Verifysoft TECHNOLOGY

ESE-Kongress 2014
Testing Code Coverage

Wenn es mal etwas weniger sein darf

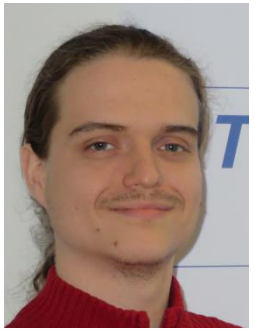
Roland Bär

Sebastian Götzinger

Überblick

1. Einleitung
2. Code Coverage
3. Instrumentation
4. Herausforderung kleiner Targets
5. Bitcov und Bytecov
6. Fazit und Ausblick

1. Wer sind wir?



Sebastian Götzinger

- Studiert seit 2011 Angewandte Informatik
- Seit 2013 tätig für die Firma Verifysoft
- Trainer für Testtools

Roland Bär

- Seit 1994 Softwareentwickler
- Maßgeblich beteiligt an Tools der Firmen Testwell und Parasoft
- Seit 2003 CTO bei der Firma Verifysoft Technology GmbH

1. Wo kommen wir her?

Verifysoft Technology GmbH

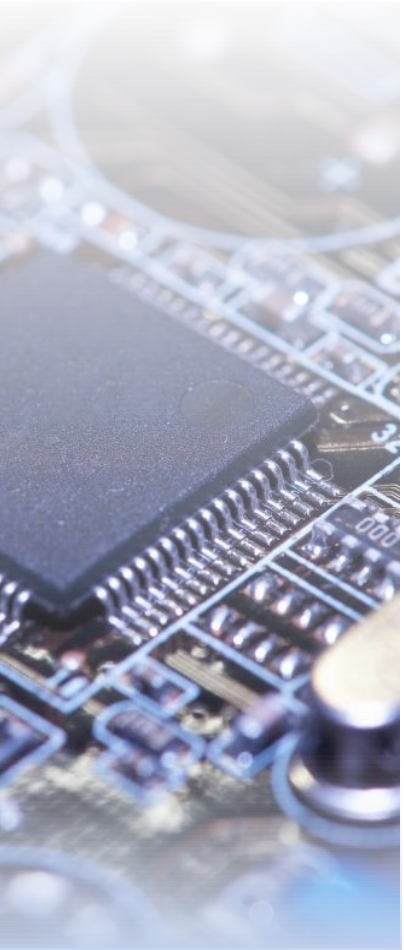
- Gegründet 2003
- Hauptsitz: Technologie Park Offenburg
- >400 Kunden in mehr als 20 Ländern (Stand 2014)
- Eigentümer der Tools
  - Testwell CTC++ (Code Coverage)
  - Testwell CMT++ (SW-Metriken)
  - Testwell CTA++ (Unit-Tests)
- Vertrieb komplementärer Software (z. B. Grammatec Codesonar®)
- Seminare zu Softwarequalität

1. Warum sind wir hier?

- Testing auf kleinen Targets ist speziell
- Welche Probleme gibt es?
- Welche Lösungen passen dazu?
- Compiler verhalten sich anders als erwartet
- Die Frage: Entspricht die Theorie der Praxis?

2. Code Coverage

verifysoft TECHNOLOGY



Was ist Code Coverage?

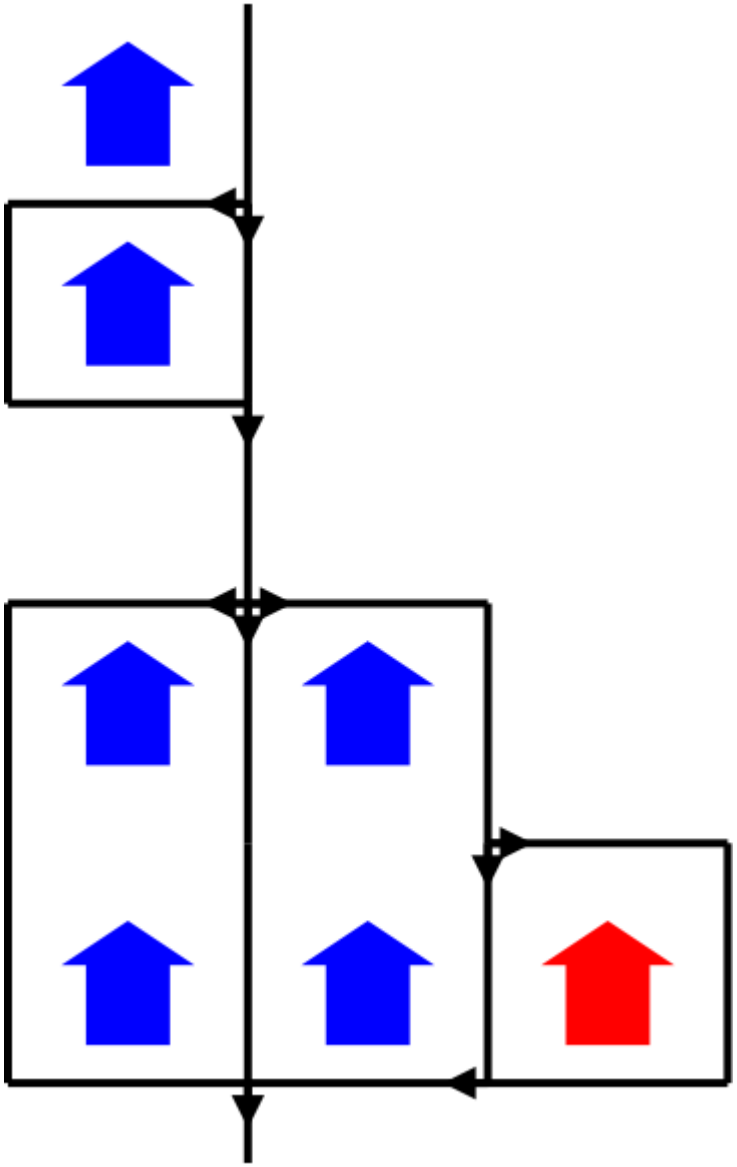
Der Coverage-Report ist der einzige Nachweis dass und was Sie überhaupt getestet haben.

Wir benötigen Testpunkte.

[1]

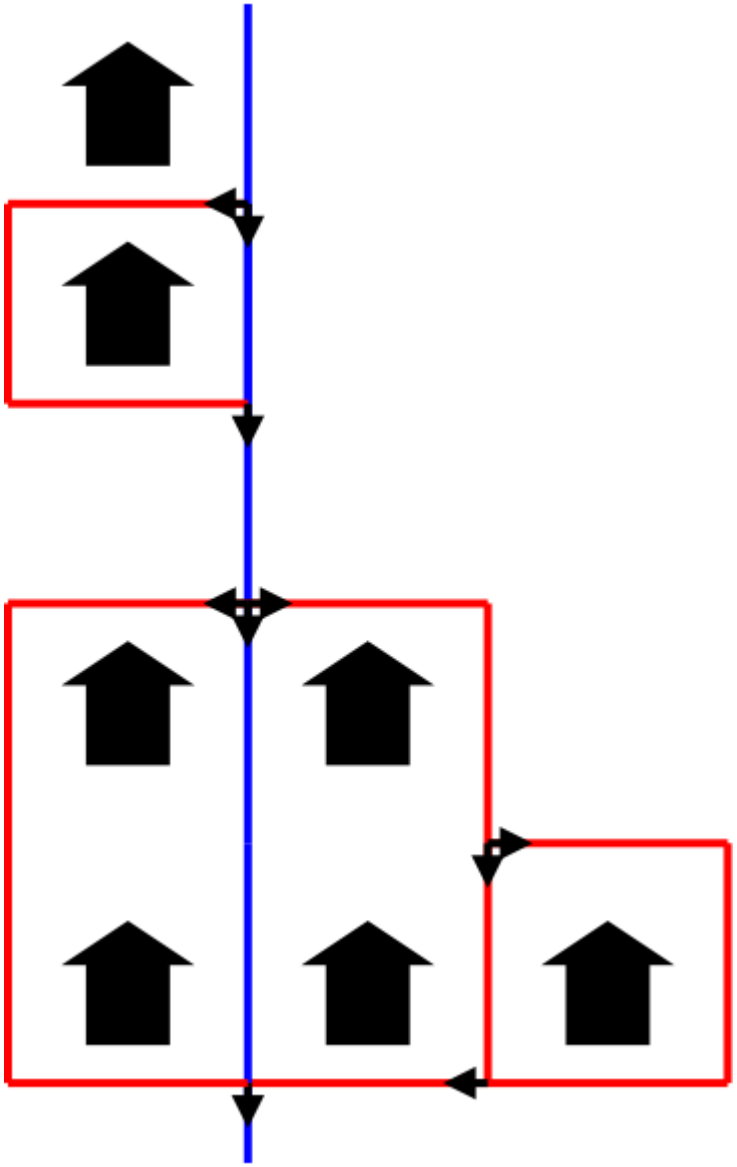
Quelle: [1]http://de.wikipedia.org/wiki/Geschwindigkeits%C3%BCberwachung#mediaviewer/File:Einseitensensor_ES3.jpg (Stand 2014)

2. Statement/Line Coverage



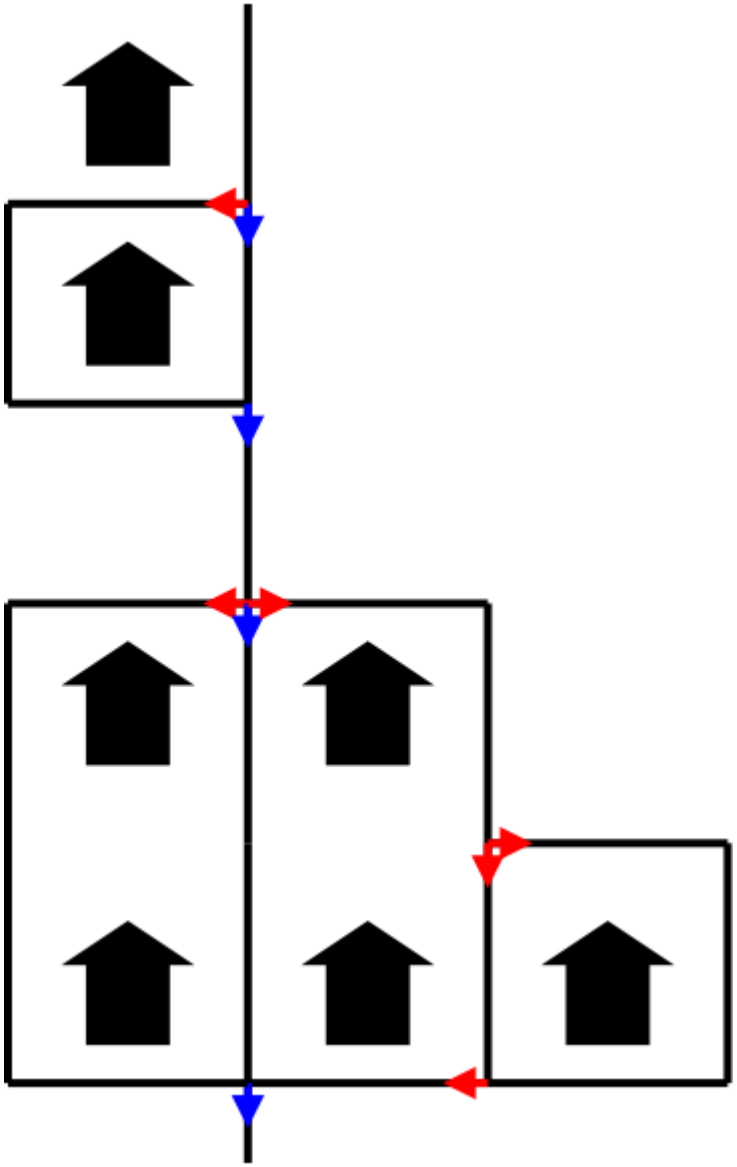
- Prüft das Durchlaufen aller Anweisungen/Zeilen
- Ausreichend nur für groben Überblick
- ‚Abfallprodukt' aus den Debuginfos, benötigt diese aber auch
- In der Praxis selten aussagekräftig
- Eher ungeeignet für sicherheitskritsche Anwendungen
- Normen fordern meist höherwertige Coverage

2. Branch Coverage



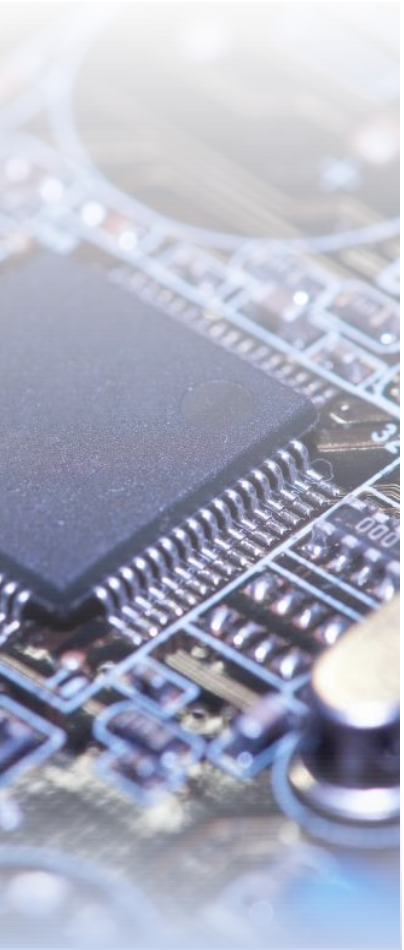
- Umfassender
- Auch ‚leere' else-Pfade werden geprüft
- oft ausreichend, wenn bestimmte Kriterien erfüllt sind
- Jedoch oft unfähig bei verschachtelten Bedingungen versteckte Fehler zu entdecken
- Nach ISO 26262-6 bis ASIL Level D stark empfohlen

2. Condition/Decision Coverage



- Prüft erschöpfend alle relevanten Bedingungen
- Prüft somit auch sonst möglicherweise nicht beachtete Seitenfälle
- Sichert auch Statement und Branchcoverage zu
- Nicht nur das Ergebnis, auch die Faktoren der Entscheidung
- Grundlage der von vielen Normen geforderten MC/DC-Coverage

3. Techniken der Instrumentierung



Ermittlung von Code Coverage

- Pseudo-Instrumentierung
- Manuelle-Instrumentierung/Debugger
- Hardware-Instrumentierung
- Objektcode-Instrumentierung
- Quellcode-Instrumentierung

„Jede Codezeile ist schuldig, bis ihre Unschuld bewiesen ist.“ – Anonymous[2]

Quelle: http://softwaretestingfundamentals.com/software-testing-quotes/
(Stand 2014)

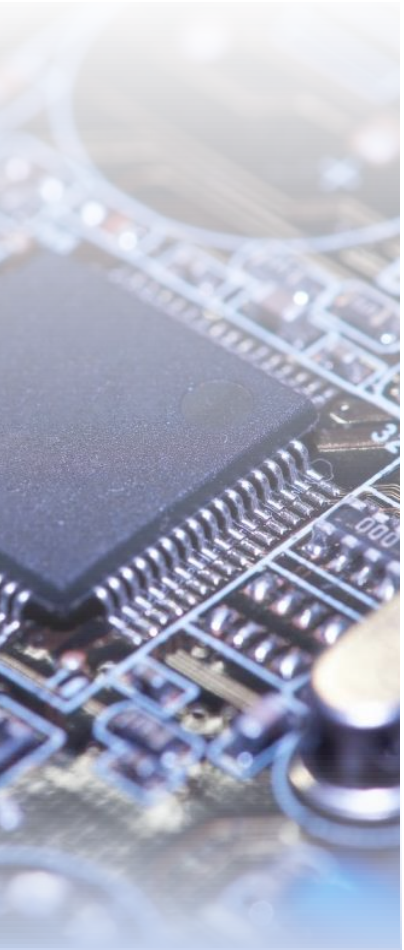
3. Pseudo-Instrumentierung

- Argumentativer Ansatz:
  ‚Bei geeigneter Auswahl von Unittests wurde die Coverage erreicht!‘
- Test werden mittels Tools erstellt.
- Aber:
  Schwere Nachweisbarkeit der Coverage!
- Nach DO178-C verboten!

3. Via Debugger

- Wenig Overhead
- Nur Linecoverage praktikabel
- Zeile für Zeile manuell 'abhaken'
- Braucht ein Debuggerinterface

3. Bus-Monitor



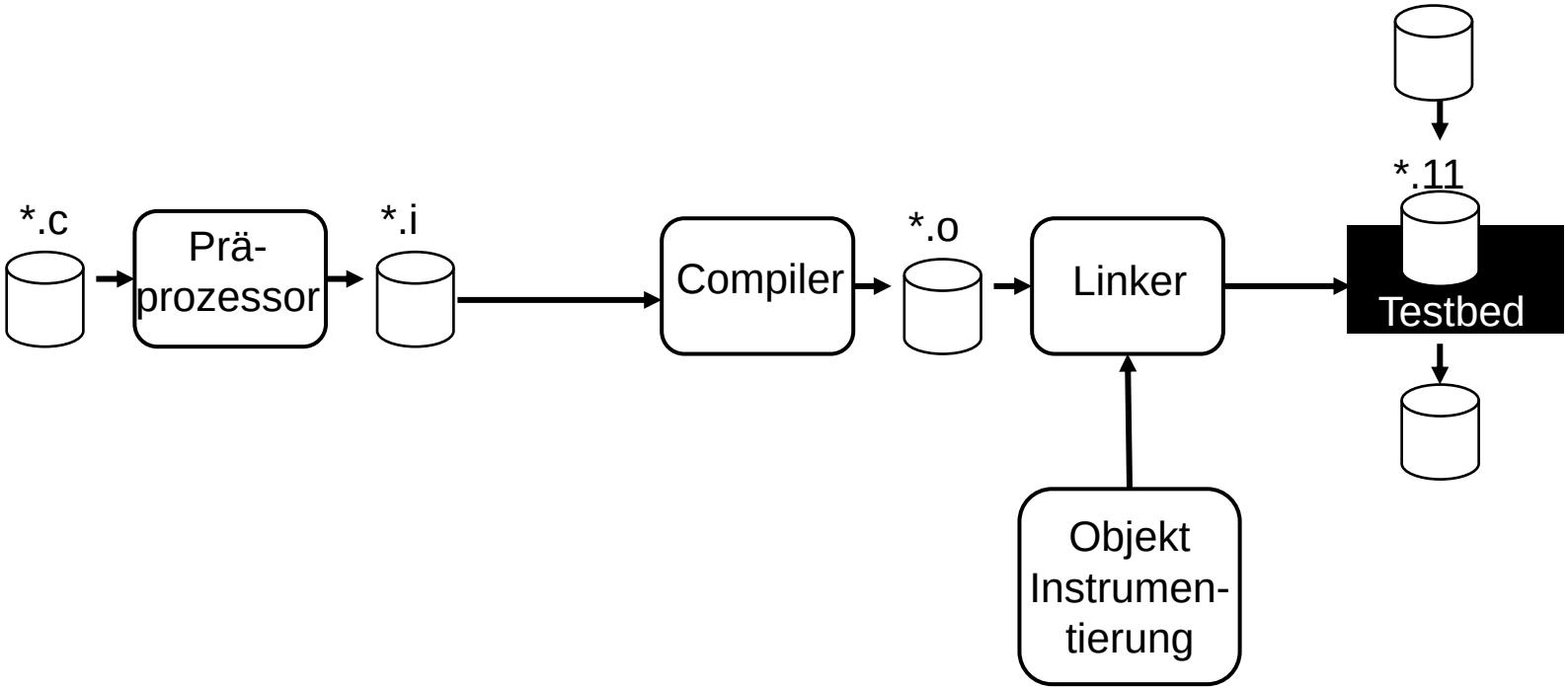
Mittels eines passendes Gerätes können alle Aktivitäten des Bus überwacht werden.

In den seltensten Fällen ökonomisch, da alleine die Überwachung (ohne 'Buchführung') möglich ist Kostenpunkt ~20000€/System

3. Objektcode-Instrumentierung

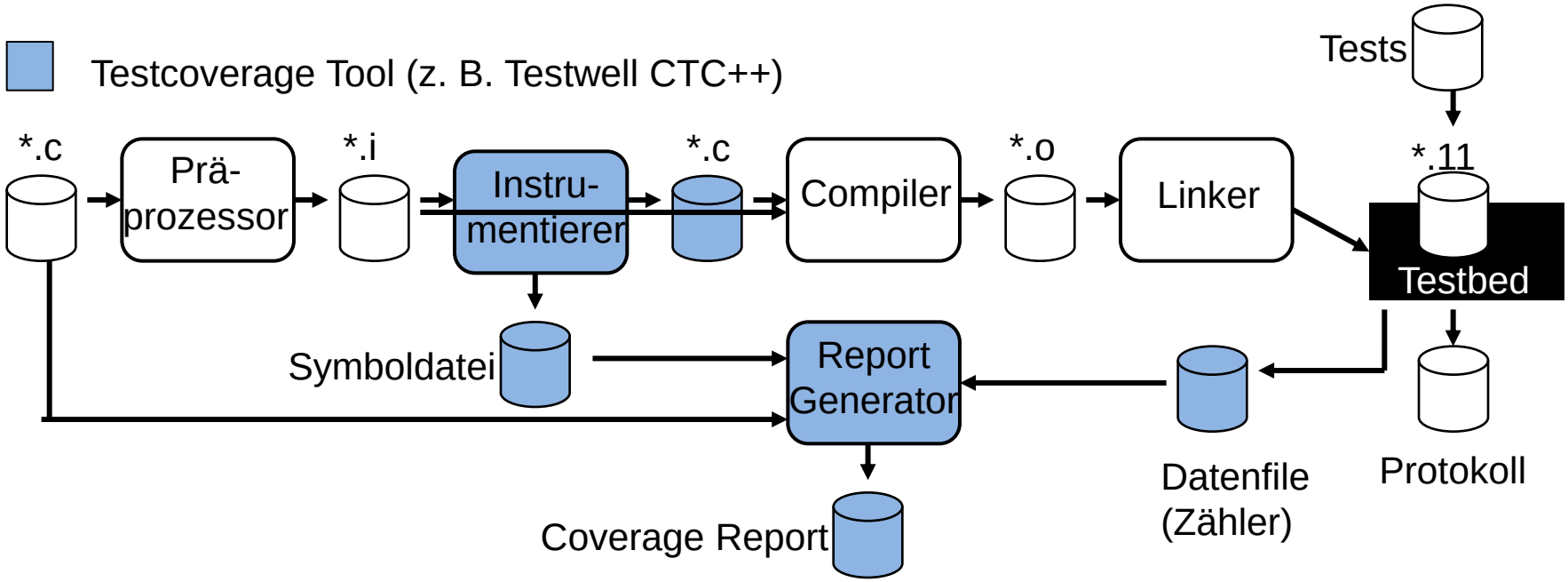
verifysoft TECHNOLOGY

Beispiel: Rational PurifyPlus



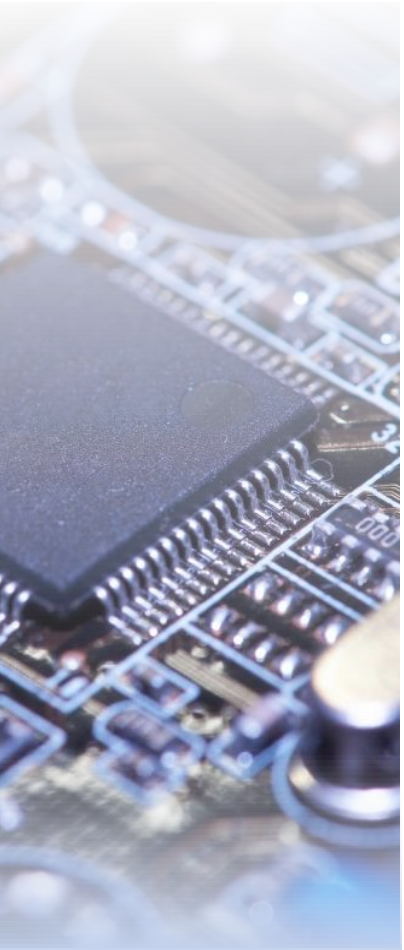
Quelle: Prof. Dr. Daniel Fischer, Seminar Effiziente Unit Tests für Embedded Systems, modifiziert

3. Sourcecode-Instrumentierung



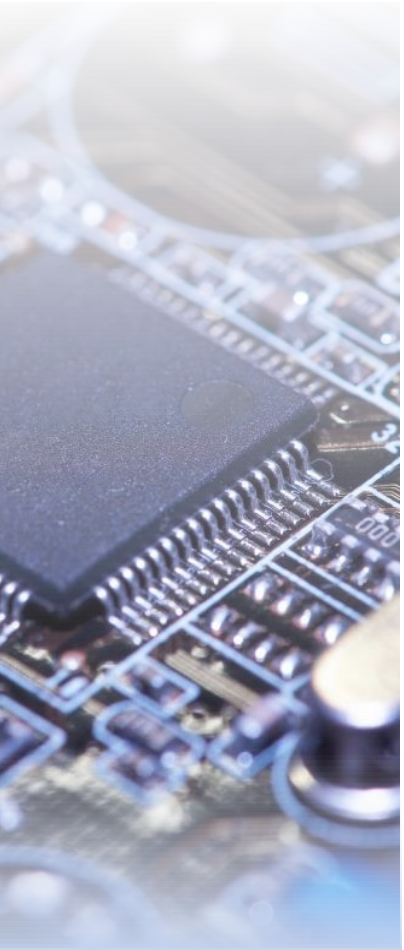
Quelle: Prof. Dr. Daniel Fischer, Seminar Effiziente Unit Tests für Embedded Systems (Stand 2014)

4. Herausforderungen auf kleinen Targets



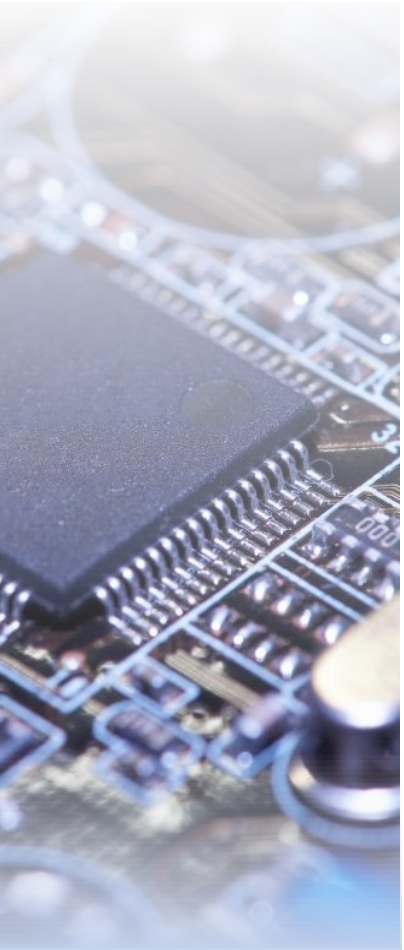
- Fehlendes Dateisystem
- Sicherung von Messdaten
- Extrahieren der Messdaten
- Begrenzter Speicher
- Keine großen APIs

4. Was kostet die Instrumentierung?



- ROM aufgrund von Overhead
- ROM aufgrund des Zählens
- RAM aufgrund der Datensammlung
- Nerven aufgrund falscher Tools

4. RAM und ROM



ROM

Wie und wo wird gespeichert?

Wie wird das Format aussehen?

Aggregation von Hits?

Setzen von Bits?

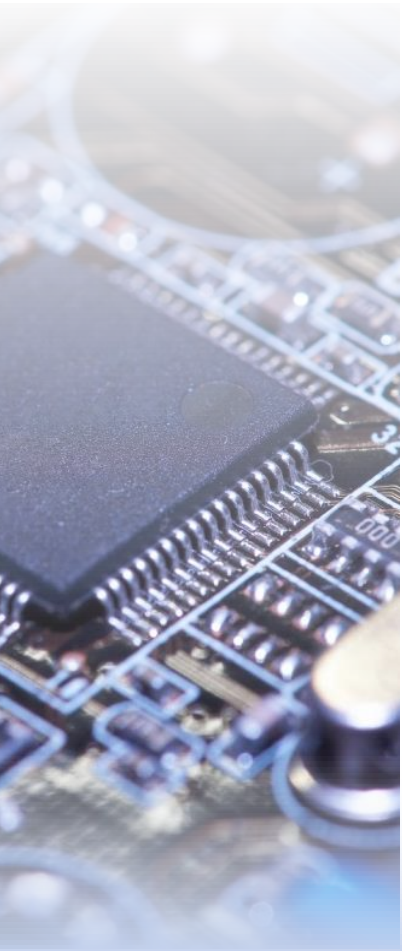
RAM

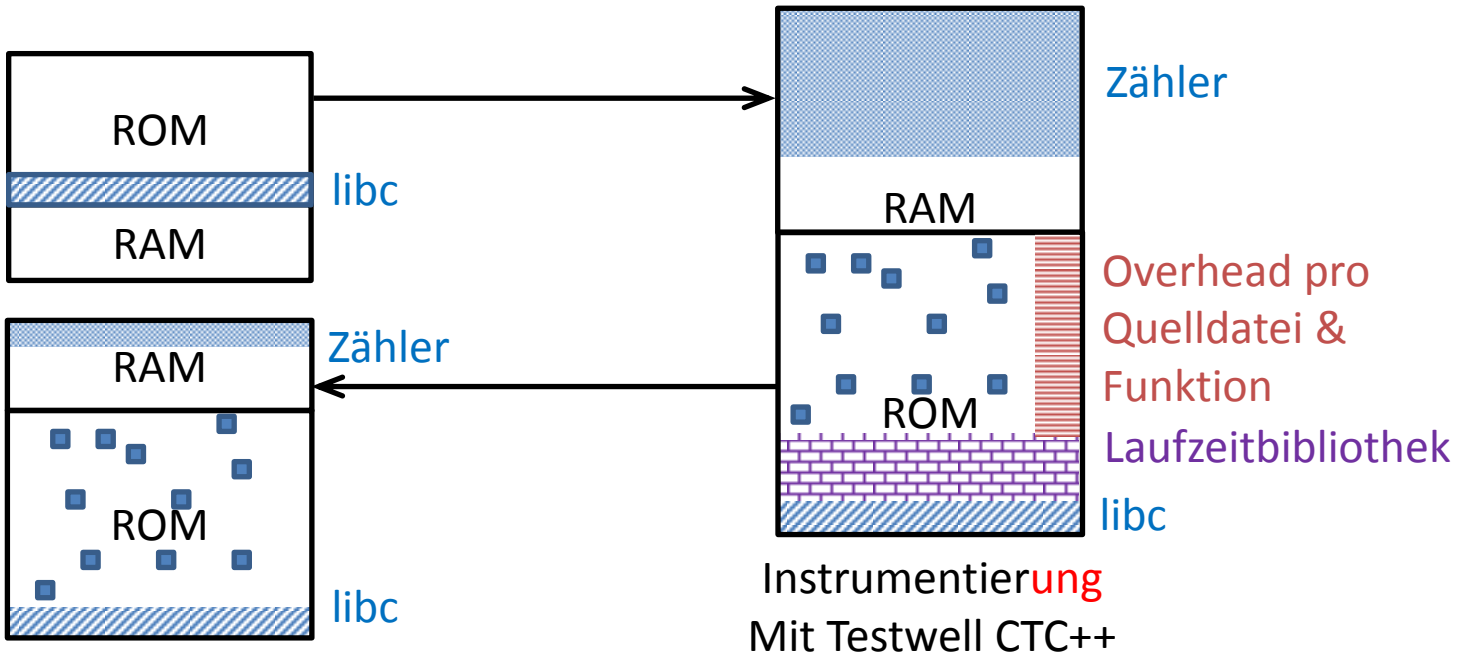
Zähler

Die Folge:

Das Instrumentierte Programm passt einfach nichtmehr in das Target!

Illustration des Speicherverbrauchs

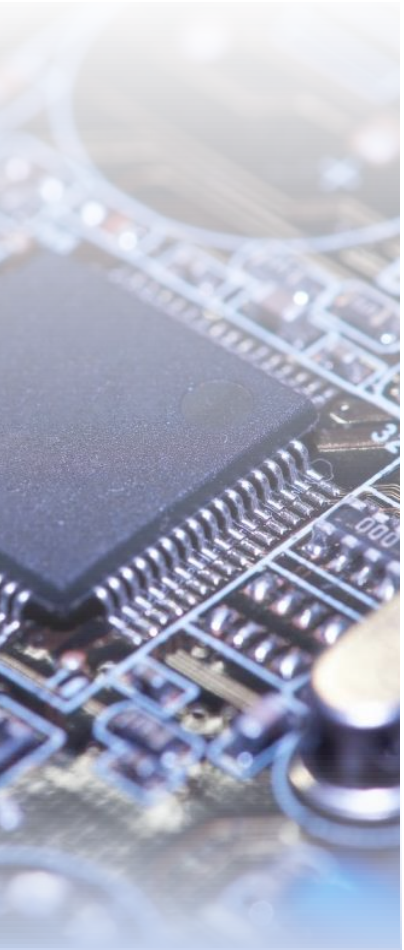




Instrumentierung
Mit Testwell CTC++
Bitcov-Verfahren

Instrumentierung
Mit Testwell CTC++
Host-Target-Verfahren

4. Instrumentierung auf dem Target



Warum ist Instrumentierung auf dem Target nötig?

- Seiteneffekte können Testabläufe beeinflussen.
- Normen verlangen diese Tests nicht grundlos!
- Auch deren Coverage muss aussagekräftig belegt werden können!
- Tools helfen hierbei Zeit, Nerven und Geld zu sparen

5. Bytecov/Bitcov

Wichtige Fragen:

- Was ist nötig?
- Warum ist es nötig?
- Auf was kann man verzichten?
- Wo liegen die Schwerpunkte?

Ansatz 1:
Aufteilung des Codes

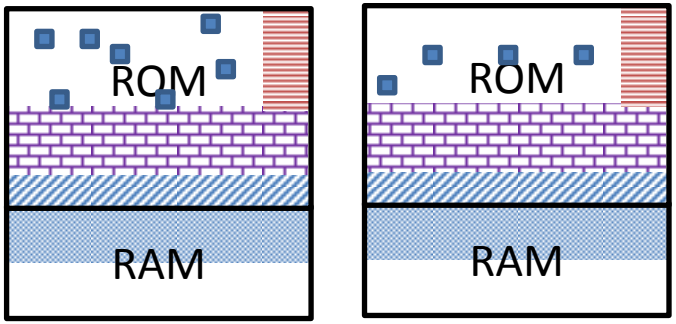
- Es ist nicht notwendig, immer das Gesamtsystem zu testen
- Häufig reicht es aus, in mehreren Läufen Fragmente des Systems zu testen.

✓ Kein anderes Tool notwendig

☠ Aufwendig

☠ Fragmente müssen aggregiert werden

☠ Fehleranfällig

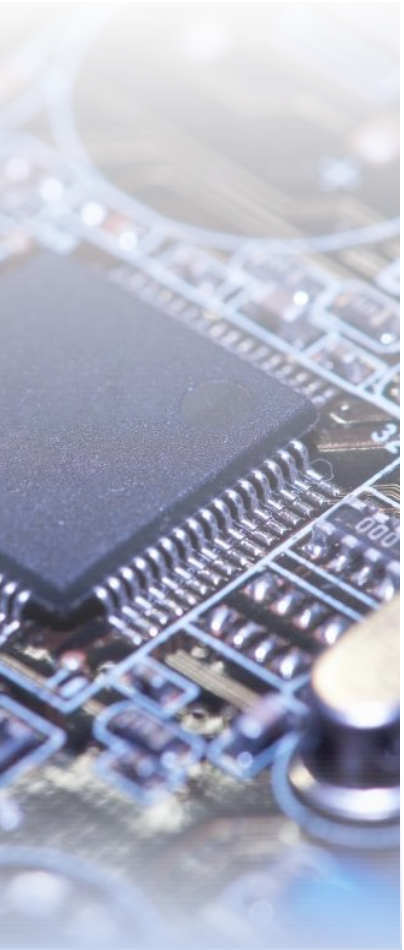


Overhead pro Quelldatei & Funktion

Laufzeitbibliothek libc

Zähler

Ansatz 2:
Auslagerung der Laufzeitbibliothek



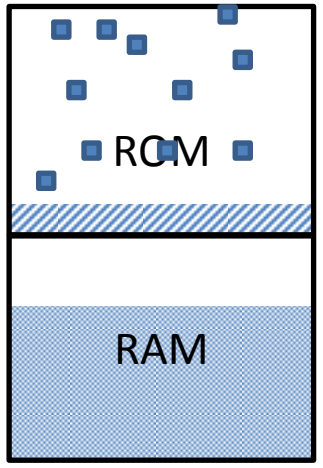
- Oft nutzen Targets eine Laufzeitbibliothek für die Instrumentierung
- Oft notwendig um auf Dateisystem oder serielle Schnittstellen zuzugreifen
- Dadurch verursachter Overhead vermeidbar (in der Praxis 4-11kB)

✓ Spart viel Platz

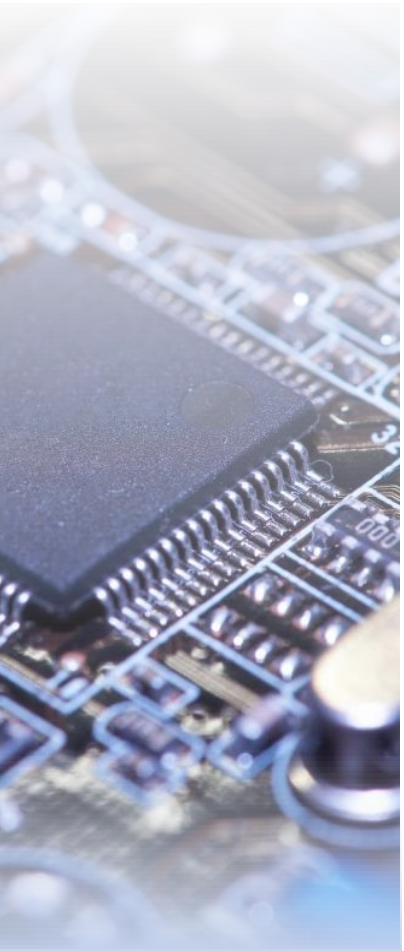
✓ Wirkt sich nicht auf die Instrumentierungsqualität aus

☠ Daten müssen extrahiert werden

☠ Daten müssen nach dem Durchlauf verarbeitet werden.



Ansatz 3:
Optimierung der Zählart



- Im Normalfall ist es irrelevant, wie oft ein Knoten besucht wurde
- Wichtig ist, ob er besucht wurde

⇨ Setzen eines Bits

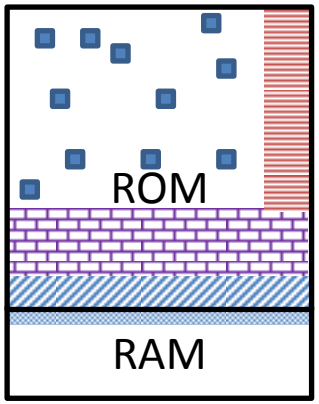
✓ Spart viel Platz hinsichtlich des RAMs

✓ Überläufe führen zu keinen false negatives mehr

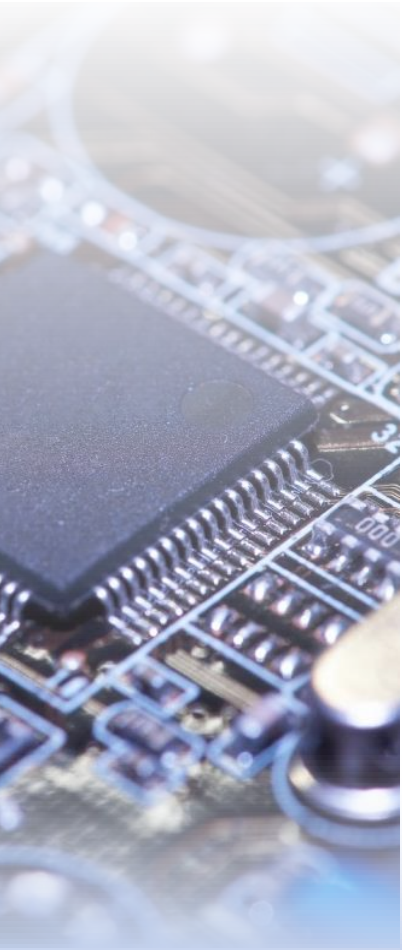
✓ Wirkt sich marginal auf die Instrumentierungsqualität aus

☠ Kleiner Anstieg der ROM-Belegung durch zusätzliche Logik

☠ Daten müssen nach dem Durchlauf verarbeitet werden



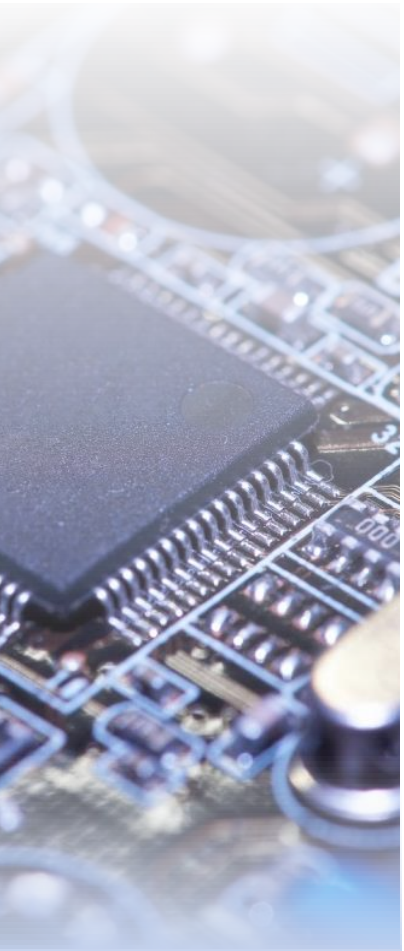
5. Bitcov



- ✓ Erfolgreich in der Minimierung von RAM-Verbrauch
- ✓ Verzichtet auf eine Laufzeitbibliothek
- ✓ Setzt Bits, anstatt Zähler zu inkrementieren
- ✓ Weitere Optimierung durch Ausnutzung chipspezifischer Funktionen

Aber: Löst das RAM, nicht das ROM-Problem

5. Bytecov

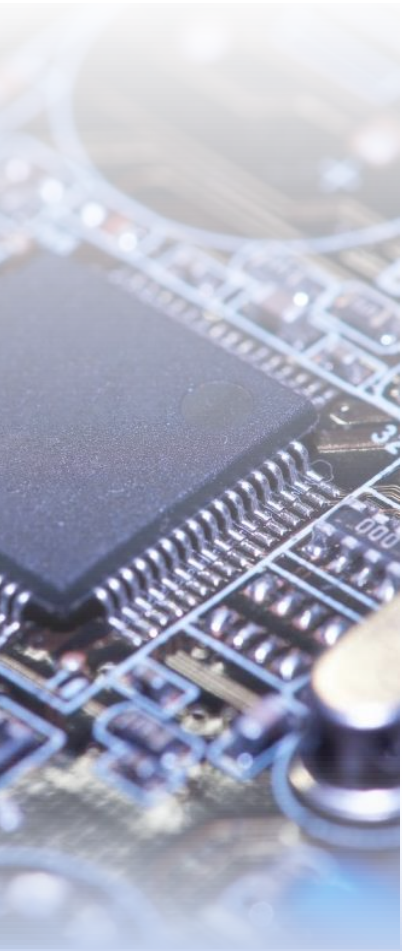


- ✓ Anstatt einem Bit wird ein Byte gesetzt
- ✓ Verminderung des ROM-Verbrauchs
- ✓ Anstieg des RAM-Verbrauchs
- ✓ Nötig, weil viele Chips keine Bits direkt addressieren und dies via Code gelöst werden muss

Variante: Bytecov++

- ✓ Inkrementiert das Byte anstatt es zu setzen
- ✓ Bei manchen Chips und Compilern weniger ROM-Verbrauch

6. Die Ergebnisse



```
/* BLINKY.C - LED Flasher for the Keil MCBx51 Evaluation Board with 80C51 device

#include <reg51f.h>

// When you have enabled the option Stop Program Execution with Serial
// Interrupt, the Monitor-51 uses the serial interrupt of the UART.
// It is therefore required to reserve the memory locations for the interrupt
// vector.  You can do this by adding one of the following code lines:

// char code reserve [3] _at_ 0x23;   // when using on-chip UART for communication
// char code reserve [3] _at_  0x3;   // when using off-chip UART for communication

void wait (void) reentrant
{                   /* wait function */
  ;                                   /* only to delay for LED flashes */
}

void main (void)  {
  unsigned int i;                     /* Delay var */
  unsigned char j;                    /* LED var */

  while (1) {                         /* Loop forever */
    for (j=0x01; j< 0x80; j<<=1)  {   /* Blink LED 0, 1, 2, 3, 4, 5, 6 */
      P1 = j;                         /* Output to LED Port */
      for (i = 0; i < 10000; i++)  {  /* Delay for 10000 Counts */
       wait ();                       /* call wait function */
      }
    }

    for (j=0x80; j> 0x01; j>>=1)  {   /* Blink LED 6, 5, 4, 3, 2, 1 */
      P1 = j;                         /* Output to LED Port */
      for (i = 0; i < 10000; i++)  {  /* Delay for 10000 Counts */
       wait ();                       /* call wait function */
      }
    }
  }
}
```

6. ARMGCC

verifysoft TECHNOLOGY

Opti	Normal	internes Array			externes Array		
		Bitcov	Bytecov	Bytecov++	Bitcov	Bytecov	Bytecov++
-O0	964	560	380	392	560	396	408
-O1	612	324	232	204	328	288	260
-O2	516	272	184	172	280	236	224
-O3	520	244	172	172	252	232	224
-Os	516	272	184	172	280	236	224

Angaben in Byte

6. Linux GCC32

verifysoft TECHNOLOGY

Opti	Normal	internes Array			externes Array		
		Bitcov	Bytecov	Bytecov++	Bitcov	Bytecov	Bytecov++
-O0	818	606	372	465	643	382	444
-O1	631	307	261	262	292	261	255
-O2	773	411	359	443	467	351	435
-O3	773	411	359	467	467	351	435
-Os	570	275	247	235	284	247	247

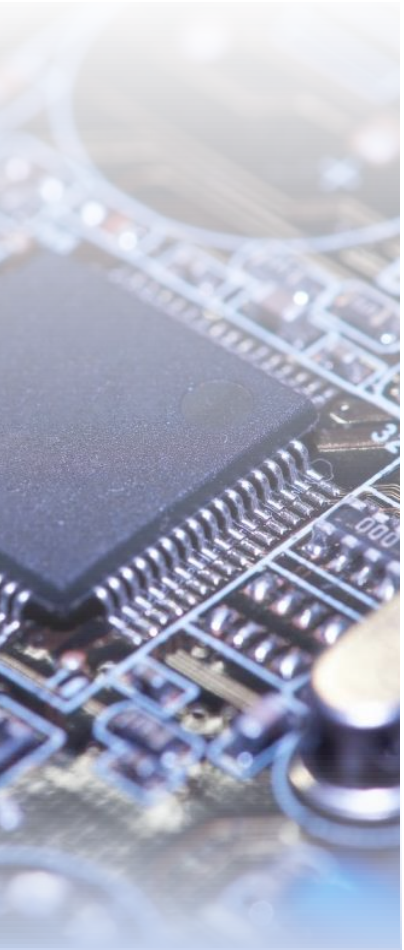
Angaben in Byte

6. ICCV850

Opti	internes Array		externes Array	
	Bitcov	Bytecov	Bitcov	Bytecov
-On	312	284	312	258
-Ol	306	278	306	252
-Om	218	216	222	192
-Oh	174	172	174	148
-Ohz	148	146	152	126

Angaben in Byte

6. Fazit



Optimierung durch den Compiler reduziert die Menge des Codes signifikant.

Weitere Reduzierung durch Bit/Byte-Coverage.

Danke für Ihre Aufmerksamkeit!

Verifysoft Technology GmbH
Technologiepark Offenburg
In der Spöck 10-12
77656 Offenburg
Deutschland
Tel.: +49 781 127 8118-0
Fax: +49 781 63 920-29
info@verifysoft.com