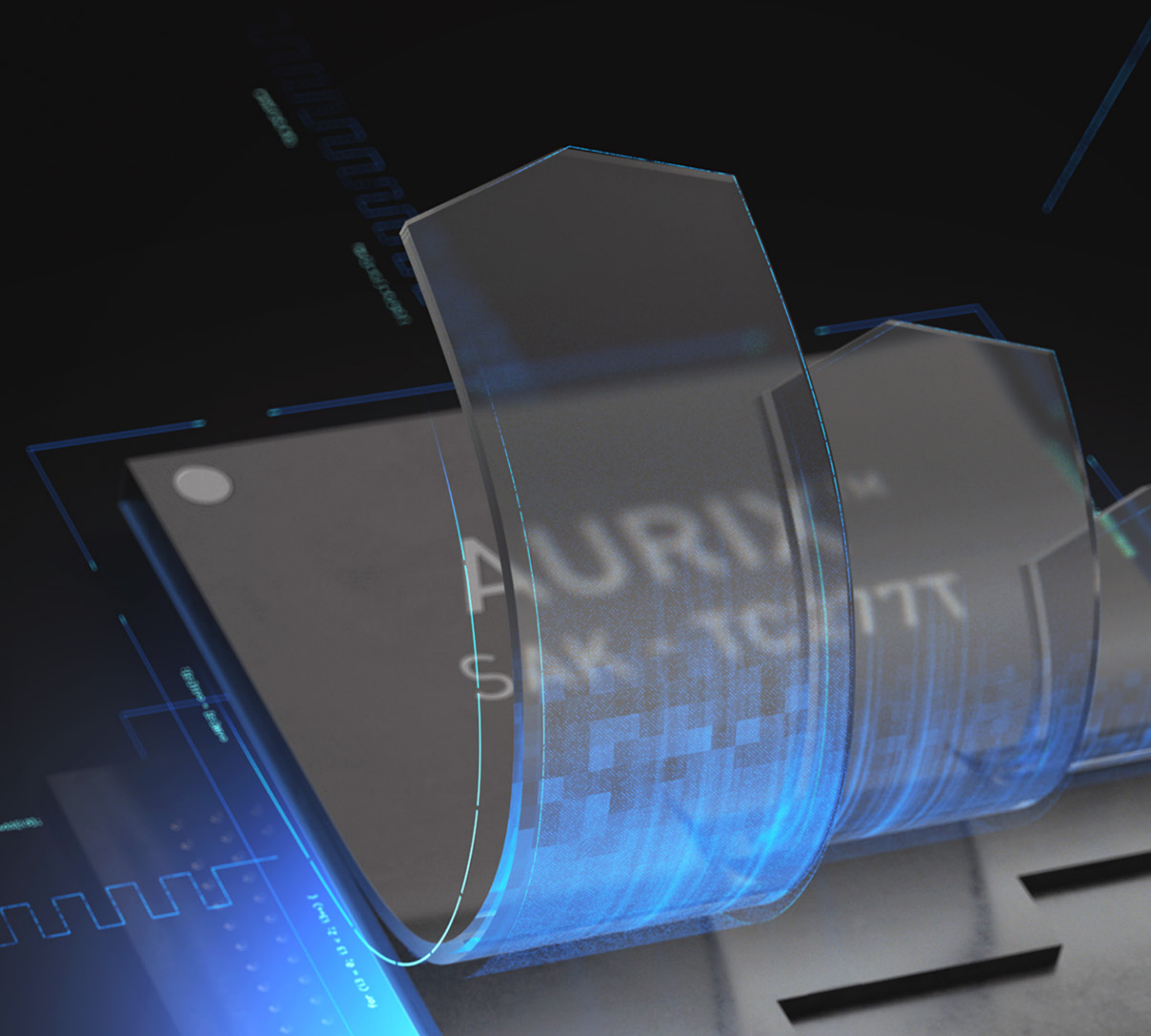
TASKING

Tasking Embedded Profiler Product Overview



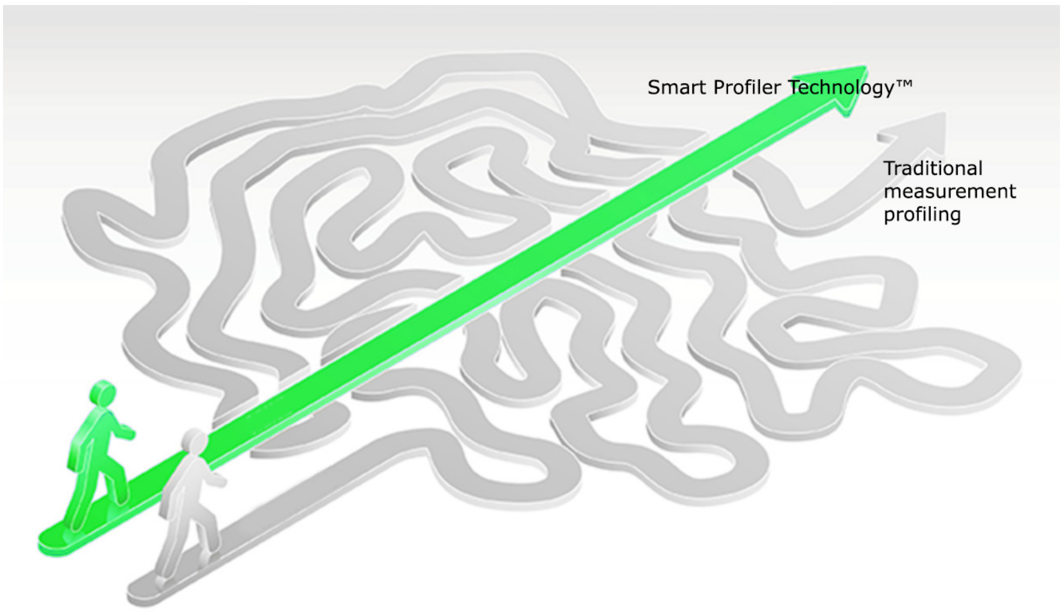
Innovative Smart Profiling Technology™ – Remove Bottlenecks in a Fraction of the Time Required for Traditional Measurement Profiling

TASKING® EMBEDDED PROFILER — OVERVIEW

Traditionally, profiling is associated with measuring execution times of functions. Knowing that a function takes a specific number of clock cycles or milliseconds to execute doesn't help determine whether the function itself can be improved. Nor does that timing information indicate which code or setting needs modification to significantly improve the function's runtime on a specific target device.

Unlike existing profilers that measure only function runtimes, the TASKING® Embedded Profiler has built-in expert-level knowledge about the AURIX™ microcontroller 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 microcontroller.
- Explain the root cause of these performance bottlenecks.
- Provide concrete mitigation suggestions to quickly resolve the bottleneck with minimal effort.



After a brief, non-intrusive analysis of your code's runtime behavior using a simple USB cable or inexpensive mini-wiggler connected to your AURIX board, the profiler tells you exactly which assembly, source lines and configuration settings cause the biggest slowdown, the root cause of that slowdown, and what you should do to fix the specific problem. After fixing the problem, you can compare function runtimes measured by the tool before and after the fix to evaluate what speedup was achieved.

The TASKING Embedded Profiler is a 64-bit application to ensure fast operation and the handling of large Embedded Profiler trace files.

When applying the profiler's Smart Profiling Technology™ to code running on a new microcontroller, significant speedups can often be achieved within a few hours of work because the small changes needed to improve the code (without altering what is computed) are found immediately, and the improved code better utilizes the microcontroller-specific hardware resources.

Smart Profiling Technology™

The TASKING Embedded Profiler's Smart Profiling Technology offers many benefits:

- Built-in, expert-level knowledge about the inner-workings of the AURIX family of microcontrollers, such as the:
  - Memory system and cache behavior
  - Branch prediction and instruction pipeline behavior
  - Clock frequency settings
  - On-chip debug systemяя
  - And more…
- Brief, non-intrusive measurement of an application's misuse of specific hardware resources.
- User-friendly 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.
- Quick turnaround time, often improving 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 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, and implement mitigations to improve performance. **No expensive probes, special knowledge about the hardware, or preparation of the application are required.**
- Standalone tool can be used with any application (regardless of compiler tool set and programming language[1]) running on the supported target cores.

The following sections discuss a typical workflow of the TASKING Embedded Profiler. They use concrete examples and give details on how the profiler functions.

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

THE ROOT CAUSE OF HARDWARE PERFORMANCE ISSUES: STALLS

A stall occurs when the microcontroller is blocked from performing useful work (i.e., executing 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, many types of stalls can occur that are not visible to the developer. The program delivers no error or warning; the same result is generated, regardless of the number of stalls.
Because many stall types are invisible to the application, it is not uncommon that—without the knowledge of the developers—even mature applications may spend between 30 and 70 percent of their operation time stalled.

Minor changes to the compiler settings, program code, or memory layout (e.g., moving a variable from a slow-to-access memory to a faster-to-access memory) can usually eliminate a large portion of the stalls so that the program runs significantly faster while computing the same results as before. However, incorrectly changing compiler settings, program code, or memory layout can introduce many 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 seems promising, and by spending a lot of trial-and-error time, you might find a few improvements.

Smart Profiling with the TASKING Embedded Profiler

In addition to the function runtimes, the TASKING Embedded Profiler displays a graphical summary about which functions generate the most stalls so that the problems causing the most significant slowdowns can be easily identified and attacked without guesswork. You can then drill down into each function to find the problematic source lines and assembly instructions.

It is not necessary to have a deep understanding of the hardware or different stall sources. The profiler compares your measurement 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 it 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 profiler itself.

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

Memory Traces Made Easy

This type of analysis traces function calls, function returns, and data accesses. You can run memory traces on the whole application or selected specific functions. Memory traces are an important aspect of optimizing applications using the Infineon® AURIX™ family of microcontrollers. These microcontrollers are very complex, with several CPU cores on the same die. Each CPU core has local memory but can also access the local memory of the other cores and the general memory on the board.

These cross-memory accesses can affect application performance because the shared data bus is used for each read or write data request from one CPU core to the memory, which is local to another CPU core or to the general memory. Accessing memory that is not local to the CPU core takes significantly longer than local memory access, and it blocks all other CPU cores that want to read or write memory from other local memory or the general memory. (This blocking is called a “stall.”) Optimizing memory access provides a substantial opportunity to speed up your system and improve its reliability.

Using a traditional debugger, tracing and optimizing memory access is quite difficult. But the TASKING Embedded Profiler traces all memory accesses in a non-intrusive way and highlights accesses to non-local memory, in correlation with the function names and the source code. It points to where you can start optimizing the source code.

You can choose to run a one-shot trace, which ends when the hardware trace buffer is full, or when the application finishes, or when you manually stop the analysis. Alternatively, you can run a continuous trace, which ends when the application finishes or when you manually stop the analysis. The profiler makes raw trace data available for advanced analysis (an example appears later in this document).

Target Device Resource Usage

The profiler uses the Multi-Core Debug Solution (MCDS) for on-chip trace support. Trace information is stored in a dedicated trace buffer. With an emulation device you can allocate part of the emulation memory (EMEM) as trace buffer memory. The EMEM is divided into blocks of RAM, also called “tiles,” which can be used as calibration or trace memory. Three types of trace memory are supported:

- Trace Calibration Memory (TCM)—used for trace memory or for calibration
- Extended Trace Memory (XTM)—used only for trace memory
- miniMCDS Trace Memory (TRAM)—used for trace memory on production devices equipped with mini-MCDS

For TCM, you can choose which part of the EMEM should be used for tracing. For XTM, both tiles are always used for tracing. Be certain that the same tile memory range used for tracing is not used by the target application, as this can lead to unexpected trace results. The number of tiles vary per emulation device.

AN EXAMPLE WORKFLOW IN THE TASKING EMBEDDED PROFILER

This example shows how the TASKING Embedded 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 profiler
3. Revise application
4. Recompile, reflash, and reprofile to compare results and verify improvement
5. Repeat the workflow until you are satisfied with your application

1. Compile and Flash Target Application

We will investigate the following 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, because you want to measure performance, the application should be compiled with the highest allowed optimization settings and flashed to your target device. Running the application will provide similar information to using a traditional profiler. That is, the execution time of the for loop computed using `clock()` is (for example) 73,754 ticks on a specific device. It seems that not much could go wrong with this application.

2. Smart Profiling

The TASKING Embedded Profiler is a standalone application. Thus, you must create a project that will store both your application data (links to folders containing the target application and sources) and the smart profiling results. This only needs to be done once before you start profiling an application.

After you create the project, select what you want to optimize in the corresponding analysis. The profiler supports several types of analysis, depending on the problem type you are investigating (see the Analysis Configuration Options sidebar). You will typically start with a “Performance Analysis” to get a high-level overview on 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 Run button (see Figure 1). For this example, we’ll change the Result name to “Initial” and then start the analysis by pressing Run.

Analysis Configuration Options

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

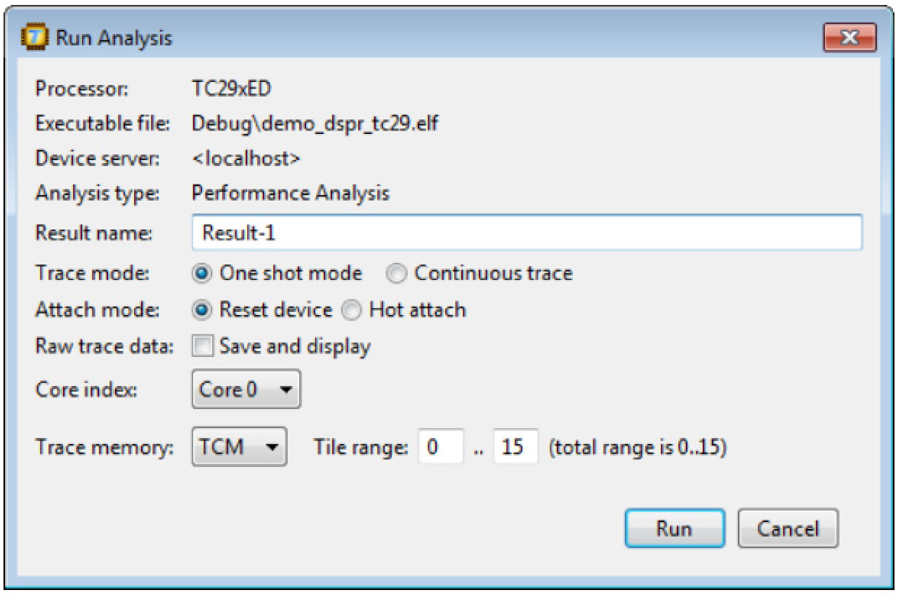
Attach mode: Reset device before measurement or keep current system state.

Raw trace data: Select "Save and display" to see the raw trace data.

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

Trace memory: Select the appropriate type, depending on the device. Options include Trace Calibration Memory (TCM), Extended Trace Memory (XTM), and miniMCDS Trace Memory (TRAM).

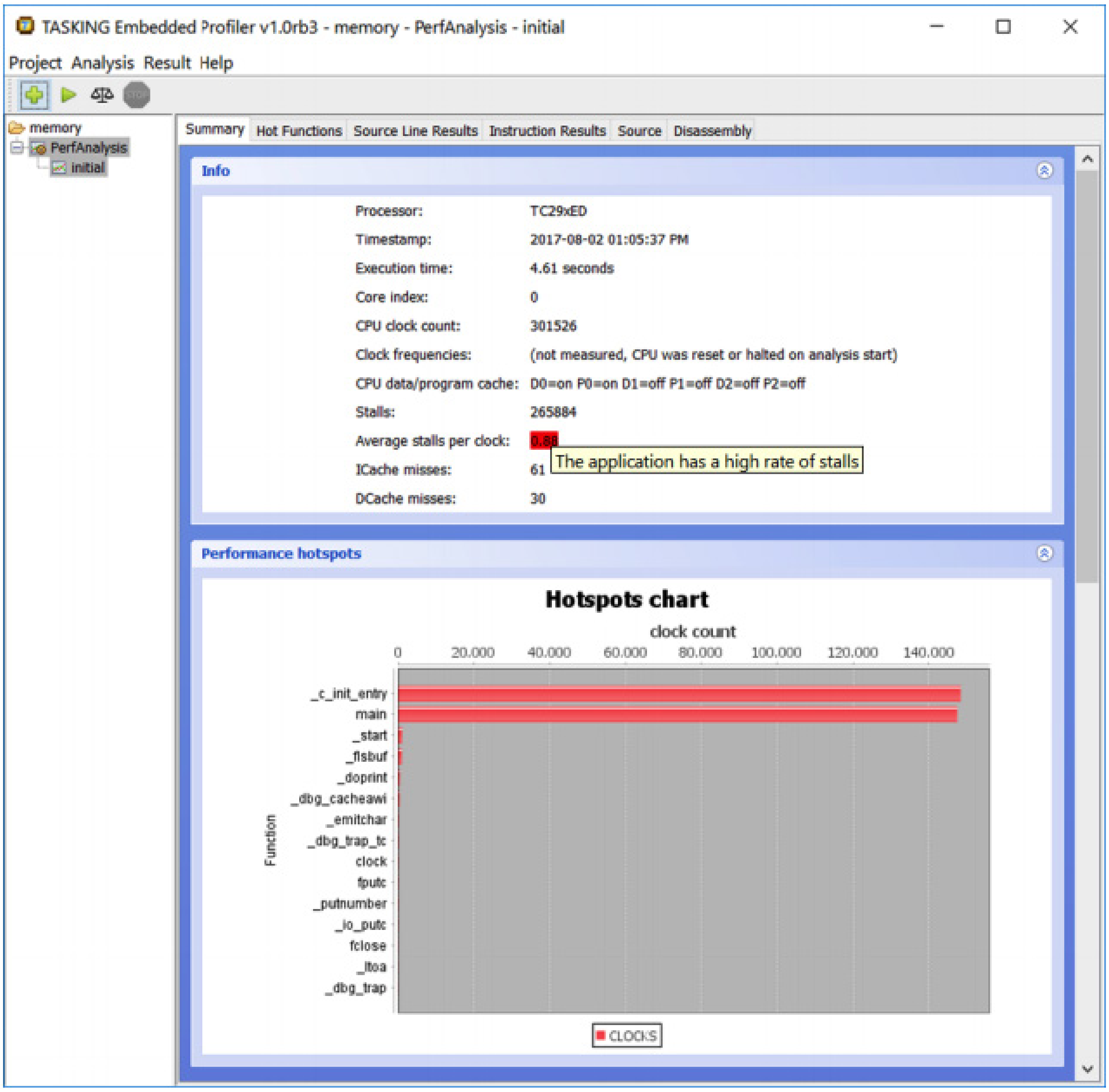
Tile range: Specify which part of the trace memory the profiler should use for tracing (this allows you to allocate parts of the trace memory for other applications).



Processor:	TC29xED
Executable file:	Debug\demo_dspr_tc29.elf
Device server:	<localhost>
Analysis type:	Performance Analysis
Result name:	Result-1
Trace mode:	[✓ One shot mode] Continuous trace
Attach mode:	[✓ Reset device] Hot attach
Raw trace data:	Save and display
Core index:	Core 0
Trace memory:	TCM Tile range: 0 .. 15 (total range is 0..15)

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

Within a few seconds, the profiler collects analysis data from the hardware and computes the results. The analysis summary appears at the top-right side of the screen (see Figure 2). The summary shows that the application spends 88 percent of the execution time stalled (red highlighted key performance indicator); without these stalls it would finish about eight times faster (in about 36K clocks instead of the current 301K clocks).



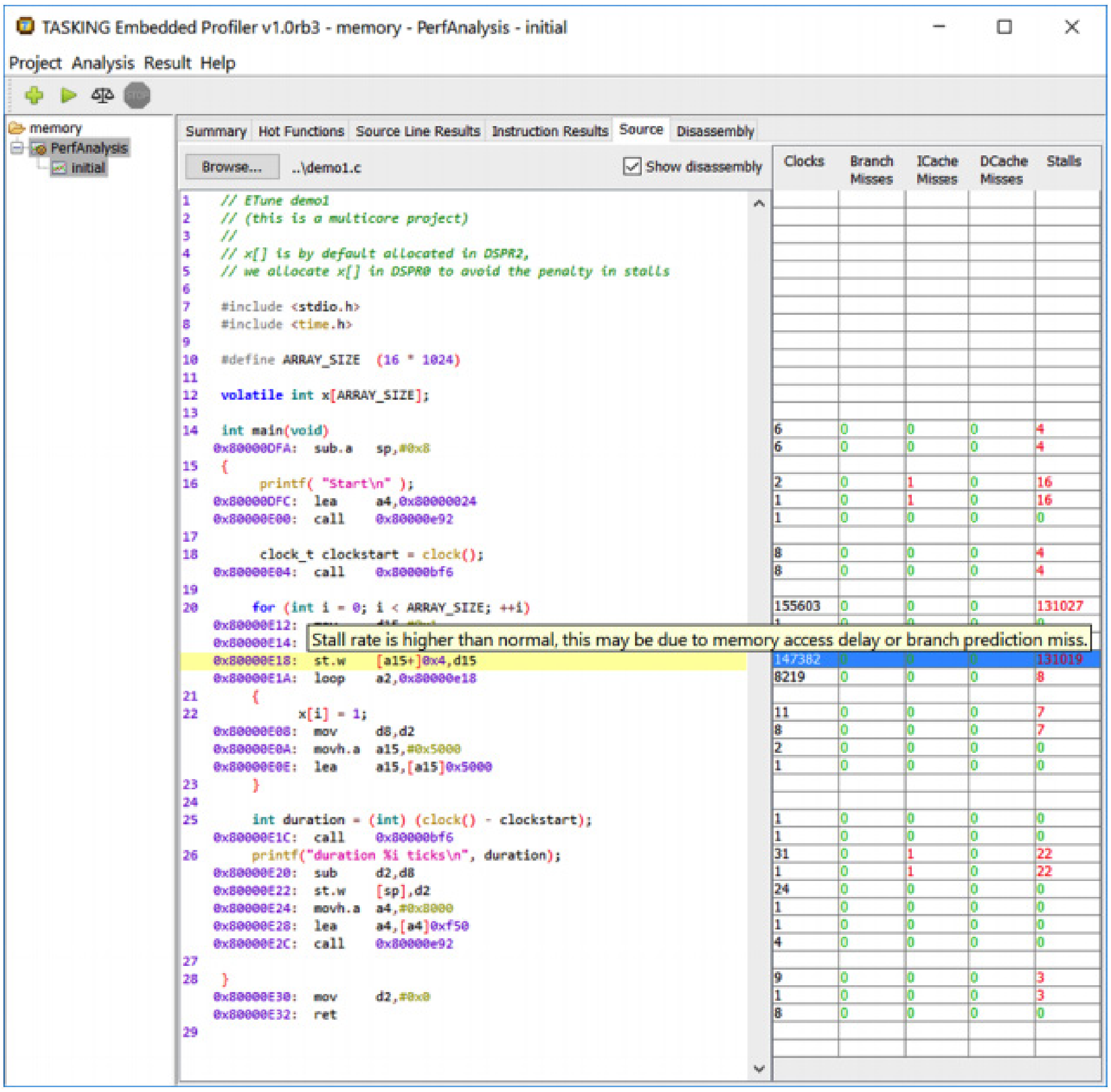
Processor: TC29xED
Timestamp: 2017-08-02 01:05:37 PM
Execution time: 4.61 seconds
Core index: 0
CPU clock count: 301526
Clock frequencies: (not measured, CPU was reset or halted on analysis start)
CPU data/program cache: D0=on P0=on D1=off P1=off D2=off P2=off
Stalls: 265884
Average stalls per clock: 0.88
ICache misses: 61
DCache misses: 30

The application has a high rate of stalls

Hotspots chart functions: _c_init_entry, main, _start, _flsbuf, _doprint, _dbg_cacheawi, _emitchar, _dbg_trap_tc, clock, fputc, _putnumber, _io_putc, fclose, _ltoa, _dbg_trap. Clock count axis: 0, 20.000, 40.000, 60.000, 80.000, 100.000, 120.000, 140.000.

Figure 2: The project tree on the left side allows you to navigate previously configured analyses and analysis results. The right side shows the Performance Analysis summary and Hotspots chart. In this example, the main function generates an unusually high number of stalls, and 88 percent of the CPU time is wasted.

The Hotspots chart shows the functions where most time is spent during program execution. These should be improved first to yield the largest performance gains. Double-clicking on the red bar beside “main” in this chart opens the combined source and analysis results view for function “main” (see Figure 3).



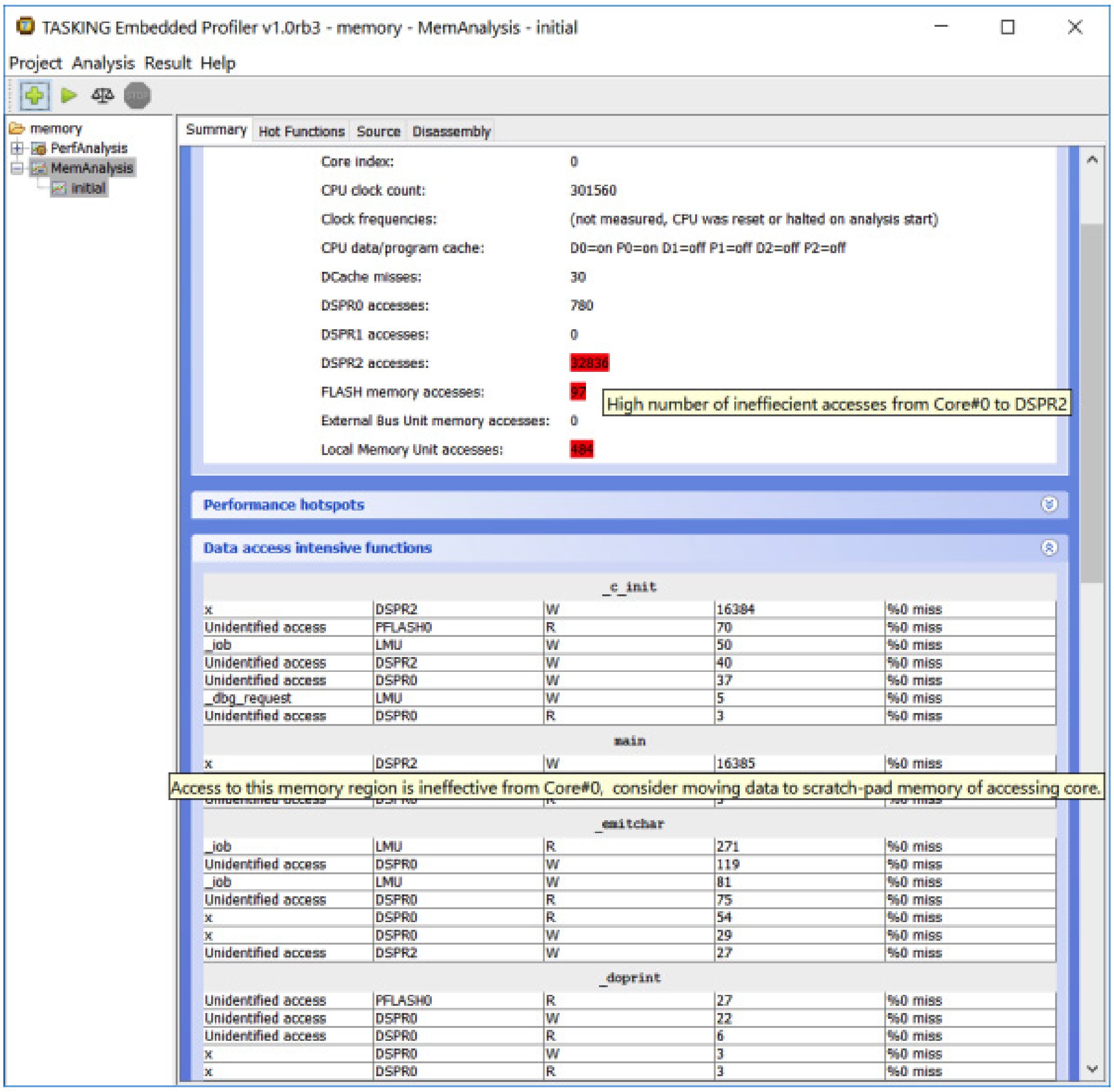
```
1     // ETune demo1
2     // (this is a multicore project)
3     //
4     // x[] is by default allocated in DSPR2,
5     // we allocate x[] in DSPR0 to avoid the penalty in stalls
6
7     #include <stdio.h>
8     #include <time.h>
9
10    #define ARRAY_SIZE  (16 * 1024)
11
12    volatile int x[ARRAY_SIZE];
13
14    int main(void)
    0x80000DFA:  sub.a   sp,#0x8
15    {
16         printf( "Start\n" );
    0x80000DFC:  lea     a4,0x80000024
    0x80000E00:  call    0x80000e92
17
18         clock_t clockstart = clock();
    0x80000E04:  call    0x80000bf6
19
20        for (int i = 0; i < ARRAY_SIZE; ++i)
    0x80000E12:  [illegible]
    0x80000E14:  Stall rate is higher than normal, this may be due to memory access delay or branch prediction miss.
    0x80000E18:  st.w    [a15+]0x4,d15
    0x80000E1A:  loop    a2,0x80000e18
21        {
22             x[i] = 1;
    0x80000E08:  mov     d8,d2
    0x80000E0A:  movh.a  a15,#0x5000
    0x80000E0E:  lea     a15,[a15]0x5000
23        }
24
25        int duration = (int) (clock() - clockstart);
    0x80000E1C:  call    0x80000bf6
26        printf("duration %i ticks\n", duration);
    0x80000E20:  sub     d2,d8
    0x80000E22:  st.w    [sp],d2
    0x80000E24:  movh.a  a4,#0x8000
    0x80000E28:  lea     a4,[a4]0xf50
    0x80000E2C:  call    0x80000e92
27
28    }
    0x80000E30:  mov     d2,#0x0
    0x80000E32:  ret
29
```

Clocks	Branch Misses	ICache Misses	DCache Misses	Stalls
6	0	0	0	4
6	0	0	0	4
2	0	1	0	16
1	0	1	0	16
1	0	0	0	0
8	0	0	0	4
8	0	0	0	4
155603	0	0	0	131027
1	0	0	0	0
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
147382	0	0	0	131019
8219	0	0	0	8
11	0	0	0	7
8	0	0	0	7
2	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
31	0	1	0	22
1	0	1	0	22
24	0	0	0	0
1	0	0	0	0
1	0	0	0	0
4	0	0	0	0
9	0	0	0	3
1	0	0	0	3
8	0	0	0	0

Figure 3: Function-level analysis results for “main”. The stall column from the results table on the right indicates that line 20, which writes the constant 1 into the array x, causes excessive stalls (almost one stall per clock executed on the instruction).

The table on the right of the source view indicates what stalls were measured for the corresponding source line or assembly instruction. Using this table, you 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 instantly identified the most significant slowdown and that the root cause lies in the memory system. To pinpoint the root cause in the memory system and find a mitigation for the problem, we will execute a Memory Analysis. After clicking the Plus button to create a Memory Analysis and running it using the default parameters, the profiler exposes the exact stall cause (see Figure 4).


TASKING Embedded Profiler v1.0rb3 - memory - MemAnalysis - initial
Project Analysis Result Help
memory
PerfAnalysis
MemAnalysis
initial
Summary
Hot Functions
Source
Disassembly
Performance hotspots
Data access intensive functions

Core index:	0
CPU clock count:	301560
Clock frequencies:	(not measured, CPU was reset or halted on analysis start)
CPU data/program cache:	D0=on P0=on D1=off P1=off D2=off P2=off
DCache misses:	30
DSPR0 accesses:	780
DSPR1 accesses:	0
DSPR2 accesses:	32836
FLASH memory accesses:	97
External Bus Unit memory accesses:	0
Local Memory Unit accesses:	484

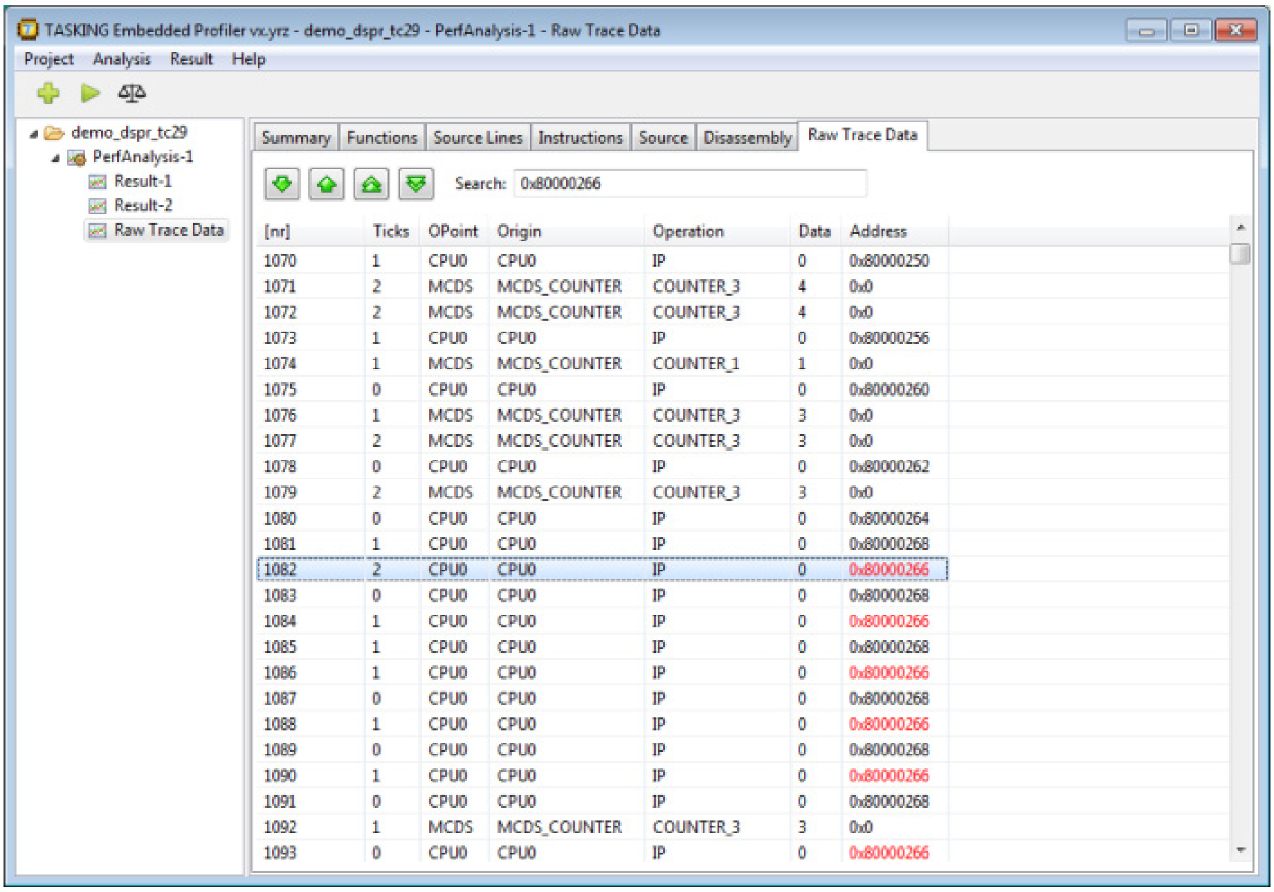
High number of ineffiecient accesses from Core#0 to DSPR2

		_c_init		
x	DSPR2	W	16384	%0 miss
Unidentified access	PFLASH0	R	70	%0 miss
_iob	LMU	W	50	%0 miss
Unidentified access	DSPR2	W	40	%0 miss
Unidentified access	DSPR0	W	37	%0 miss
_dbg_request	LMU	W	5	%0 miss
Unidentified access	DSPR0	R	3	%0 miss
		main		
x	DSPR2	W	16385	%0 miss
Unidentified access	DSPR0	R	3	%0 miss
		_emitchar		
_iob	LMU	R	271	%0 miss
Unidentified access	DSPR0	W	119	%0 miss
_iob	LMU	W	81	%0 miss
Unidentified access	DSPR0	R	75	%0 miss
x	DSPR0	R	54	%0 miss
x	DSPR0	W	29	%0 miss
Unidentified access	DSPR2	W	27	%0 miss
		_doprint		
Unidentified access	PFLASH0	R	27	%0 miss
Unidentified access	DSPR0	W	22	%0 miss
Unidentified access	DSPR0	R	6	%0 miss
x	DSPR0	W	3	%0 miss
x	DSPR0	R	3	%0 miss

Access to this memory region is ineffective from Core#0, consider moving data to scratch-pad memory of accessing core.

Figure 4: The example code is running in core 0. The "DSPR2 accesses" KPI highlighted 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). The 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.

To further examine the program flow, you can access the Raw Trace Data tab (see Figure 5). Raw trace data is useful, for example, to see why stall cycles are assigned to instructions that do not access memory. This can be the case when an instruction is the target of a branch. The Raw Trace Data tab has a search field that you can use to search the address column.



Search: 0x80000266

[nr]	Ticks	OPoint	Origin	Operation	Data	Address
1070	1	CPU0	CPU0	IP	0	0x80000250
1071	2	MCDS	MCDS_COUNTER	COUNTER_3	4	0x0
1072	2	MCDS	MCDS_COUNTER	COUNTER_3	4	0x0
1073	1	CPU0	CPU0	IP	0	0x80000256
1074	1	MCDS	MCDS_COUNTER	COUNTER_1	1	0x0
1075	0	CPU0	CPU0	IP	0	0x80000260
1076	1	MCDS	MCDS_COUNTER	COUNTER_3	3	0x0
1077	2	MCDS	MCDS_COUNTER	COUNTER_3	3	0x0
1078	0	CPU0	CPU0	IP	0	0x80000262
1079	2	MCDS	MCDS_COUNTER	COUNTER_3	3	0x0
1080	0	CPU0	CPU0	IP	0	0x80000264
1081	1	CPU0	CPU0	IP	0	0x80000268
1082	2	CPU0	CPU0	IP	0	0x80000266
1083	0	CPU0	CPU0	IP	0	0x80000268
1084	1	CPU0	CPU0	IP	0	0x80000266
1085	1	CPU0	CPU0	IP	0	0x80000268
1086	1	CPU0	CPU0	IP	0	0x80000266
1087	0	CPU0	CPU0	IP	0	0x80000268
1088	1	CPU0	CPU0	IP	0	0x80000266
1089	0	CPU0	CPU0	IP	0	0x80000268
1090	1	CPU0	CPU0	IP	0	0x80000266
1091	0	CPU0	CPU0	IP	0	0x80000268
1092	1	MCDS	MCDS_COUNTER	COUNTER_3	3	0x0
1093	0	CPU0	CPU0	IP	0	0x80000266

Figure 5: The Raw Trace Data tab is for advanced users who want to examine program flow. Raw trace data is useful, for example, to see why stall cycles are assigned to instructions that do not access memory.

3. Revise Application

The root cause of the biggest performance problem has been identified using two quick measurements, and the analysis results from the profiler are shown in Figures 2 and 3. The application can now 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];`

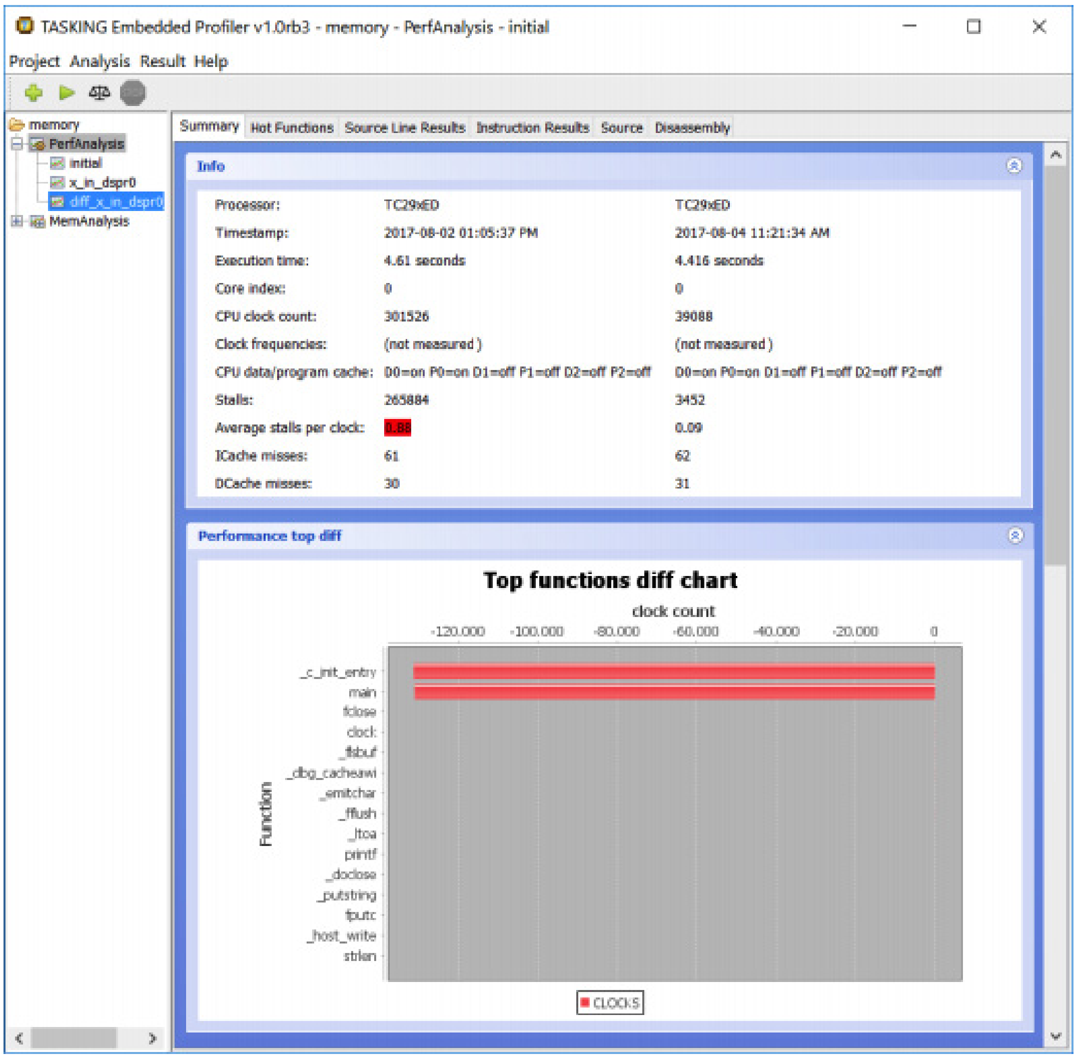
4. Recompile, Reflash, and Reprofile to Verify Improvement

After recompiling and reflashing the application, including the minor change described above, you can execute another Performance Analysis to verify that you fixed the problem and to find new optimization opportunities.

The diff in Figure 6 clearly shows that the improvement was highly successful. The improved application computes the same result as the original one about eight times faster. Average stalls per clock are reduced from almost 90 percent to around 10 percent. No relevant number of new stalls was introduced by the change.

5. Iterative Improvements

After verifying the effectiveness of the first improvement, the next biggest performance issue can be resolved by investigating the results of the second Performance Analysis. You can repeat the entire procedure until all relevant performance issues have been resolved. One iteration typically takes 10 to 20 minutes, and the profiler exposes the problems in order of decreasing severity so that after a few iterations you realize the most serious performance gains.



Processor:	TC29xED	TC29xED
Timestamp:	2017-08-02 01:05:37 PM	2017-08-04 11:21:34 AM
Execution time:	4.61 seconds	4.416 seconds
Core index:	0	0
CPU clock count:	301526	39088
Clock frequencies:	(not measured)	(not measured)
CPU data/program cache:	D0=on P0=on D1=off P1=off D2=off P2=off	D0=on P0=on D1=off P1=off D2=off P2=off
Stalls:	265884	3452
Average stalls per clock:	0.88	0.09
ICache misses:	61	62
DCache misses:	30	31

Figure 6: The profiler computes the difference between analysis results so that you can easily compare results from before and after a modification. This chart shows the diff between the "initial" result and the result "x_in_dspr0" obtained after moving x into DSPR0.

In addition to the memory problem investigated here, the TASKING Embedded Profiler exposes all other types of stall that are visible through the hardware so you feel confident that the hardware is being used to its maximum after eliminating the problems visible in the profiler.

PERFORMANCE REGRESSIONS

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

RETURN ON INVESTMENT

Table 1 shows the cost savings (in terms of time spent to achieve comparable results, the necessary hardware, and the software licenses) when comparing traditional profiling to the TASKING Embedded Profiler solution.

Traditional Profiling	Smart Profiling	Relative Cost
For a typical powertrain project, an expert-level engineer spends about 2 man months optimizing the linker layout alone.	A 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.	The profiler connects to the hardware using a mini-wiggler (costing about 100 EUR) or a simple USB-cable like the one used to charge a 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; in the worst case, weeks might be spent with no overall improvement.	Smart profiling guarantees that the most relevant problems are resolved first, and the problem's root cause is exposed so that 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 is included.	> 2:1

Table 1: Return on Investment (ROI) Details for the TASKING® Embedded Profiler

USAGE SCENARIOS WITH THE TASKING EMBEDDED PROFILER

Typical use cases for smart profiling tools include several scenarios.

Quality Assurance for Software from Suppliers

The profiler enables automated monitoring and verification of the performance characteristics of your suppliers' software. To ensure the best software quality, you can define performance thresholds like maximum stall rate, maximum jitter, or maximum execution time for functions that are developed by a supplier, and then use the profiler to automatically find violations of these criteria for code delivered to you.

Suppliers can use the profiler to ensure 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 are essential. Applying smart profiling early during the development process of your software functions ensures that adverse surprises are not 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 likely to introduce new hardware stalls unwittingly. Instead of hoping for the best and handing your project to the firefighters when it becomes apparent that core utilization or timing behavior become untenable, using smart profiling early puts you in control of your software performance. Deliver better code more quickly and retain control of your code's performance.

Regressions after Code Changes (All Code)

Even minor and seemingly innocent code changes have the potential to introduce several stalls and other performance-degrading events. Make sure that you consciously change your code's performance characteristics and identify mistakes before any harm is done by running automated stall regressions before checking in the code change.

Fire-Fighting Timing and Performance Issues

At a project's end, you may 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 because several side-effects may skew the timing in other event chains. What you really need is to make one specific task or runnable execute faster. If you haven't yet used the profiler to investigate the specific task/runnable, then there is a good chance you can reduce the runtime of the target task/runnable by 50 percent or more with smart profiling—all within an hour. The TASKING Embedded Profiler can target the entire application or a set of specific, user-selected functions.

SUMMARY

The TASKING Embedded Profiler provides smart profiling for deeply embedded devices like the AURIX family of microcontrollers. It gives you 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.

Register for a free evaluation today, and get started with the TASKING Embedded Profiler.

Copyright © 2019 TASKING BV

All rights reserved. You are permitted to print this document provided that (1) the use of such is for personal use only and will not be copied or posted on any network computer or broadcast in any media, and (2) no modifications of the document are made. Unauthorized duplication, in whole or part, of this document by any means, mechanical or electronic, including translation into another language, except for brief excerpts in published reviews, is prohibited without the express written permission of TASKING BV. Unauthorized duplication of this work may also be prohibited by local statute. Violators may be subject to both criminal and civil penalties, including fines and/or imprisonment.

Altium ®, TASKING ®, and their respective logos are registered trademarks of Altium Limited or its subsidiaries. All other registered or unregistered trademarks referenced herein are the property of their respective owners and no trademark rights to the same are claimed.

0219/CMOD/KC/PDF

TASKING®

www.tasking.com