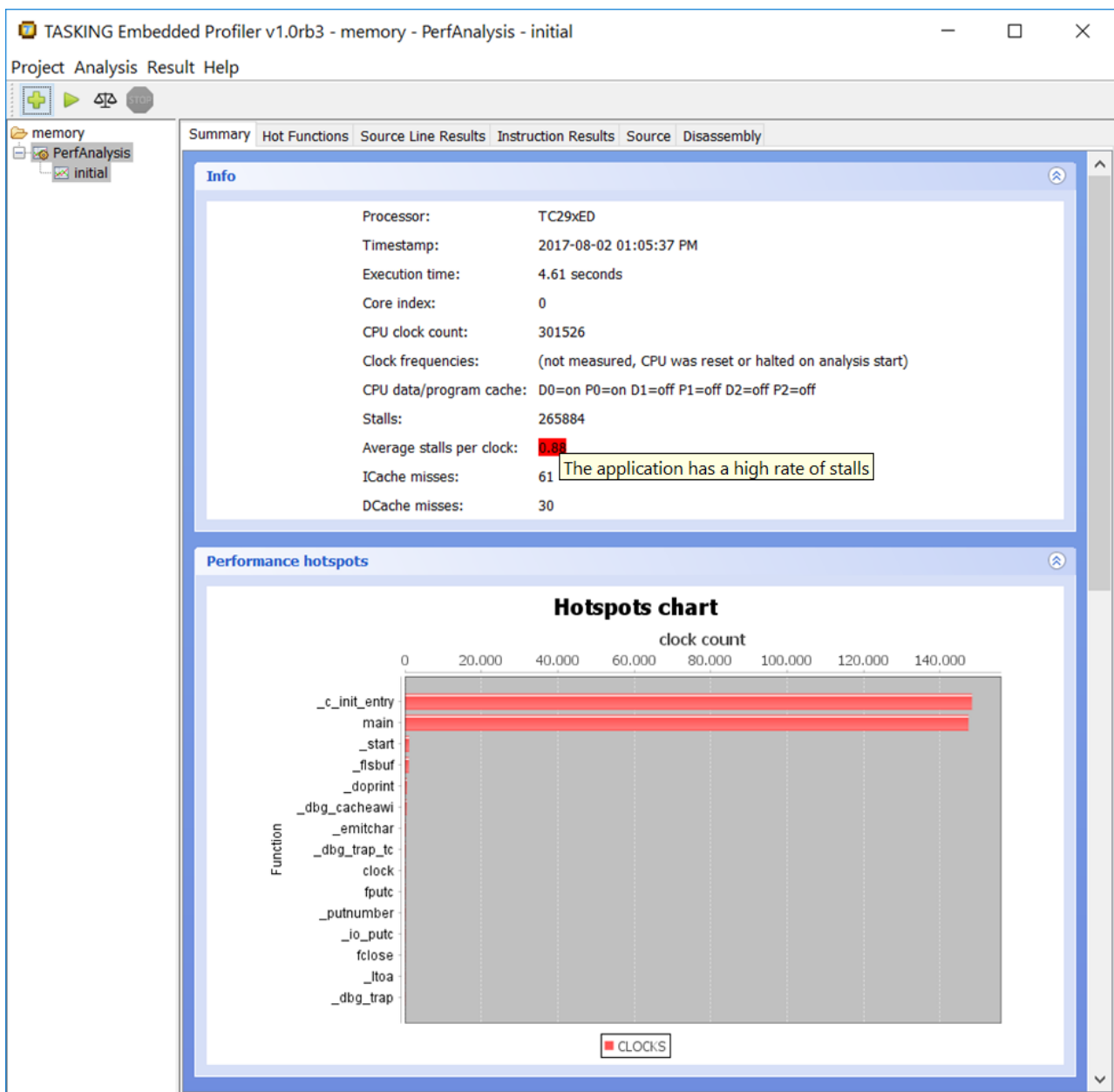

Figure 2: The project tree on the left hand side allows you to navigate previously configured analyses and analysis results. The right hand side shows the summary of the performance analysis. In this example, the main function generates an unusually high number of stalls. 88% of the CPU time is wasted. Clicking on "main" in the graph will drill down into the function level results, as shown in the Hotspots chart.

The summary shows that the applications spends 88% of the execution time stalled (red underlaid key performance indicator), without these stalls it would finish about 8 times faster in about 36k clocks instead of the current 301k clocks.

The "Hotspots Chart" shows the functions where most time is spent during program execution. These should be improved first as this will yield the largest performance gains. Double clicking on the red bar next to "main" in this chart opens the combined source and analysis results view for function main shown in Figure 3.
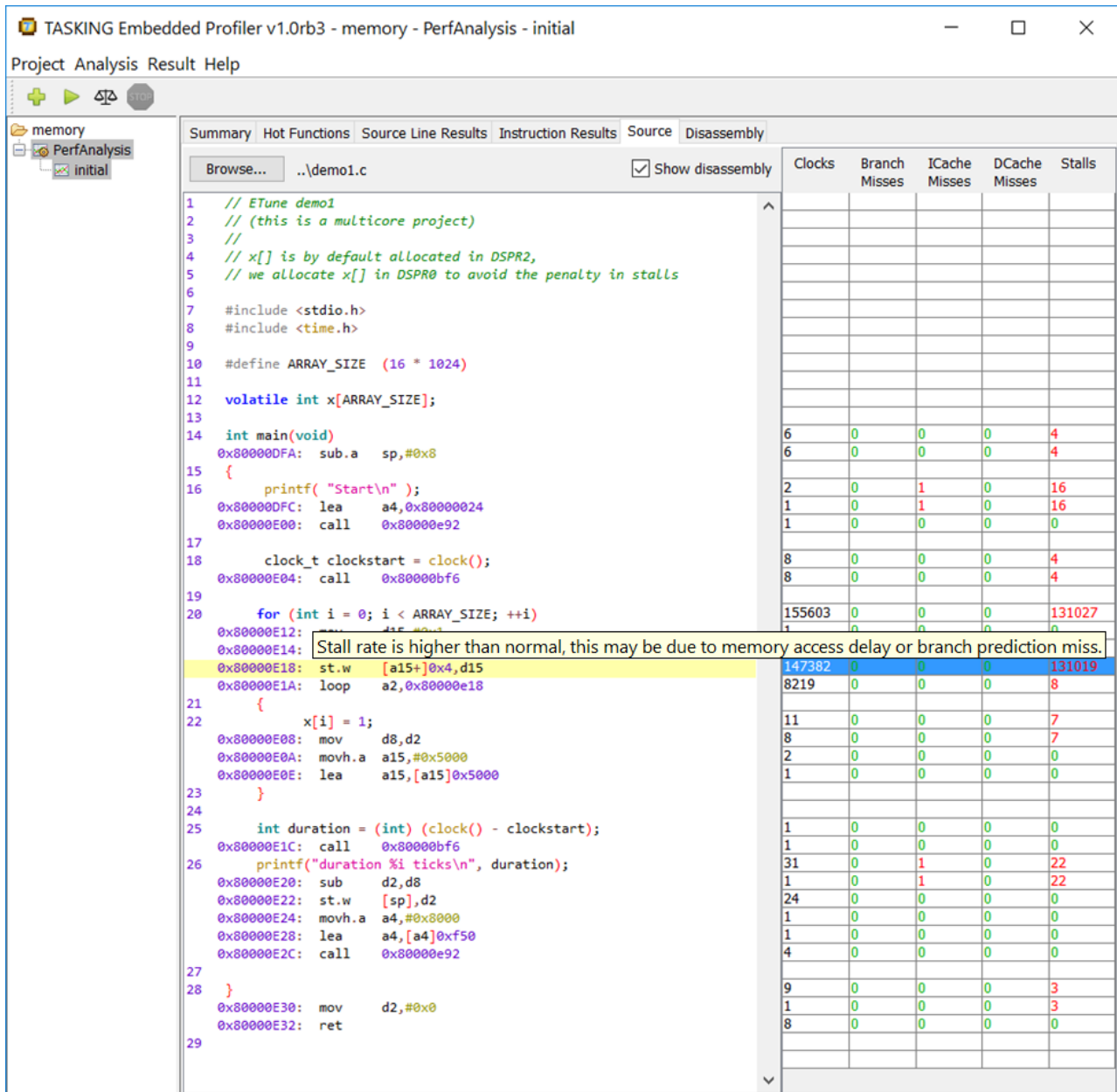

Figure 3: Function level analysis results for "main". The stall column from the results table on the right indicates that the write to the array in x in line 22 causes excessive stalls (almost one stall per clock executed on the instruction).

Next to the source view, the table on the right indicates what stalls were measured for the corresponding source line or assembly instruction. Using this table, it is easy to see that the for loop in line 20 of Figure 3 causes a high number of generic stalls (almost one stall for every clock cycle). The stalls are generated by the assembly instruction (highlighted in yellow) which writes the constant 1 into the memory that holds the array x and the root cause lies in the memory subsystem.

Using the "performance analysis" we have instantly identified the most significant slowdown and that the root cause lies in the memory system. In order to pinpoint the root cause in the memory system and to find a mitigation for the problem, we will execute a "memory analysis". After using the "plus" button to create a "memory analysis" and running it using the default parameters, the tool exposes the exact stall cause as shown in Figure 4.
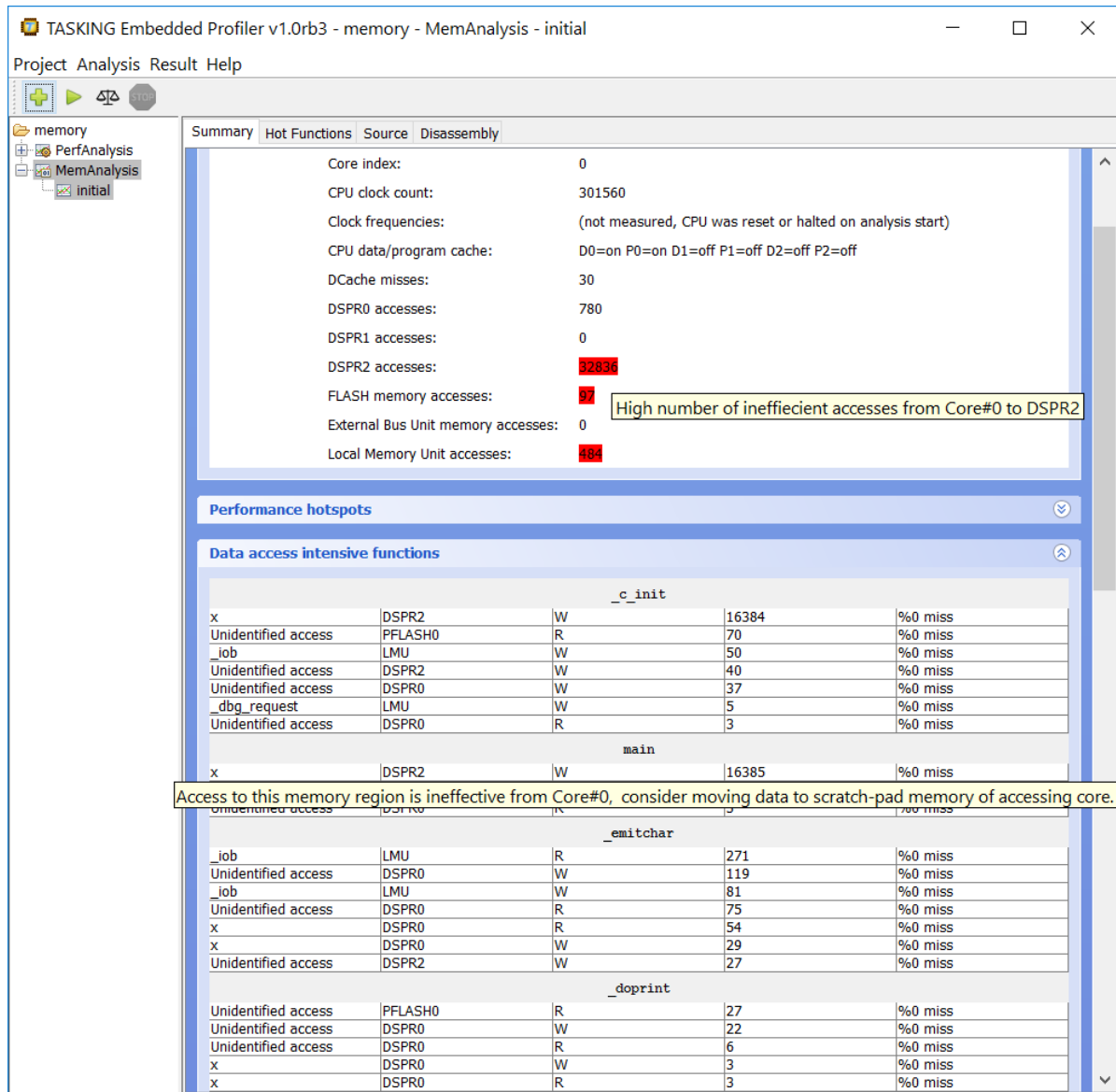

Figure 4: The example code is running in core 0. The "DSPR2 accesses" KPI underlaid in red shows the main problem. When hovering the mouse over the KPI, the tool indicates that core 0 makes too many accesses to DSPR2 which is the core local memory of core 2 (rather than using its own and much faster to access DSPR0). Our main function is listed in the "Data access intensive functions" table and hovering the mouse over the row that contains most inefficient accesses in main reveals that the cause is the accesses to the global array x. As mitigation for this problem, the tooltip suggests to move x into the scratch-pad memory of core 0, i.e. DSPR0.

3. Revise Application

Now that the root cause of the biggest performance problem has been identified using two quick measurements and the analysis results from the smart profiler as shown in Figure 2 and Figure 3, the application can be easily improved. When using the TASKING toolset, the global variable x can be moved into DSPR0, as suggested by the profiler in Figure 4, by changing the lsl linker script or by adapting the declaration of the global array:

```
volatile __private0 int x[ARRAY_SIZE];
```

4. Recompile, Reflash and Reprofile to Verify Improvement

After recompiling and reflashing the application including the minor change from Example 2, another performance analysis can be executed in order to verify that we fixed the problem and to find new optimization opportunities.
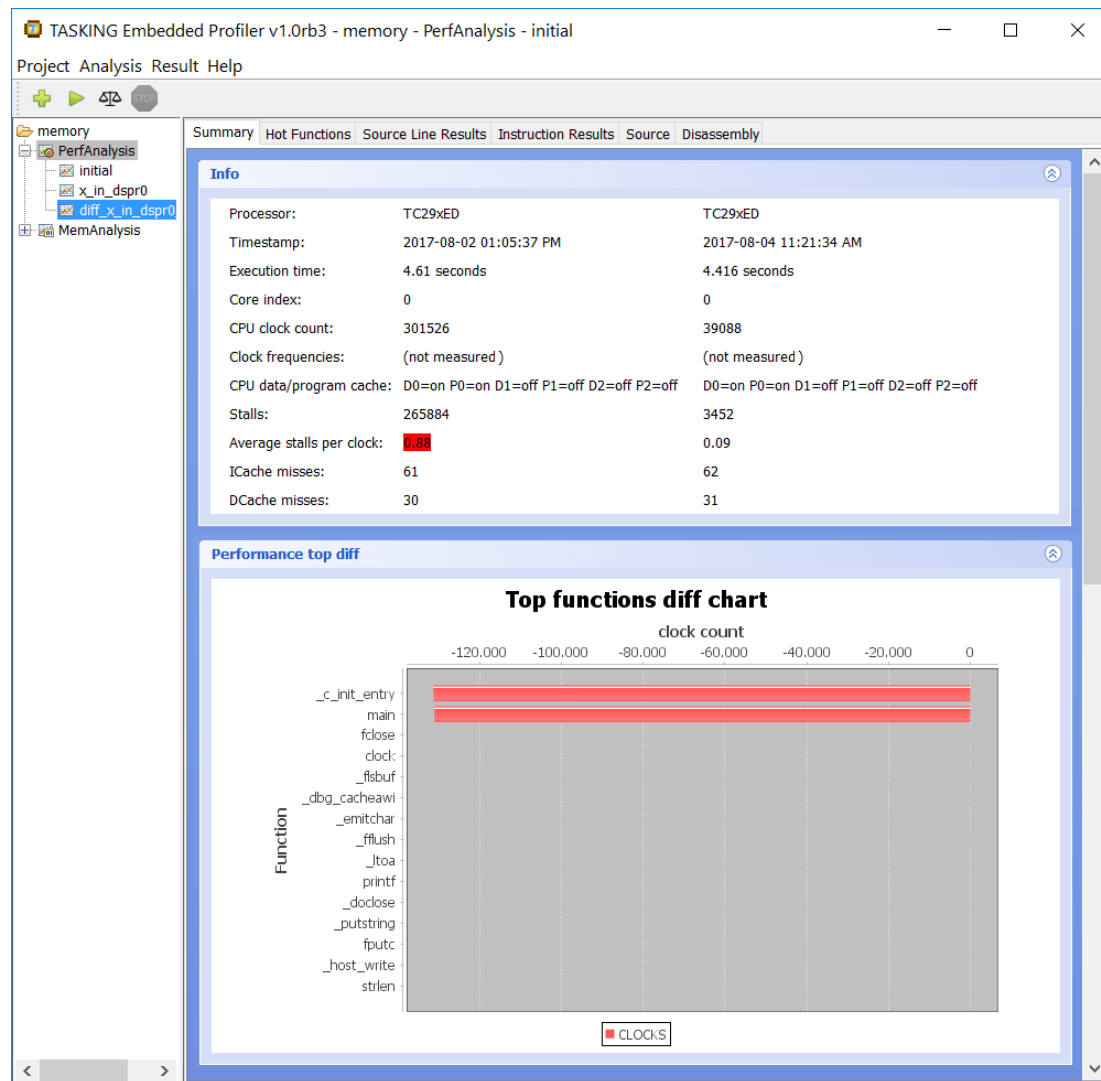

Figure 5: The tool computes the difference between analysis results so that results from before and after a modification can be easily compared. Here, the diff between the "initial" result and the result "x_in_dspr0" obtained after moving x into DSPR0 is shown.

The diff shown in Figure 5 clearly shows that the improvement was highly successful. The improved application computes the same result as the original one in about 1/8th of the clock cycles, i.e. it is about 8 times faster. Average stalls per clock are reduced from almost 90% to around 10%. No relevant number of new stalls was introduced by the change.

5. More Improvements

After the effectiveness of the first improvement has been verified, the next biggest performance issue of the application can be attacked by investigating the results of the second performance analysis and the whole procedure can be repeated until all relevant performance issues have been addressed. One iteration typically only takes 10 - 20 minutes and the tool exposes the problems in order of decreasing severity so that after a few iterations the most serious performance gains are realized.

In addition to the memory problem investigated in this example, the TASKING Embedded Profiler exposes all other stall kinds that are visible through the hardware so that you can rest assured that the hardware is used to its maximum after having eliminated the problems visible in the smart profiler.

Performance Regressions

Once you have addressed the major slowdowns caused by stalls in your applications, drivers and OS you will want to monitor changes made to any of these components. Through the smart profiler's command line interface, you can automatically re-execute a previously configured analysis and export the result (or the diff) as comma separated value file. By comparing current to previous performance statistics as part of your build or development process, changes to any software component that adversely affects the performance through unnecessary hardware stalls can be easily identified and resolved.

Return On Investment

In the following table we investigate the cost savings (in terms of time spent to achieve comparable results, in terms of the necessary hardware and in terms of software licenses) when comparing traditional profiling to TASKING's smart profiling solution.

Traditional Profiling	Smart Profiling	Relative Cost
For a typical powertrain project an expert level engineer spends ca. 2 man months optimizing the linker layout alone.	Non-expert can identify and resolve the biggest issues in a few hours.	> 20:1
Existing tracing tools require expensive hardware probes to get "all data" from the hardware. The data needs extensive post processing and expert knowledge to extract relevant information.	TASKING's smart profiler connects to the hardware using a mini-wiggler (ca. 100 EUR) or a simple USB-cable like the one used to charge your phone. Only relevant data is extracted, automatically post-processed and interpreted.	> 10:1
Trial and error based search for optimizations does not guarantee any positive results, worst case weeks are spent with no overall improvement.	Smart profiling guarantees that the most relevant problems are attacked first and exposes the problem's root cause so that the time is spent fixing the relevant problems rather than guessing what the problem is.	> 2:1
Cost of debugger software for tracing	Smart profiler software	> 2:1

Table 1: ROI

USAGE SCENARIOS WITH THE TASKING EMBEDDED PROFILER

Typical use cases for smart profiling tools include:

Quality Assurance for Software from Suppliers

Automated monitoring and verification of the performance characteristics of the software delivered to you from your suppliers. Define performance thresholds like maximum stall rate, maximum jitter or maximum execution time for functions which are developed by a supplier and automatically find violations of these criteria for code delivered to you to ensure best software quality.

By suppliers themselves, the tool is used make sure that they deliver the best possible code to their customers.

Realistic Timing during Function Development (Applications)

For hard real-time applications, reliable and predictable timings of your functions is essential. By applying smart profiling early on during the development process of your software functions, you can ensure that no nasty surprises are hidden in the code and that the timings found during development are close to what you will see when deploying the code.

Optimization (OS, Drivers, Libraries, Applications)

When developing new functionality, integrating existing functionality or porting applications to new target hardware, you are bound to introduce new hardware stalls unwittingly. Instead of hoping for the best and handing your project over to the firefighters when it becomes apparent that core utilization or timing behavior become untenable, using smart profiling early on allows you to be in control of your software performance. Deliver better code more quickly, stay in control of your code's performance.

Regressions after Code Changes (All Code)

Even minor and innocent looking code changes have the potential to introduce a large amount of stalls and other performance degrading events. Make sure that changes to your code's performance characteristics are made consciously and mistakes are identified before any harm is done by running automated stall regressions before any code change is checked in.

Fire-Fighting Timing and Performance Issues

At the end of the project you find that that some of your end to end timings violate the project requirements or some cores are overutilized. Changing the OS schedule to attempt a fix is highly risky as all sorts of knock on effects may skew the timing in other event chains. If only you could make that one task/runnable execute 20% faster, then all would be well. If the specific task/runnable has not yet been investigated using smart profiling then there is a good chance that you can reduce the runtime of the target task/runnable by 50% or more using smart profiling in an hour. The TASKING Embedded Profiler can target the whole application or a set of specific, user selected functions.

SUMMARY

The TASKING Embedded Profiler makes **smart profiling** available for deeply embedded devices like the AURIX providing you with control of the hardware rather than treating it as a black box that can only be demystified by large investments in experts, time, hardware and software.

Get started with the TASKING Embedded Profiler today by registering for a free evaluation.

TASKING