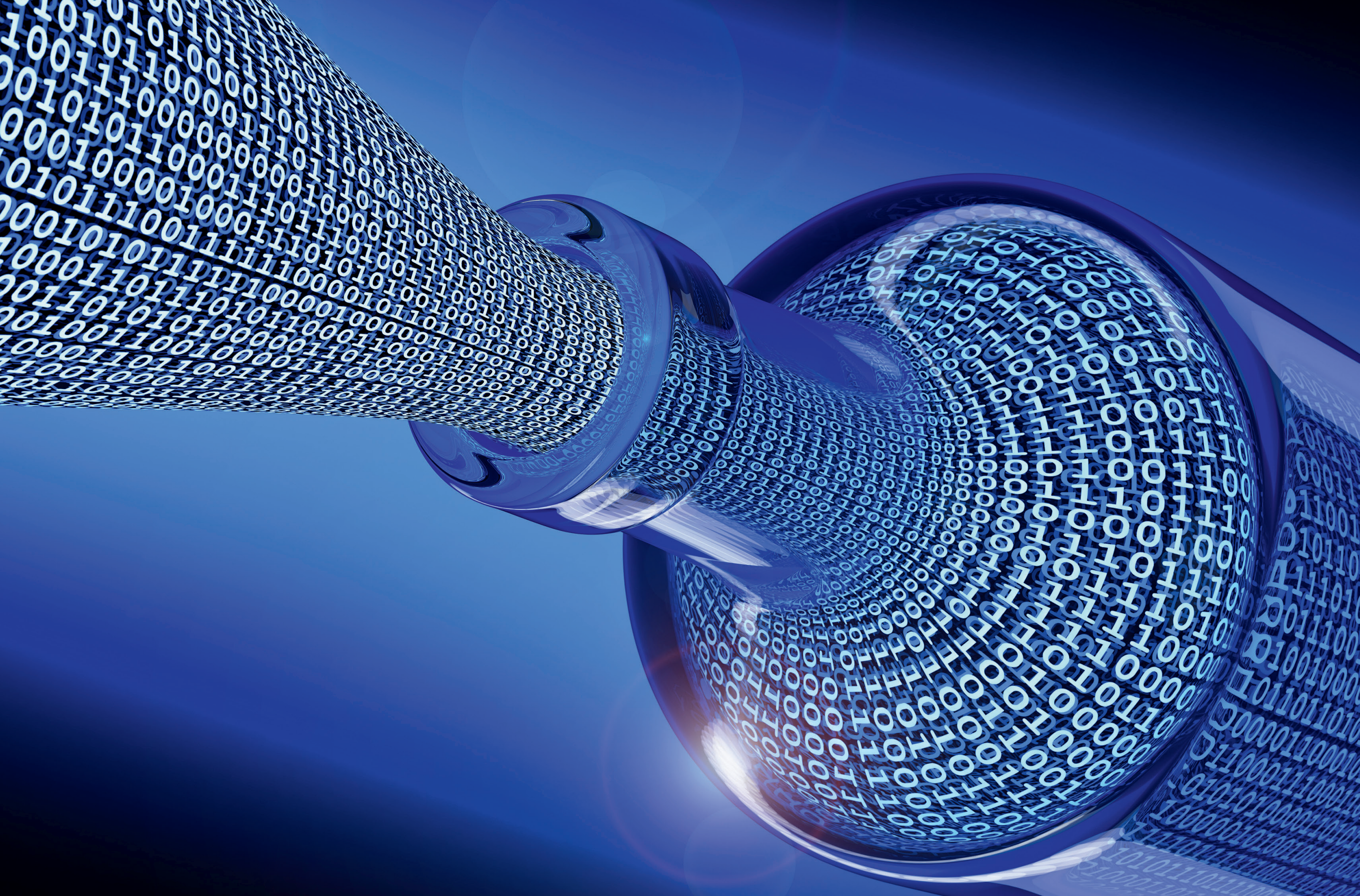


Optimierte Laufzeitanalyse in komplexen Automotive-Applikationen

Bottlenecks auf der Spur

Die Suche nach echten funktionalen Bugs nimmt in der Embedded-Software-Entwicklung erfahrungsgemäß nur wenig Zeit in Anspruch. In den allermeisten Fällen jagt man bei der Fehlersuche stattdessen Performance-Problemen und Verletzungen des vorgegebenen Zeitverhaltens hinterher. Solchen Fehlern kann man zukünftig mit auf Trace basierender Verfahren schneller auf die Spur kommen.

Jens Braunes

Erfreulicherweise lässt sich inzwischen das Laufzeitverhalten von Software für Embedded Systeme während der gesamten Entwicklungs- und Testphase sehr genau beobachten. Die zum Einsatz kommenden hochintegrierten Mikrocontroller oder Embedded-Prozessoren bieten für diesen Zweck dafür extra eine auf den Chips integrierte Trace-Einheit nebst einer passenden Schnittstelle zum Debugger beziehungsweise Trace-Tool an. Gerade bei Echtzeitsystemen und Multicore-Anwendungen, die heutzutage in vielen Automotive-Anwendungen zum Einsatz kommen, liefert Trace wertvolle Informationen, um eine genaue Vorstellung über das Zeitverhalten zu bekommen und die Ursachen für Timing-Probleme zu finden.

On-Chip-Trace

Vereinfacht gesagt, verbergen sich hinter dem sogenannten On-Chip-Trace zusätzliche, direkt an den Rechenkernen angedockte Logikeinheiten, die die ausgeführten Maschinenbefehle aufzeichnen. Das Ganze passiert nichtinvasiv, also ohne dass davon die eigentliche Programmabarbeitung beeinflusst wird. Zusätzlich liefern die Trace-Einheiten Zeitstempel, sodass sich aus den aufgezeichneten Informationen zum Beispiel die Zeit ermitteln lässt, wie lange eine bestimmte Funktion zur Ausführung benötigte.

Für die Aufzeichnung und die spätere Verarbeitung im Trace-Tool steht wie im Fall der Universal Debug Engine (UDE) von PLS entweder ein On-Chip-Trace-Speicher zur Verfügung, oder die Daten werden über eine spezielle Trace-Schnittstelle direkt zum Tool übertragen und dort gespeichert. Dafür haben die jeweiligen Halbleiterhersteller entsprechend breitbandige Trace-Schnittstellen vorgesehen. Fehlt als Gegenstelle nur noch eine entsprechend leistungsfähige Debugger-Hardware mit ausreichend Trace-Speicher. Hierfür bie-

© Carl Hanser Verlag, München

Bild 1: Im Execution Sequence Chart erkennt man schnell die Unregelmäßigkeiten in der Task-Ausführung des Schedulers. Auch die Ursache dafür ist leicht zu identifizieren. © PLS

ten sich je nach Anforderungsprofil wahlweise die Universal Access Devices UAD2next oder UAD3+ mit 512 MByte bzw. bis zu 8 GByte Speicherkapazität an.

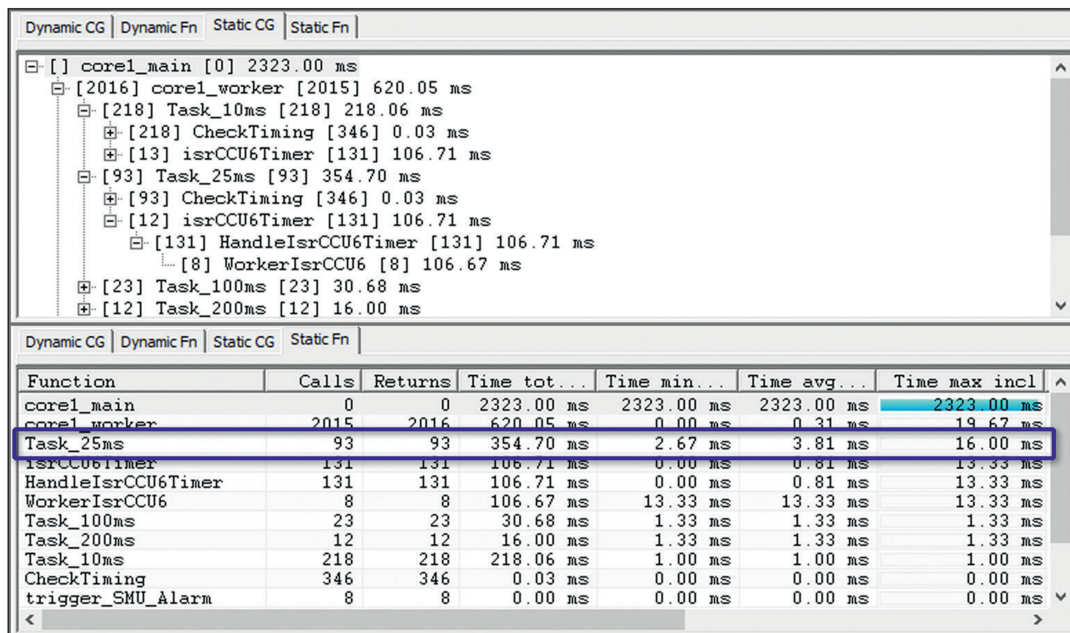
Das Zeitverhalten visualisiert

Die aufgezeichneten Trace-Daten liefern unter anderem wertvolle Informationen, mit deren Hilfe sich das Zeitverhalten der Applikation auf unterschiedliche Art effizient untersuchen lässt. Eine Möglichkeit besteht darin, den zeitlichen Ablauf einer Programmausführung grafisch zu visualisieren. Dazu bietet sich ein Gantt-Diagramm an. Gantt-Diagramme stellen die zeitliche Abfolge von Aktivitäten mit Hilfe von Balken über einer Zeitachse dar. Die UDE nutzt eine solche Darstellung beispielsweise für das Execution Sequence Chart (ESC), das sowohl die Ausführung von Funktionen der Applikation als auch von Tasks eines Betriebssystems visualisiert.

Bild 1 zeigt die Trace-Analyse eines einfachen kooperativen Task-Schedulers. Der Scheduler führt zeitscheibengesteuert vier Tasks mit unterschiedlichen Ausführungsintervallen zwischen 10 und 200 ms aus. Für ein reibungsloses Scheduling ergibt sich somit eine Zeitschranke von 10 ms. Das ist die maximal zulässige Gesamtausführungsdauer aller Tasks innerhalb einer Zeitscheibe.

Im ESC erkennt man sofort Unregelmäßigkeiten in der eigentlich „gleichmäßigen" Ausführungssequenz der Tasks. Ganz offensichtlich gibt es im vorliegenden Beispiel ein Problem mit dem 10-ms-Task, der von Zeit zu Zeit im Diagramm gut sichtbar nicht ausgeführt wird, wie die Pfeile in Bild 1 verdeutlichen. Die Ursache findet man durch die Visualisierung auch recht schnell: Es ist der 25-ms-Task, der mit seiner Aufgabe ab und an nicht rechtzeitig fertig wird. Man kann nun entweder zum Code wechseln und versuchen, die Ursache dafür dort zu finden, oder man nutzt weiterhin das ESC und schaut sich den 25-ms-Task genauer an. Dieser wird nämlich unterbrochen und damit seine Ausführung verzögert. Die Unterbrechung ist durch das Label „Stack" und den Wechsel der Balkenfarbe gekennzeichnet. Das kann entweder bedeuten, dass hier regulär eine Unterfunktion gerufen wurde oder aber ein Interrupt auftritt. Im vorliegenden Beispiel hilft die Betrachtung der anderen Funktionseinträge weiter. Hierfür wird der Cursor auf einen Zeitpunkt in der Nähe des Wechsels von „Running" in „Stack" der 25-ms-Task-Funktion platziert. Anschließend überprüft man, welche der anderen Funktionen zu diesem Zeitpunkt in den „Running"-Zustand wechseln. Hier ist es die Behandlungsfunktion „WorkerIsrCCU6" des Timer Interrupts der Capture Compare Unit 6 (CCU6) des für das Beispiel verwendeten Aurix-Bausteins von Infineon.

Die Nutzung des Execution Sequence Charts ist insbesondere dann sinnvoll, wenn sich mit Hilfe der Visualisie-



Function	Calls	Returns	Time tot...	Time min...	Time avg...	Time max incl
core1_main	0	0	2323.00 ms	2323.00 ms	2323.00 ms	2323.00 ms
core1_worker	2015	2016	620.05 ms	0.00 ms	0.31 ms	19.67 ms
Task_25ms	93	93	354.70 ms	2.67 ms	3.81 ms	16.00 ms
isrCCU6Timer	131	131	106.71 ms	0.00 ms	0.81 ms	13.33 ms
HandleIsrCCU6Timer	131	131	106.71 ms	0.00 ms	0.81 ms	13.33 ms
WorkerIsrCCU6	8	8	106.67 ms	13.33 ms	13.33 ms	13.33 ms
Task_100ms	23	23	30.68 ms	1.33 ms	1.33 ms	1.33 ms
Task_200ms	12	12	16.00 ms	1.33 ms	1.33 ms	1.33 ms
Task_10ms	218	218	218.06 ms	1.00 ms	1.00 ms	1.00 ms
CheckTiming	346	346	0.03 ms	0.00 ms	0.00 ms	0.00 ms
trigger_SMU_Alarm	8	8	0.00 ms	0.00 ms	0.00 ms	0.00 ms

Bild 2: Die Call-Graph-Visualisierung der UDE bietet neben der Graphen-Darstellung oben im Bild auch eine Zusammenfassung der gesammelten Profiling-Informationen. Mithilfe der Sortierung nach der maximalen Laufzeit sind Zeitfresser schnell ermittelt. © PLS

Dynamic CG | Dynamic Fn | Static CG | Static Fn

Function	Tick	Calls	Returns	Time tot incl	Time min...
core1_worker	380.00 ms	1	1	16.00 ms	
core1_worker	980.00 ms	1	1	16.00 ms	
Task_25ms	1280.00 ms	1	1	16.00 ms	
Task_25ms	680.00 ms	1	1	16.00 ms	
Task_25ms	80.00 ms	1	1	16.00 ms	
Task_25ms	256.00 ms	1	1	16.00 ms	
Task_25ms	1580.00 ms	1	1	16.00 ms	
Task_25ms	38[illegible].00 ms	1	1	16.00 ms	
Task_25ms	980.00 ms	1	1	16.00 ms	
Task_25ms	556.00 ms	1	1	16.00 ms	
HandleIsrCCU6Timer	258.47 ms	1	1	13.33 ms	
isrCCU6Timer	258.47 ms	1	1	13.33 ms	

Dynamic CG | Dynamic Fn | Static CG | Static Fn

```
[1] core1_worker [1] 0.00 ms
[1] core1_worker [1] 0.00 ms
[1] core1_worker [1] 0.00 ms
[1] core1_worker [1] 16.00 ms
  [1] Task_25ms [1] 16.00 ms
    [1] CheckTiming [1] 0.00 ms
    [1] isrCCU6Timer [1] 13.33 ms
      [1] HandleIsrCCU6Timer [1] 13.33 ms
        [1] WorkerIsrCCU6 [1] 13.33 ms
[1] core1_worker [1] 0.00 ms
[1] core1_worker [1] 0.00 ms
[1] core1_worker [1] 0.00 ms
```

***Bild 3:** Im Timeline-Modus können Entwickler gezielt das abnormale/unerwünschte Zeitverhalten untersuchen. Per Navigationsfunktion finden sie sofort die relevante Stelle im Call-Graphen und identifizieren dort schnell die eigentliche Ursache.*

© PLS

rung alleine schon durch reines Betrachten ein ungewöhnliches oder abweichendes Verhalten der Ausführungssequenz ableiten lässt. Wenn sehr viele Daten dargestellt werden müssen, also entweder sehr viel Funktionen beteiligt sind oder es sich um einen großen beobachteten Zeitbereich handelt, werden allerdings schnell die Grenzen von Execution Sequence Charts sichtbar, und das im wahrsten Sinne des Wortes.

Auf den Call-Graphen geschaut

Für diesen Fall empfiehlt sich eine andere Option, um Probleme im Zeitverhalten aufzudecken: die detaillierte Analyse des Call-Graphen. Ein Call-Graph zeigt zunächst einmal die Aufrufbeziehungen zwischen den Funktionen eines Programms – also welche Funktion wird durch welche anderen Funktionen aufgerufen, oder umgekehrt, welche Funktion ruft welche anderen Funktionen auf. Solche Call-Graphen lassen sich über statische Code-Analysen oder, auf einer realen Ausführung basierend, auch mittels Trace ermitteln. Gerade die letztere Variante kann interessante Erkenntnisse insbesondere beim Auftreten von Interrupts liefern.

Neben den reinen Ausrufbeziehungen können Call-Graphen auch weitere wichtige Informationen liefern, zum Beispiel die Anzahl der Aufrufe oder Profiling-Informationen, d. h. Zeitinformationen zu den einzelnen Ausführungen sowie zur minimalen, maximalen und durchschnittlichen Laufzeit einer Funktion. Diese und weitere Informationen liefern wertvolle Anhaltspunkte, um etwaige Zeitfresser zu identifizieren.

Bild 2 visualisiert die Analyseergebnisse der beschriebenen Applikation, wie sie in der UDE dargestellt werden. Der obere Teil zeigt den den Call-Graphen, der Screenshot unten die Profiling-Informationen für jede beteiligte Funktion. Sortiert nach der maximalen Laufzeit, wird schnell klar, dass der 25-ms-Task für die eigentlich erlaubte Zeitschranke von maximal 10 ms für die Ausführung aller Tasks in einigen Fällen zu lange dauert. Die maximale Laufzeit übersteigt mit 16 ms diese Grenze deutlich. Nur warum ist das so? Klären lässt sich das nur, indem man diejenigen Ausführungen des 25-ms-Tasks, die zu lange dauern, genauer unter die Lupe nimmt. Sehr hilfreich erweist sich hier der Timeline-Modus, eine spezielle Option in der Call-Graph-Analyse der UDE. Wird für die normale Call-Graph-Darstellung eine konkrete Call-Beziehung in einem einzigen Knoten zusammengefasst und die tatsächliche Anzahl der Aufrufe dafür lediglich annotiert, kommt im Timeline-Modus für jeden einzelnen Aufruf ein neuer Knoten hinzu. Für jeden dieser Knoten sind dann auch die zur zugehörigen Ausführung gehörenden Profiling-Informationen dargestellt.

Ist der Timeline-Modus erst einmal aktiviert, muss in der Profiling-Ansicht nur noch nach der Ausführungszeit sortiert werden (**Bild 3**). Ziemlich weit oben in der sortierten Liste sind nun die zu lang dauernden Ausführungen des 25-ms-Tasks zu finden. Um diese näher zu untersuchen, reicht es ausgehend von einer dieser Ausführungen über die Navigationsfunktion „Show in Graph" in den Call-Graphen zu wechseln. Dieser zeigt nun alle aufgerufenen Funktionen für die zu untersuchende Ausführung des 25-ms-Tasks. In Bild 3 unten ist der entsprechende Teilbaum des Call-Graphen bereits ausgeklappt. Man sieht, dass die eigentliche Task-Funktion nicht sonderlich zeithungrig ist. Auffällig hingegen ist die bereits bekannte Behandlungsfunktion des CCU6-Timer-Interrupts, die offensichtlich während der Task-Ausführung „zuschlägt" und das vergebene Zeitregime durcheinanderbringt. Das hatte sich bereits in der Visualisierung im ESC gezeigt. Eine mögliche Lösung des Problems im vorliegenden Falle wäre, die offensichtlich recht aufwendige Interrupt-Behandlung auf einen anderen Core zu verlagern, der weniger zeitkritische Applikationsteile ausführt.

Die aufgezeigten Beispiele verdeutlichen, dass gerade bei Software-Tests im Bereich echtzeitkritischer Fahrzeugelektronik Trace-basierte Methoden zur Laufzeitanalyse in vielen Fällen einen hohen Mehrwert bieten. Das gilt insbesondere dann, wenn die reale Ausführung nur ungenügend durch statische Code-Analyse ermittelt werden kann und die Applikation stark Interrupt-getrieben ist. ■ (eck)

www.pls-mc.com

Jens Braunes ist Product Marketing Manager bei PLS Programmierbare Logik & Systeme. © PLS

© Carl Hanser Verlag, München