# Praktische Tipps und Tricks für die Laufzeitoptimierung
## Optimierungsansätze auf RTOS-/Codeebene; Single-/Multicore

Peter Gliwa, GLIWA GmbH embedded systems

**Die Ressource „Rechenzeit“ wird in vielen Projekten im Verlauf der Entwicklung knapp. Im Folgenden sollen einige praktische Ansätze beleuchtet werden, um in solchen Situationen Steuergerätesoftware hinsichtlich der Laufzeit zu optimieren. Zum anderen werden Maßnahmen angesprochen, um frühzeitig beim Design, bei der Konfiguration und Implementierung Laufzeitprobleme zu verhindern.**

In den letzten 20 Jahren hat sich mit Blick auf das Timing bei der Entwicklung von Steuergerätesoftware viel getan. Wurde das Timing früher meist nur dann explizit betrachtet, wenn es Probleme verursachte, wird dem Timing heute oft systematisch und frühzeitig Aufmerksamkeit zuteil. Die Betriebssystemkonfiguration wird beispielsweise nicht einfach vom Vorgängerprojekt übernommen und hier und da angepasst. Stattdessen finden eingehende Überlegungen statt, welche Anforderungen an das Timing existieren und wie diese mit einer geeigneten Betriebssystemkonfiguration, Verteilung von Tasks auf die verschiedenen Kerne eines Multicoreprozessors etc. begegnet werden kann.

Immer öfter kommen dabei Techniken wie die Schedulingsimulation oder Schedulinganalyse zum Einsatz. Dennoch kommt praktisch jedes Projekt an einen Punkt, bei dem das Verhalten des *realen* Systems mitunter gravierend vom *erwarteten* Verhalten – dem simulierten oder modellierten Timing – abweicht. Hier hilft nur eine Analyse der realen Systems, falls erforderlich in der realen Umgebung, also im Fahrzeug.

**Laufzeitanalyse: welche Technik für welche Situation?**
Bevor wir uns den konkreten Timinganalysetechniken zuwenden, soll verdeutlicht werden, dass bei der Timinganalyse grundsätzlich zwei Fragen vorangestellt werden sollten. Erstens: in welcher Phase befindet sich das Projekt? Mögliche Phasen sind „früh“ oder „spät“ beziehungsweise „Design“, „Implementierung“ oder „Verifikation“, um es mit den Begriffen des V-Modells zu sagen. Zweitens: auf welcher Abstraktionsebene soll Timinganalyse betrieben werden? Möglich Ebenen sind „Codeebene“, „Schedulingebene“ oder „Netzwerkebene“. Es lässt sich feststellen, dass sich jede Auseinandersetzung mit einem konkreten Timingaspekt in einem Koordinatensystem mit den Achsen „Phase/Projektlaufzeit“ und „Ebene“ verorten lässt. Abbildung 1 stellt dies anhand von V-Modellen auf der jeweiligen Ebene dar.

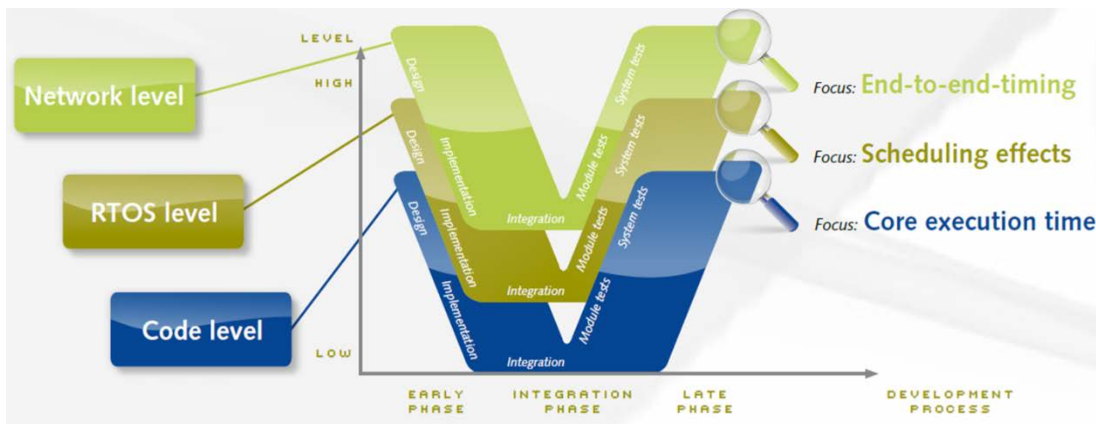


Abb. 1: Timing auf verschiedenen Ebenen und in verschiedenen (Projekt-) Phasen

Die gleichen Ebenen finden sich in Abbildung 2 unterhalb der Granularitätsachse wieder. Darüber sind die verschiedenen Timinganalysetechniken aufgetragen; es besteht eine gewisse Zuordnung anhand der horizontalen Position. Die „statische Codeanalyse“ beispielsweise findet auf der Codeebene statt und deckt den Bereich von „Opcode States“ bis hin zu „TASK/ISR“ ab.

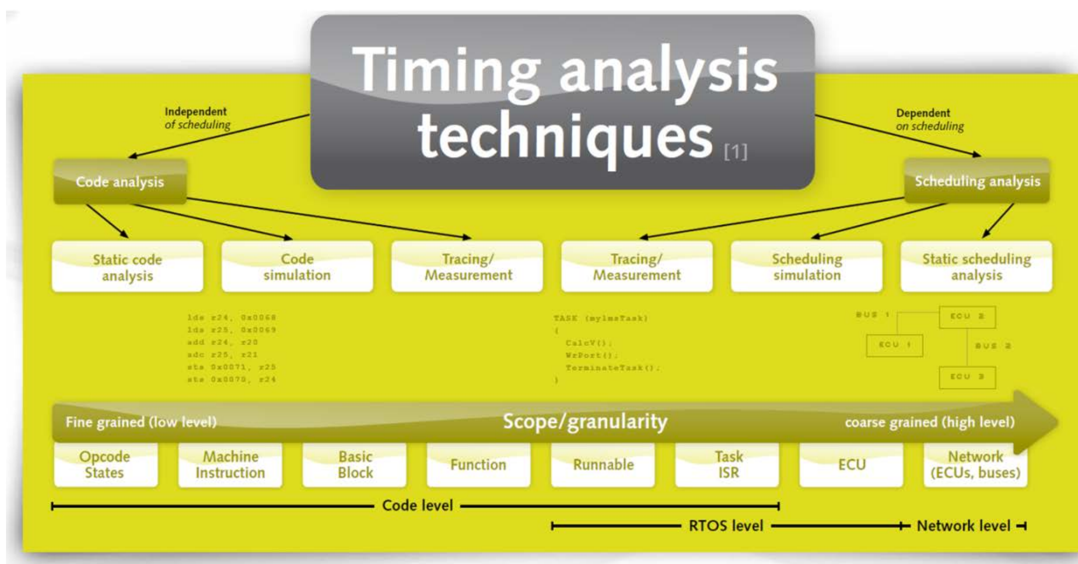


Abb. 2: Übersicht der Timinganalysetechniken

Abbildung 3 ergänzt die Übersicht um die Aspekte „Art der Timinganalyse“ und die bereits angesprochene (Projekt-) Phase.

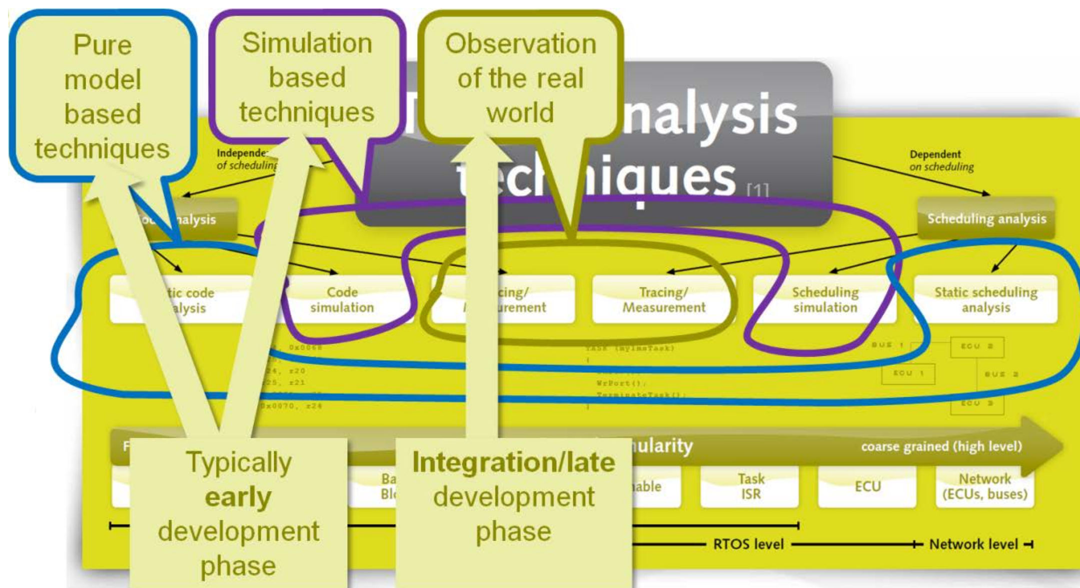


Abb. 3: Übersicht der Timinganalysetechniken ergänzt um weitere Aspekte

Eine kurze Beschreibung der Funktionsweise der verschiedenen Analysetechniken findet sich auf dem Timingposter [1], dem auch die bisherigen Abbildungen entnommen sind.

Die folgende Liste zeigt Anwendungsfälle („Use-cases“) für die verschiedenen Timinganalysetechniken auf.

- Statische Codeanalyse
    - **Use-Case:** Ermittlung der WCET (worst-case core execution time) unabhängig von der Verfügbarkeit von Hardware und unabhängig von Testvektoren
    - **Anmerkungen:** Die Auswirkung von Interrupts auf Cache und Pipeline wird ignoriert, ebenso diverse Multicoreeffekte wie zum Beispiel Zugriffskonflikte am Memory-Interface.
      Indirekte Funktionsaufrufe und Schleifenobergrenzen können unter Umständen nicht aufgelöst und müssen manuell „annotiert“ (von Hand ergänzt) werden, was fehleranfällig ist.
- Codesimulation
    - **Use-Case:** Grobe Abschätzung der CET (core execution time) für das gegebene Testszenario
    - **Anmerkung:** Spielt in der Timinganalyse keine große Rolle.
- Messen
    - **Use-Ceses:** Analyse des realen Systems (die Software läuft auf der Zielhardware) zwecks Profiling, Verifikation oder Überwachung.
    - **Anmerkungen:** Unter (Timing-) Profiling versteht man die Ermittlung von Timingparametern wie der CPU-Auslastung, der CET (core execution time), der RT (response time) etc.
      Die Ergebnisse hängen von den Testvektoren ab, die die Software während der Messung bearbeitet hat.

- Tracing
    - **Use-Ceses:** Analyse des realen Systems (die Software läuft auf der Zielhardware) zwecks Visualisierung, Debugging, Optimierung, Profiling oder Verifikation.
    - **Anmerkungen:** Unter Tracing versteht man die Aufzeichnung von Ereignissen zur späteren Analyse und Visualisierung. Mit Blick auf das Timing eignen sich Schedulingtraces mit Ereignissen wie „Aktivierung“, „Start“, „Unterbrechung“, „Terminierung“ von Tasks besonders gut.
      Man unterscheidet Hardware-basiertes Tracing und Software-basiertes Tracing. Ersteres kann ohne Modifikation der Software auskommen, für letzteres wird die Software instrumentiert, was die Verwendung der serienidentischen Hardware im Fahrzeug erlaubt.
- Schedulingsimulation
    - **Use-Case:** Design und Optimierung von Schedulingkonzepten und der Betriebssystemkonfiguration; Analyse des typischen Schedulingverhaltens
- Statische Schedulinganalyse
    - **Use-Case:** Design und Optimierung von Schedulingkonzepten und der Betriebssystemkonfiguration; Analyse des worst-case Schedulingverhaltens (zum Beispiel Ermittlung der WCRT (worst-case response time)

**Laufzeitoptimierung**
Eine komplette Liste aller Laufzeitoptimierungsansätze zu erstellen ist unmöglich. Daher sollen im Folgenden lediglich ein paar grundsätzliche Aspekte behandelt und exemplarisch einige konkrete Maßnahmen vorgestellt werden.

Als allgemein gültige Regel lässt sich festhalten, dass Laufzeitoptimierung immer „top-down“ erfolgen sollte, also von den oberen zu den unteren Ebenen hin. Würde man direkt mit der Codeoptimierung beginnen, wüsste man nicht, ob der betreffende Code überhaupt zu einem Zeitpunkt ausgeführt wird, an dem das Timing kritisch ist. Besser ist es, zunächst auf der Schedulingebene ein *gutes Verständnis* der aktuellen Situation herbeizuführen, Optimierungen auf dieser Ebene durchzuführen und anschließend mittels Codeoptimierung die verbleibenden „Hot-spots“ anzugehen. Die Bedeutung des *guten Verständnisses* wird leider oft unterschätzt. In vielen Projekten konnte ich beobachten, wie Integratoren beim Blick auf den ersten heruntergeladenen Trace mit einer gewissen Fassungslosigkeit feststellen mussten, dass sich das System ganz anders verhält, als gedacht und zuvor simuliert.

**Laufzeitoptimierung auf der RTOS (Scheduling-) Ebene**
Die oberste Regel für Laufzeitoptimierung auf der Schedulingebene wird leider oft missachtet, obwohl sie denkbar einfach ist: „*Keep it simple!*“
Konkret heißt das beispielsweise, dass die Betriebssystemkonfiguration möglichst einfach gehalten wird. Am besten sollte BCC1 (Basic Conformance Class ohne Mehrfachaktivierungen) zum Einsatz kommen. Leider legen die meisten AUTOSAR RTE Generatoren die Verwendung von ECC nahe, indem sie eine nicht terminierende ECC Task anlegen und darin eine zweite Ebene des Schedulings einführen: die RTE Runnables werden in dieser ECC Task mittels Events getriggert. Die damit verbundene Komplexität wird von den meisten Projektverantwortlichen

nicht mehr überklickt und die Analyse von Timingproblemen wird deutlich erschwert.
Ganz nebenbei lässt sich mit der Verwendung von BCC der Stackbedarf signifikant verringern.

Eine weitere Maßnahme, um typische Echtzeitprobleme zu vermeiden, den Laufzeitbedarf und gleichzeitig den Stackbedarf weiter zu minimieren ist die Verwendung von kooperativem Multitasking. „Kooperativ“ heißt hier, dass Taskwechsel nur zu bestimmten Zeitpunkten erfolgen dürfen – sinnigerweise dann, wenn gerade kein Runnable läuft. Die RTE oder vergleichbare Mechanismen stellen dann fest, dass zur Sicherung der Datenkonsistenz keine Kopien der Daten notwendig sind. Ergebnis: Einsparung bei RAM und Laufzeit. Es leuchtet ein, dass darüber hinaus der Stackbedarf typischerweise drastisch reduziert wird, da eine Verschachtelung von Runnables nicht mehr möglich ist.

Abbildung 4 zeigt den Trace eines sehr positiven Beispiels für ein gelungenes Timingdesign: die Aktivlenkung des BMW X5 (e70). Das System ist – je nach Zustand – bis über 93% ausgelastet aber niemals überlastet. Durch die Optimierungsmaßnahmen konnte ein günstigerer, weniger leistungsfähiger Prozessor als in der Vorgängergeneration zum Einsatz kommen – und das bei zusätzlicher Funktionalität.

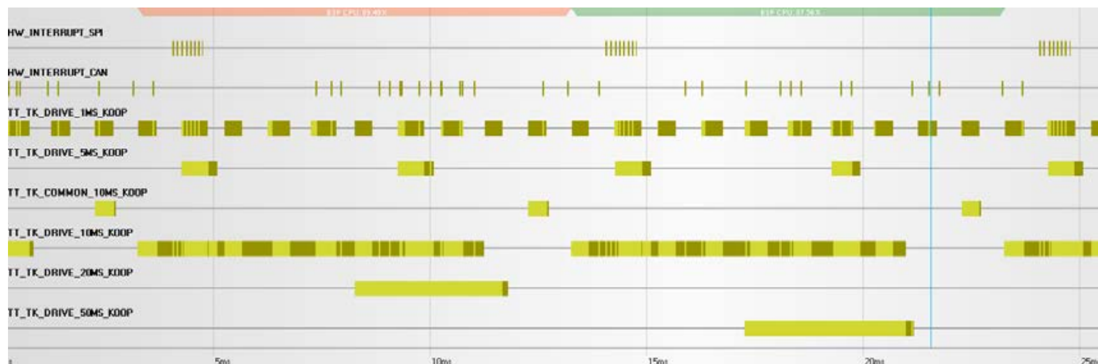


Abb. 4: Effizient, hoch ausgelastet, zuverlässig und sicher: Aktivlenkung des BMW X5

**Laufzeitoptimierung auf der Codeebene**

Als Beispiel für die Laufzeitoptimierung auf der Codeebene soll an dieser Stelle die wohlbekannte Funktion `memcpy` optimiert werden. memcpy kopiert eine definierte Anzahl von Bytes von einem Speicherbereich in einen anderen. Eine Standardimplementierung ist in Abbildung 5 ersichtlich.

```
/*---------- The ‘standard’ memcpy routine ------------------------
*   Parameters:
*      *pDest   - The destination to which data is copied across to
*      *pSrc    - The source of the data to be copied across. The
*                 addresses of pSrc and pDest are passed as arguments.
*                 This avoids having to pass the complete arrays in as
*                 arguments in order to do manipulations. Note, they
```

```
*                are void pointers to allow any type of array to be
*                passed.
*       nBytes - The number of bytes to copy from pSrc to pDest
*                Remember that a 'char' is 1 byte and an 'int' is 4
*                bytes (or a word)
*-------------------------------------------------------------------*/
void *memcpy_(void *pDest, void const *pSrc, unsigned short nBytes)
{
    /* Assign pSrc and pDest to 'char' Auto-variable pointers on
       the stack. This allows byte per byte transfer */
    char *pD = pDest;
    char const *pS = pSrc;
    /* Iterate through the number of bytes to copy across,
       decrementing nBytes until it reaches zero */
    while( nBytes-- )
    {
      /* Copy one byte from the source to the destination and then
         increment the index */
      *pD++ = *pS++; /* E.g. pD[i++] = pS[i++]; */
    }
    return pDest;
}
```

Abb. 5: Standardimplementierung der memcpy Funktion

Wird diese Funktion zum Kopieren von einem Kibibyte (1024 Bytes) auf einem Infineon AURIX TC275 mit 200 MHz Taktfrequenz und unter Verwendung eines TASKING Compilers verwendet, so ergibt sich für die Default Memory Locations eine minimale CET (core execution time) von 114,395µs oder **111,7ns pro Byte**. Dies ist der Ausgangspunkt für die nun folgenden Optimierungen. Der generierte Assemblercode ist in Abbildung 6 zu sehen.

```
80006e6e:  40 42  memcpy_:  mov.aa %a2,%a4
80006e70:  a0 0f            mov.a %a15,0
80006e72:  01 f2 10 40      add.a %a4,%a2,%a15
80006e76:  01 f5 10 30      add.a %a3,%a5,%a15
80006e7a:  9f 04 03 80      jned %d4,0,80006e80 <memcpy_+0x12>
80006e7e:  00 90            ret
80006e80:  79 3f 00 00      ld.b %d15,[%a3]0
80006e84:  2c 40            st.b [%a4]0,%d15
80006e86:  b0 1f            add.a %a15,1
80006e88:  3c f5            j 80006e72 <memcpy_+0x4>
```

Abb. 6: Der generierte memcpy Assemblercode des Ausgangspunktes der Optimierung

Im ersten Schritt sollen nun andere Speicherorte verwendet werden. Abbildung 7 zeigt einen Auszug aus dem AURIX Handbuch, dass die Zugriffsdauer auf verschiedene Speicher in Taktzyklen angibt. Ein entscheidender Multicoreaspekt sei an dieser Stelle schon mal erwähnt: die in der Tabelle angegebenen Zahlen gelten für den Fall, dass am Memory-Interface kein Zugriffskonflikt auftritt, also nicht etwas ein anderer Kern gerade mit höherer Priorität auf den Speicherbereich zugreift. Im Falle eines Konfliktes kann die Verzögerung wesentlich größer sein.

Infineon

**TC27x C-Step**

**On-Chip System Buses and Bus Bridges**

**Table 3-16 CPU access latency in CPU clock cycles for TC27x**

| CPU Access Mode | CPU clock cycles |
|---|---|
| Data read access to own DSPR | 0 |
| Data write access to own DSPR | 0 |
| Data read access to own or other PSPR | 5 |
| Data write access to own or other PSPR | 0 |
| Data read access to other DSPR | 5 |
| Data write access to other DSPR | 0 |
| Instruction fetch from own PSPR | 0 |
| Instruction fetch from other PSPR (critical word) | 5 |
| Instruction fetch from other PSPR (any remaining words) | 0 |
| Instruction fetch from other DSPR (critical word) | 5 |
| Instruction fetch from other DSPR (any remaining words) | 0 |
| Initial Pflash Access (critical word) | 5 + configured PFlash Wait States[1)] |
| Initial Pflash Access (remaining words) | 0 |
| PMU PFlash Buffer Hit (critical word) | 4 |
| PMU PFlash Buffer Hit (remaining words) | 0 |
| Initial Dflash Access | 5 + configured DFlash Wait States[2)] |
| TC1.6E/P Data read from System Peripheral Bus (SPB) | 4 ($f_{CPU}=f_{SPB}$)<br>7 ($f_{CPU}=2*f_{SPB}$) |
| TC1.6E/P Data write to System Peripheral Bus (SPB) | 0 |

1) FCON.WSPFLASH + FCON.WSECPF (see PMU chapter for the detailed description of these parameters).

2) FCON.WSDFLASH + FCON.WSECDF (see PMU chapter for the detailed description of these parameters).

Abb. 7: Zugriffszeiten auf verschiedene Speicher des Infineon AURIX TC275

Doch auch ohne Konflikte ist der Unterschied zwischen „schnellen“ und „langsamen“ Speichern erheblich. Die Default Memory Locations des Ausgangsstands waren: Cached Flash0 für den Code, LMU RAM für das Ziel der Kopie und Cached Flash0 für die Quelle.

Wird das Ziel nun statt im LMU RAM im Local DSPR0 abgelegt, verringert sich die Kopiergeschwindigkeit auf **100,6ns pro Byte**.

Im nächsten Schritt wird dem Compiler per #pragma oder per Compileroption –t0 vorgegeben, bei der Kompilierung möglichst schnellen Code zu erzeugen. Der Compiler generiert nun anderen Assemblercode, siehe Abbildung 8, unter der Verwendung von prinzipiell erstrebenswerten „post-increment“ Speicherzugriffen und unter der Verwendung von speziellen Befehlen des AURIX Befehlssatzes, in diesem Fall der loop Anweisung.

```
8020011c 40 4f    memcpy_:  mov.aa      a15,a4
8020011e 8e 46              jlez        d4,0x8020012a
80200120 60 42              mov.a       a2,d4
80200122 b0 f2              add.a       a2,#-0x1
80200124 04 5f              ld.bu       d15,[a5+]0x1
80200126 24 ff              st.b        [a15+]0x1,d15
80200128 fc 2e              loop        a2,0x80200124
8020012a 40 42              mov.aa      a2,a4
8020012c 00 90              ret
```

Abbildung 8: Der generierte memcpy Assemblercode mit Compileroptimierung

Der so erzeugte Code läuft wesentlich schneller und benötigt nur noch **59,6ns pro Byte**.

Abschließend wird der Code von Hand optimiert. In den meisten Fällen ist dies auf der C Code Ebene möglich, wobei permanent a) der erzeugte Assemblercode überprüft werden und b) mittels Messung die tatsächliche Laufzeit ermittelt werden muss. Im Bereich der Codeoptimierung ist die Verlockung groß, sich Annahmen und Vermutungen hinzugeben. Selbst die Besten der Besten erleben Überraschungen und müssen sich eingestehen, dass das erwartete Laufzeitverhalten eines Optimierungsansatzes weit entfernt liegt vom realen Verhalten und nur die Messung am realen System dies offenbart.

Vor der eigentlichen manuellen Optimierung ein paar Überlegungen zu memcpy. Laut Spezifikation kann sie Daten mit der Granularität von einem Byte kopieren. Der AURIX kann als 32bit Prozessor sehr effizient mit vier Byte großen Wörtern umgehen. Analysiert man die Datenobjekte einer typischen Anwendung im Automobilbereich, stellt man fest, dass die Größe der meisten Datenobjekte einem ganzzahligen Vielfachen von vier entspricht und sie auch vier Byte aligned sind – also eine Speicheradresse aufweisen, die ebenfalls ein ganzzahliges Vielfaches von vier ist. Die in Abbildung 9 dargestellte Implementierung der entscheidenden Bereiche von memcpy macht sich diese Erkenntnisse zu Nutzen und überprüft, ob Quelle und Ziel vier Byte aligned sind und ob auch die Anzahl der zu kopierenden Bytes ein ganzzahliges Vielfaches von vier ist. Ist dies der Fall, werden immer vier Bytes in einem einzelnen Schleifendurchlauf kopiert.

```
/* Divide nBytes by 4 get rid of EXTR.U and get word decrements. */
uint32_t wordCount = nBytes >> 2u;
/* Check for word alignment. */
if( 0u == (( (uint32_t)pDest | (uint32_t)pSrc | nBytes ) & 3u )) {
    /* Assign Word Pointers */
    uint32_t *pD = (uint32_t *)pDest;
    uint32_t const *pS = (uint32_t const *)pSrc;
    while( 0u != wordCount-- ) {
        *pD++ = *pS++; /* Copy words (4 bytes at a time) */
    }
} else {
        .... /* else copy byte by byte as previously */
}
```

Abbildung 9: Von Hand optimierte Version von `memcpy` (Auszug)

Diese manuelle Optimierung ergibt zusammen mit allen zuvor genannten Optimierungen einen Laufzeitbedarf von **14,7ns pro Byte**. Immerhin eine Laufzeitersparnis von rund **87%** gegenüber der Ausgangsversion.

**Multicore-spezifische Aspekte**
Für Multicoresysteme hat es sich bewährt, die Funktionalität so aufzuteilen, dass intensive Berechnungen und die Bearbeitung vieler Interrupts auf verschiedene Kerne aufgeteilt werden. Als Folge werden bei den komplexen Berechnungen Pipeline und Cache besser genutzt, was den Durchsatz erhöht.

„Busy-Spinning" also das Warten eines Kerns auf die Freigabe einer Ressource auf einem anderen Kern sollte wann immer möglich konzeptionell vermieden werden. In vielen Fällen lässt sich der Einsatz von Spin-locks durch geeignetes Design der Software komplett vermeiden. Bei der Portierung von Singlecore Software auf Multicore ist das blinde Suchen/Ersetzen von DisableInterrupts()/EnableInterrupts() mit GetSpinlock()/ ReleaseSpinlock() eine schlechte Idee. Der beste Mechanismus zum Schutz von Daten ist der, den man nicht braucht.

**Zusammenfassung, Ausblick**
Laufzeitoptimierung ist komplex und vielschichtig. Sie kann und sollte auf verschiedenen Ebenen stattfinden: auf der Schedulingebene und auf der Codeebene und zwar in dieser Reihenfolge, um zu verhindern, dass (Codeoptimierungs-) Aufwand betrieben wird, der die Gesamtsituation gar nicht verbessert.

Voraussetzung für eine zielgerichtete Analyse und Optimierung ist die Kenntnis der verschiedenen Analysetechniken. Nur so kann man für den jeweiligen Anwendungsfall das richtige Werkzeug bemühen.

Der wichtigste erste Schritt, eine existierende Anwendung auf dem Weg zu einer effizienten und sicheren Software zu bringen, ist das Verständnis darüber, wie sich das System tatsächlich verhält. Vermutungen und Annahmen sind hier fehl am Platz; der Einblick und die Analyse des realen Systems unter realen Bedingungen (also auch im Fahrzeug) sind entscheidend.

Seit einigen Jahren machen sogenannte C-to-C Compiler von sich reden, die gegebenen C Code in parallelisierbaren C Code übersetzen sollen. Ich bin diesem Ansatz gegenüber skeptisch. Meine Vermutung (oder soll ich sagen: „Hoffnung"?) geht eher in die Richtung, dass eine bessere Ausnutzung von Multicore dadurch erzielt werden wird, dass die Codegeneratoren grundsätzlich anders arbeiten werden. Zukünftige Generatoren werden es erlauben, aus einem generischen Modell („generisch" im Sinne von nicht auf Singlecore oder Multicore festgelegt) entweder Code für Singlecore (bzw. Single-thread) oder aber gut parallelisierbaren Code für Multicore zu erzeugen.

**Literatur- und Quellenverzeichnis**
[1] Timingposter, Peter Gliwa, Februar 2013, https://www.gliwa.com/downloads/Timing\%20Poster.pdf

**Autor**
Seit der Gründung 2003 steht Peter Gliwa als geschäftsführender Gesellschafter an der Spitze von GLIWA. Über viele Jahre hinweg entwickelte er die Timingsuite T1 und berät heute international Kunden in Timingfragen und bei Themen rund um Echtzeitbetriebssysteme.
Zuvor war er bei ETAS zunächst Entwickler, danach Produktmanager des Echtzeitbetriebssystems ERCOS$^{EK}$. Zwischen 2001 und 2006 war er darüber hinaus als Dozent für das Fach „Mikrocomputertechnik" tätig.
Peter Gliwa hat Elektrotechnik an der Berufsakademie Stuttgart studiert.

**Kontakt**
Internet: https://gliwa.com
Email: peter.gliwa@gliwa.com