Was wird nur aus meinem Code?
Software-Performance endlich fundiert bewerten

Daniel Penning, embeff GmbH

Die Performance von Software spielt bei nahezu jedem Embedded-Projekt eine entscheidende Rolle. Schneller Code führt zu besseren Reaktionsraten und höherem Systemdurchsatz. Eine spezifizierte Aufgabenstellung kann so gegebenenfalls mit weniger Leistung und dementsprechend kleinerem Mikrocontroller bewältigt werden. Der Energiebedarf sinkt und führt so insbesondere bei batteriebetriebenen Systemen zu längerer Laufzeit bzw. einer geringeren Dimensionierung der Batteriekapazität. Diese Effekte resultieren schlussendlich in einer günstigeren Hardware.

Im Kontext dieser positiven Wirkungskette erstaunt es, dass bei vielen Projekten dem Zusammenhang zwischen einzelnen Softwareteilen und der resultierenden Performance wenig Beachtung geschenkt wird. Hochoptimierende Compiler und innovative Prozessor-Instruktionen bieten inzwischen ein enormes Potential, performanten Code zu schreiben.

Dennoch werden im Embedded Umfeld viele Diskussionen von Pauschalisierungen und Vorurteilen geprägt. Abbildung 1 zeigt exemplarisch Techniken, gegen die oft im Namen der Performance Bedenken erhoben werden.

Technik	**Ungenutzte positive Effekte**
Konsequente Kapselung in neue Typen & Module (Abstraktion)	Wiederverwendbarkeit, Wartbarkeit
Einsatz externer Bibliotheken	Weniger Fehler durch erprobte Implementierungen, reduzierte Time-2-Market, höhere Performance
Moderne C++ Sprachfeatures	Wiederverwendbarkeit, Fehler bereits während der Implementierung finden, höhere Performance

Abbildung 1: Techniken des modernen Software-Engineerings

Die grundlegende Ablehnung dieser Techniken verhindert eine Innovation im Embedded Software Engineering, die für die stetig komplexer werdenden Aufgaben zwingend erforderlich ist.

Performance-Bewertung für Embedded

Eine Performance-Bewertung für Embedded-Code erweist sich aus diversen Gründen als schwierig:

- Es gibt sehr verschiedene Ziel-Architekturen mit deutlich unterschiedlichem Laufzeitverhalten.
- Profiling ist oft nur mit teuren Tools und einer speziellen Hardware möglich.
- Das Einrichten einer Profiling-fähigen Umgebung kann komplex sein.
- Die Ziel-Hardware muss Features zur Performance-Bewertung aufweisen.

Im Folgenden soll gezeigt werden, wie eine Performance Bewertung mit einfachen Mitteln exemplarisch realisiert werden kann.

Performance-Bewertung für ARM Cortex-M4
Mikrocontroller der ARM Cortex-M4 Reihe sind von verschiedensten Herstellern lizensiert und vielseitig einsetzbar. Die zugrunde liegende armv7m-Architektur [1] soll daher hier als Ausgangspunkt für eine Betrachtung dienen.

In diesen Prozessoren kann optional eine „Data Watchpoint and Trace Unit“ (DWT) [2] vom Hersteller vorgesehen werden. In den allermeisten Modellen ist dies der Fall.
Die DWT unterstützt das Auslesen von Performance-Registern.

CMSIS Register	Beschreibung
DWT_CYCCNT	Cycle Count Register
DWT_CPICNT	CPI Count Register
DWT_EXCCNT	Exception Overhead Count Register
DWT_SLEEPCNT	Sleep Count Register
DWT_LSUCNT	LSU Count Register
DWT_FOLDCNT	Folded-instruction Count Register

Abbildung 2: ARM DWT Register

Für eine Performance-Messung einzelner Code-Teile kann das DWT_CYCCNT Register benutzt werden. Dieses Register zählt taktgenau die Zyklen. Es stellt damit die genaueste Einheit dar, die prinzipiell auf einem Prozessor gemessen werden kann. Durch die bei Embedded-MCUs übliche feste Taktfrequenz kann bei Bedarf aus einer Anzahl Zyklen auf die absolute Zeit zurückgerechnet werden.

In Pseudocode gestaltet sich eine Messung also wie folgt:

```
preCycleCount = DWT->CYCCNT
CodeUnderTest(<Parameter>); // Laufzeit messen
postCycleCount = DWT->CYCCNT
cyclesUsed = postCycleCount – preCycleCount
```

Abbildung 3: Pseudocode zur Laufzeit-Messung

So könnte man zur Laufzeit im Debugger die cyclesUsed Variable auslesen und hätte das gewünscht Ergebnis. Dabei gibt es jedoch zwei Probleme:

- Bei eingeschalteter Optimierung sortiert der Compiler ggf. Lese-Zugriffe auf das DWT_CYCCNT Register um.
- Auf Assembly-Ebene verfälschen die Load/Store Anweisungen aus dem DWT_CYCCNT Register in ein internes Prozessor-Register die Messergebnisse.

Ein besserer Weg ist daher die Verwendung einer speziellen HALT-Instruktion, die den Prozessor direkt vor- und nach Ausführung der Messung ohne Seiteneffekte anhält. In armv7m gibt es dazu die BKPT-Instruktion. Zu diesem Zeitpunkt kann bspw. per SWD-Schnittstelle [3] über eine Debug-Probe das DWT_CYCCNT Register ausgelesen werden. Der Pseudocode reduziert sich damit auf:

```
BKPT //< Extern CYCCNT lesen
CodeUnderTest(<Parameter>)
BKPT //< Extern CYCCNT lesen
```

Abbildung 4: Verbesserter Pseudocode zur Laufzeit-Messung

Mit dieser Variante können für beliebige Programmteile Zyklus-genaue Laufzeiten bestimmt werden. Bei 100MHz Taktfrequenz liegt die zeitliche Auflösung beispielsweise bei beachtlichen 10ns.

Compiler-Optimierungen und Performance-Messungen
Der Compiler führt bei eingeschalteter Optimierung Maßnahmen zur Performance-Verbesserung des Codes durch. Eine der wirkungsvollsten Techniken ist dabei, Verzweigungen in kurze Funktionen mit dem eigentlichen Funktionsinhalt zu ersetzen. Dies wird als Inlining bezeichnet. Weiterhin wird der Compiler versuchen, möglichst viele Werte bereits selbst – während der Kompilierung - zu berechnen.
So kann es leicht passieren, dass der Compiler einen zu messenden Funktionsaufruf selbst völlig herausoptimiert.
Der generelle Verzicht auf Optimierung ist keine Lösung, da gerade diese Maßnahmen einen Großteil zur Gesamtperformance beitragen.
Es gibt verschiedene Wege solche Optimierungen nur lokal gezielt zu unterbinden. Die Dokumentation der Google Benchmark Bibliothek [4] zeigt dazu interessante Möglichkeiten auf.

Beispiel: FPU gegen Soft-FPU
Ein einfaches Beispiel soll zeigen, wie mit dem oben vorgestellten Ansatz grundlegend Performance auf einem sehr feinen Level evaluiert werden kann. Dazu soll die Laufzeit einer einzelnen Funktion gemessen werden.

Die zu testende Funktion (Function Under Test, FUT) multipliziert lediglich einen ganzzahligen Eingangswert mit der Kreiszahl in einfacher Fließkomma-Genauigkeit.

```
int fut(int input) {
   return input * 3.14159265359f;
}
```

Abbildung 5: Funktion, deren Laufzeit gemessen werden soll

Die Beispiele wurden mit der arm-none-eabi-gcc Toolchain (Version 7-2017-q4-major) und eingeschalteter Optimierung (O2) auf einem STM32F4 ausgeführt.
Der verwendete Mikrocontroller hat eine eingebaute FPU für Fließkommazahlen. Wird diese per Compiler-Option abgeschaltet, muss die Multiplikation in Software nachgebildet werden.

Mit FPU		Ohne FPU (Soft-FPU)
`fut(int):`	`# Zyklen`	`fut(int):`
`vmov s15, r0@int`	`# 1`	`push {r3, lr}`
`vldr.32 s14, .L3`	`# 2`	`bl __aeabi_i2f`
`vcvt.f32.s32 s15, s15`	`# 1`	`ldr r1, .L4`
`vmul.f32 s15, s15, s14`	`# 1`	`bl __aeabi_fmul`
`vcvt.s32.f32 s15, s15`	`# 1`	`bl __aeabi_f2iz`
`vmov r0, s15@int`	`# 1`	`pop{r3, pc}`
`bx lr`	`# 2-4`	`.L4:`
`.L3:`		`.word1078530011`
`.word 1078530011`		

Abbildung 6: Assembly Listing für zu messende Funktion

Abbildung 6 zeigt das Assembly Listing für beide Varianten.
Man erkennt bereits, dass in der Soft-FPU Variante Sprünge in die FPU-Emulationen vorhanden sind. Diese Funktionen werden von der Toolchain in der Regel nur als kompilierter Objektcode ausgeliefert. Bei proprietären Compilern ist deren Implementierung also unbekannt und kann lediglich aus dem Assembly reverse-engineered werden. Insbesondere ist also nicht klar, ob diese Funktionen eine konstante Laufzeit aufweisen.

Bei der FPU-Variante dagegen sind keine Sprünge notwendig – alle Operationen können direkt von Instruktionen übernommen werden. Hinter dem Assembly wurden hier die benötigten Zyklen pro Instruktion aus dem Reference Manual [5] entnommen und notiert. Lediglich bei der Branch-Instruktion sind die Zyklen nicht deterministisch (2-4), da je nach Alignment unterschiedlich viele Zyklen für einen Refill der Pipeline nötig sind.

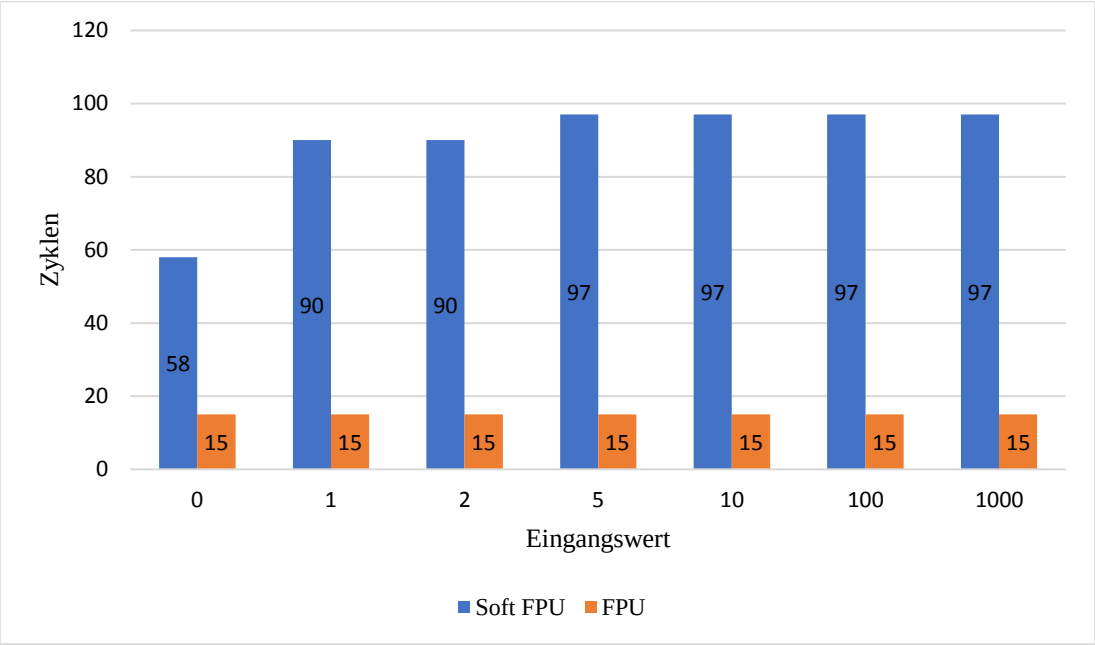

Abbildung 7: Benötigte Zyklen für ausgewählte Eingangswerte

Tatsächlich zeigt sich bei Messung der Laufzeiten (Abbildung 7), dass die FPU-Variante eine konstante Laufzeit hat, die emulierte Variante dagegen variabel ist. Die 15 Zyklen ergeben sich aus den vorhergesagten 9-11 Zyklen plus wiederum 2-4 Zyklen, die für den Sprung in die Funktion selbst benötigt werden. Die Branch-Instruktion benötigt hier also gemessen jeweils 4 Zyklen.

Bei kritischen Programmstellen ist es wichtig über Laufzeit-variable Programmteile Kenntnis zu haben. In diesen Fällen muss für die Ermittlung der Worst-Case-Execution-Time (WCET) der längst mögliche Pfad ausgewählt werden. Eine einzelne Messung hätte hier leicht zu plausibel erscheinenden, aber falschen Schlüssen geführt.

Zusammenfassung
Die vorgestellte Methodik eignet sich dazu, Funktionen und Code-Fragmente einer genauen Performance-Messung zu unterziehen. Die hochgenaue zeitliche Auflösung erlaubt Untersuchen für alle Einsatzzwecke, insbesondere auch kritischer Interrupt Service Routinen und Regelschleifen.

Eine solche Methodik liefert die notwendige Grundlage, die in Tabelle 1 genannten Software Techniken im Einzelfall einer Bewertung zu unterziehen. Zyklen-genaue Ergebnisse ermöglichen eine fundierte Aussage über die Anwendbarkeit von Sprachen, Bibliotheken und Designfeatures. Wenn Kompromisse notwendig werden, können Entscheidungen auf Basis realer Daten erfolgen.

Hinweis: Der Autor betreibt eine kostenfreie Web-Plattform zur komfortablen Performance-Auswertung kleiner Code-Fragmente [6].

Quellen

[1] ARMv7-M Reference Manual
https://static.docs.arm.com/ddi0403/eb/DDI0403E_B_armv7m_arm.pdf

[2] Data Watchpoint and Trace Unit
https://developer.arm.com/docs/ddi0439/latest/data-watchpoint-and-trace-unit/dwt-programmers-model

[3] ARM Serial Wire Debug
https://www.arm.com/files/pdf/Serial_Wire_Debug.pdf

[4] google Benchmark Bibliothek
https://github.com/google/benchmark

[5] Cortex-M4 Reference Manual
http://infocenter.arm.com/help/topic/com.arm.doc.100166_0001_00_en/arm_cortexm4_processor_trm_100166_0001_00_en.pdf

[6] Online Plattform für MCU Performance Messungen
https://barebench.com

Autor

Daniel Penning studierte Elektrotechnik in Karlsruhe und ist Geschäftsführer der embeff GmbH. Er verfügt über mehr als 10 Jahre Erfahrung in verschiedenen Bereichen der Softwareentwicklung. Inzwischen konzentriert er sich ausschließlich auf die speziellen Anforderungen eingebetteter Systeme. Dabei ist ihm besonders die Effizienz von Produkten und Entwicklungsprozessen eine Herzensangelegenheit.

Kontakt

Email: daniel.penning@embeff.com