Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Analyse

Florian Schmaus, Simon Schuster

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
https://www4.cs.fau.de

11. Juni 2018

Überblick

Überblick

Annahmen

- C99
- x86 bzw. x86-64, d. h.
  - vorzeichenbehaftete Integer als Zweierkomplement implementiert
  - `char` hat 8 Bit
  - `short` hat 16 Bit
  - `int` hat 32 Bit
  - `long` hat 32 Bit auf x86 und 64 Bit auf x86-64

Frage 17

Angenommen `x` hat Typ `int`. Ist `x - 1 + 1` ...

1. definiert für alle Werte
2. definiert für manche Werte
3. definiert für keinen Wert

von x?

Erklärung

- additive Operatoren sind linksassoziativ

⇒ nicht definiert für `INT_MIN`

Frage 18

Angenommen `x` hat Typ `int`. Ist `(short)x + 1` ...

1. definiert für alle Werte
2. definiert für manche Werte
3. definiert für keinen Wert

von x?

Erklärung

- wenn x nicht in `short` passt

  ⇝ Verhalten ist implementierungsabhängig

Frage 19

Angenommen `x` hat Typ `int`. Ist `(short) (x + 1)` ...

1. definiert für alle Werte
2. definiert für manche Werte
3. definiert für keinen Wert

von x?

Erklärung

- wenn `x + 1` nicht in `short` passt

  ⇝ implementierungsabhängig
- die meisten Compiler schneiden beim Cast ab

Frage 20

Hat die Auswertung von `INT_MIN % -1` definiertes Verhalten?

1. ja
2. nein
3. das weiß niemand ...

Erklärung

- C99 macht dazu keine Aussage
- in C11 gilt folgendes:
  - wenn `(a/b)*b + a%b` darstellbar ist, haben `a/b` und `a%b` definiertes Verhalten
  - sonst nicht
  - `INT_MIN / -1` entspricht `INT_MAX + 1`
  - was auf x86/x86-64 nicht darstellbar ist

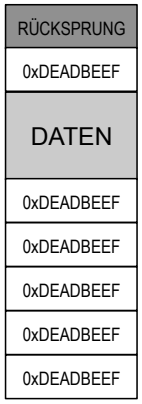
Überblick

Analyse Harter Echtzeitsysteme

- Harte, verlässliche Echtzeitsysteme
  - Garantien über Ressourcenbedarf notwendig
    ☞ statische Analyse unabdingbar
- Mögliche Ressourcen: Speicherbedarf, Laufzeit, etc.
- Übung: Analyse des Stackverbrauchs einer Feldbibliothek
- Stack-Analyse
  1. Dynamisch: Wasserstandstechnik
  2. Statisch: „Eigenbau“ und aiT (Stack-Analyzer der a^3 Suite)
- WCET-Analyse mittels aiT (bereits in EZS behandelt)

Dynamische Analyse des Stapelspeicherverbrauchs

- Messung zur Laufzeit: Water-Marking (siehe Vorlesung)
- Grundidee: Einfügen von **Stack Canaries**
- Explizite Verwaltung des Stapelspeichers notwendig
- pthread-Bibliothek ermöglicht Verwaltung
- Mögliche Canaries
  - Lesbare Bitmuster: `0xDEADBEEF`
  - Unwahrscheinliche Bitmuster: `0b101010101010...`



pthread-Bibliothek

1. (Globalen) Stack anlegen:

```
1  static unsigned int g_data[DATA_SIZE];
```

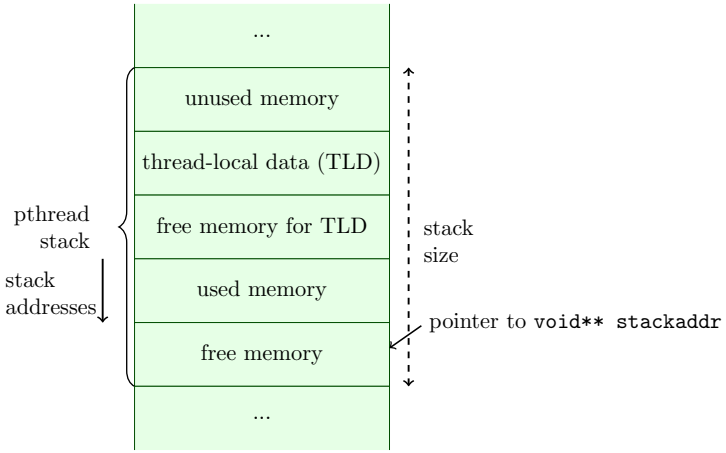
2. Thread anlegen & starten:

```
1  pthread_t thread;
2  pthread_attr_t attr;
3  pthread_attr_init(&attr);
4  pthread_attr_setstack(&attr, &g_stack, STACK_SIZE);
5  // worker function: void *run(void *param)
6  int status = pthread_create(&thread, &attr, run, NULL);
7  if (status != 0) { ... // handle error
```

3. Auf Thread warten:

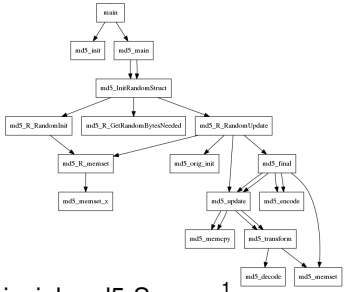
```
1  pthread_join(thread, &ret);
```

pthread Stack



Statische Worst-Case Stack Usage Analyse

Vorgehensweise



```
1  /* Objective function */
2  max: +16 md5_orig_init +64 md5_update
       +64 md5_final\
3    +16 md5_memset +208 md5_transform
       +16 md5_encode ...;
4
5
6  /* Constraints */
7  +main = 1;
8  +md5_init +md5_main <= +main;
9  ...
10
```

- Beispiel: md5-Summe[1]
- Vorgehen
  1. Callgraph bestimmen
  2. Stackverbrauch einzelner Funktionen (`gcc -fstack-usage`)
  3. ILP[2] aufstellen (Nebenbedingungen aus 1., Kosten aus 2. verwenden)
  4. ILP z.B. mittels `lp_solve` ⇝ **worst-case Stackverbrauch**

[1] `https://github.com/tacle/tacle-bench/`

[2] Integer Linear Program (dt. ganzzahliges lineares Programm)

Optimierungsziel

- Jeder Stapelrahmen einer Funktion *f* hat eine Größe *size*
- Jede Funktion kann auf einem Pfad ein- oder mehrfach (Rekursion), insgesamt *n*-fach auf dem Stapel vorkommen
- Gesucht ist Pfad durch den Aufrufgraphen, welcher Stapelbedarf maximiert
- Dabei müssen **Flussbedingungen** eingehalten werden
  - Aufruferbeziehung
  - Alternativen
  - ...

Optimierungsziel

$$max \sum_{\text{Funktion } f} size_f \cdot n_f$$

In `lp_solve`-Syntax: `max: +64 n_f1 +48 n_f2 +42 n_f3`

Flussbedingung: Initialer Aufruf

Semantik

Der initiale Aufruf erfolgt maximal (wahlweise auch genau) ein mal

Formalisierung

$$n_{\texttt{main}} \leq 1$$

`lp_solve`-Syntax

```
n_main <= 1;
```

Flussbedingung: Mehrere Vorgänger

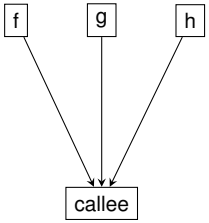
Semantik

Jede Funktion kann nur so oft ausgeführt werden, wie sie von den Vorgängern aus aufgerufen wird

Formalisierung

Sei $f_{a \to b}$ die Anzahl der Aufrufe von b durch a:

$$n_{callee} \leq \sum_{p \in \text{Aufrufer(callee)}} f_{p \to callee}$$



lp_solve-Syntax

```
n_caller <= +f_f_callee +f_g_callee +
            f_h_callee;
```


Flussbedingung: Immer nur ein Nachfolger pro Funktion

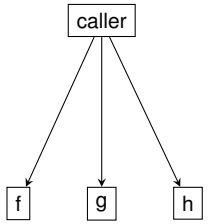
Semantik

Jede Funktionsinkarnation ruft jeweils maximal eine weitere Funktion auf

Formalisierung

Sei $f_{a \to b}$ die Anzahl der Aufrufe von b durch a:

$$\sum_{c \in \text{Aufgerufene(caller)}} f_{caller \to c} \leq n_{\text{caller}}$$



lp_solve-Syntax

```
+f_caller_f +f_caller_g +f_caller_h <=
            n_caller;
```

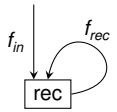

Flussbedingung: Rekursion

Semantik

Rekursive Funktionen können pro Aufruf von außen bis zu ihrer maximalen Rekursionstiefe (d) oft ausgeführt werden.

Formalisierung

$$f_{rec} \leq d_{rec} \cdot f_{in}$$
$$n_{rec} \leq f_{in} + f_{rec}$$



lp_solve-Syntax

```
f_rec <= +42 f_in;
n_rec <= f_in + f_rec;
```

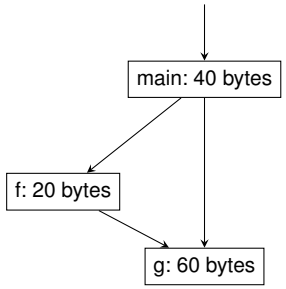

Beispiel

- Problemformulierung in lpsolve:

```
max: +40 n_main +20 n_f +60 n_g;

n_main <= 1;
+f_main_f +f_main_g <= n_main;
n_f <= +f_main_f;
+f_f_g <= n_f;
n_g <= +f_f_g +f_main_g;
```

- Ausgabe von lp_solve

```
Value of objective function:
    120.00000000

Actual values of the variables:
n_main                          1
n_f                             1
n_g                             1
f_main_f                        1
f_main_g                        0
f_f_g                           1
```



LP-Solve Fallstricke: Infeasible model

Infeasible Models

Eine oder mehrere der Nebenbedingungen führen zu einem logischen Widerspruch

Leider bietet `lp_solve` selbst direkt keine Hilfestellung zur Lokalisation. Die Entwickler empfehlen das Einführen von "slack"-Variablen:[3]

```
max: x + y;
x + 1 <= x;
y > y + 1;
x <= 20;
y <= 20;
```

```
max: x + y;
x + 1 - e_1 <= x;
y + e_2 > y + 1;
x <= 20;
y <= 20;
```

```
x:    20
y:    20
e_1:   1
e_2:   1
```

[3] http://lpsolve.sourceforge.net/5.5/Infeasible.htm

LP-Solve Fallstricke: Unbounded model

Unbounded Models

Eine oder mehrere der Variablen sind nach oben unbeschränkt

Durch künstliche Beschränkung aller Variablen im System (auf einen sehr großen Wert) lassen sich unbeschränkte Variablen detektieren:

```
max: x + y + z;
z <= y + 1;
y <= 20;
```

```
max: x + y + z;
z <= y + 1;
y <= 20;
x <= 5000;
y <= 5000;
z <= 5000;
```

```
x:  5000
y:    20
z:    21
```


LP-Solve Fallstricke: Syntax

- `lp_solve` ist auf die Lösung linearer Gleichungssysteme ausgelegt
- Es ist dementsprechend nicht möglich, zwei Variablen zu multiplizieren
  - `a * b` ⇒ Syntaxfehler
  - `max: a b` ⇒ optimiert $a + b$
- Lösung in VEZS für Konstanten (Stapelrahmengrößen): C-Präprozessor:

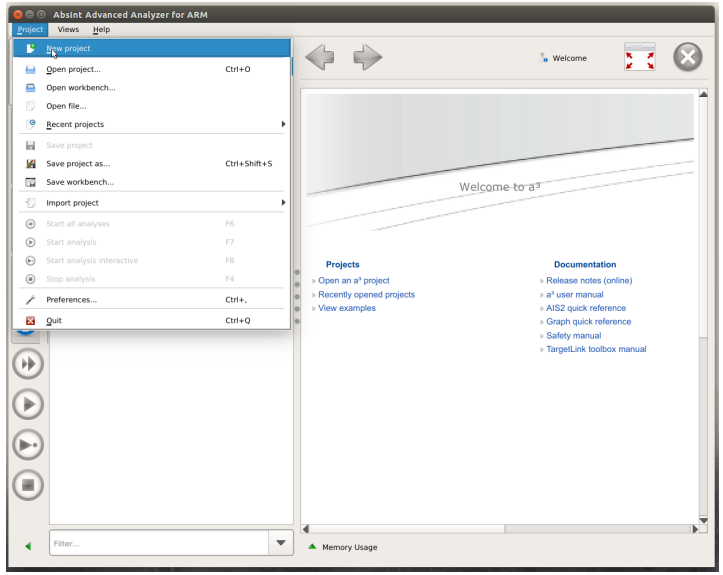
```
#define s_main 40
#define s_f    20
#define s_g    60

max: +s_main n_main +s_f n_f +s_g n_g;
```

⇝ `stackusage/lp_solvepp`

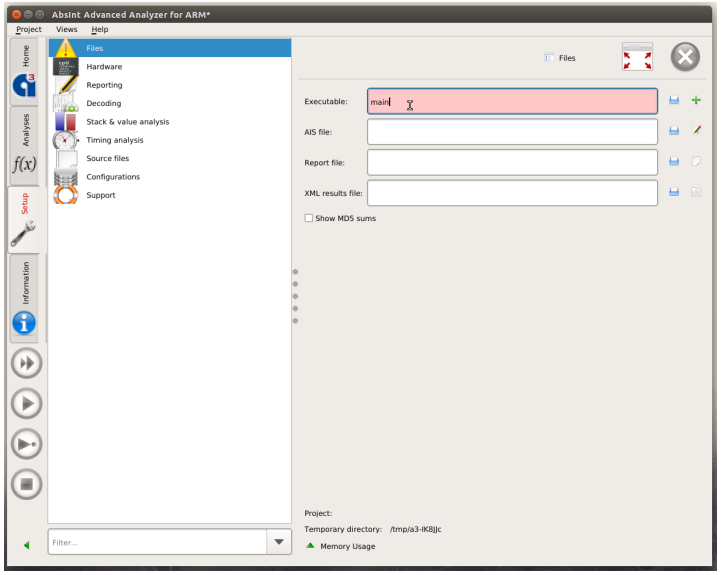
Statische Analyse des Stackverbrauchs

- Statische Code-Analyse mit a^3 Tool-Suite
  1. aiT: WCET-Analyse
  2. Stack-Analyzer: Stackverbrauch
  3. ...
- Installiert im CIP-Pool
- Verfügbare Rechner:
  - 00.153-113 CIP3: faui0da-u
  - 00.156-113 CIP4: faui0ca-q

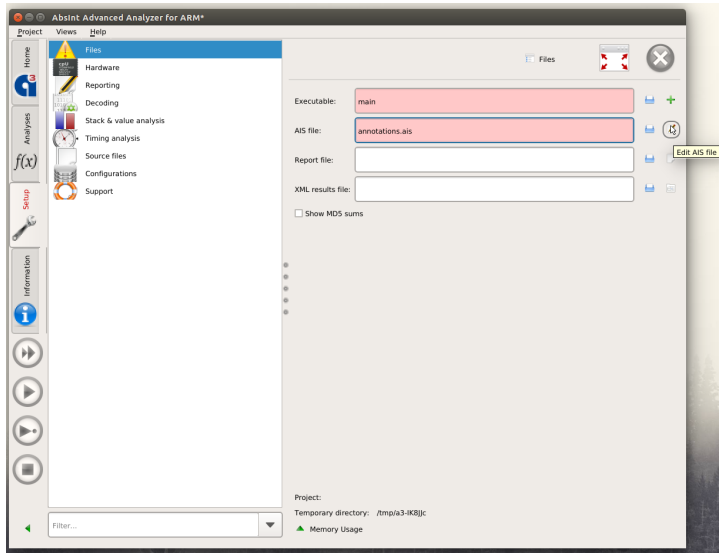
a^3 Analyzer – Neue Projekt Anlegen



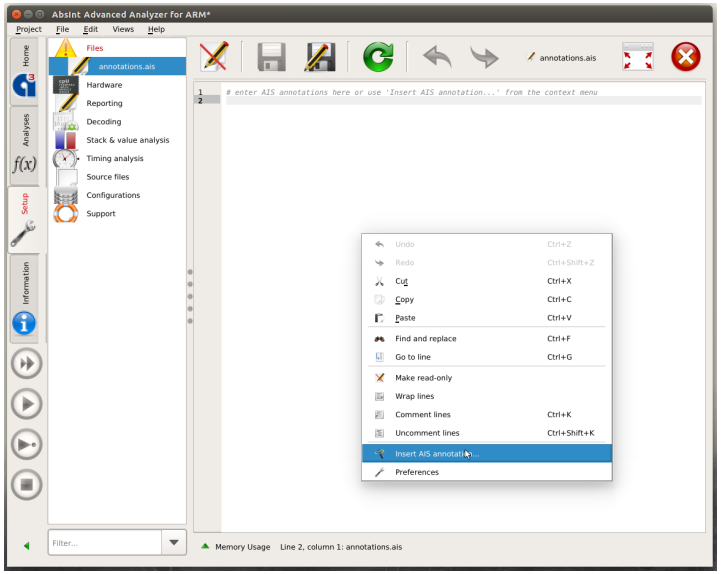
a^3 Analyzer – Executable Angeben



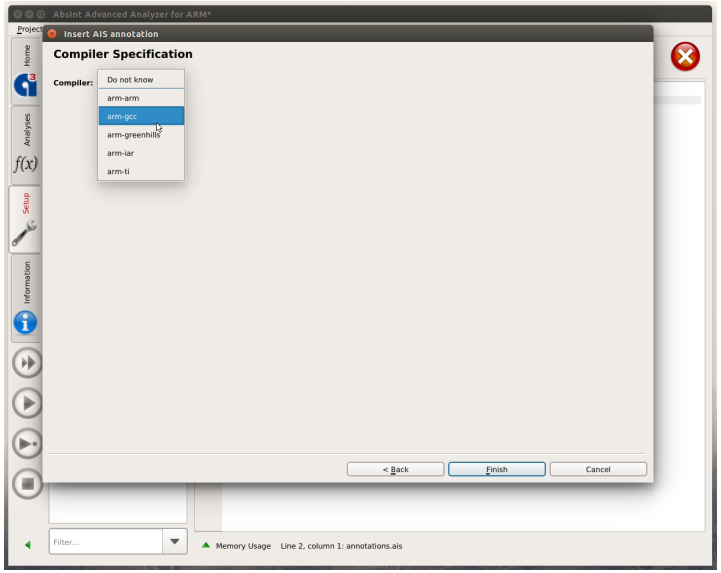
a^3 Analyzer – Annotations-Datei Anlegen



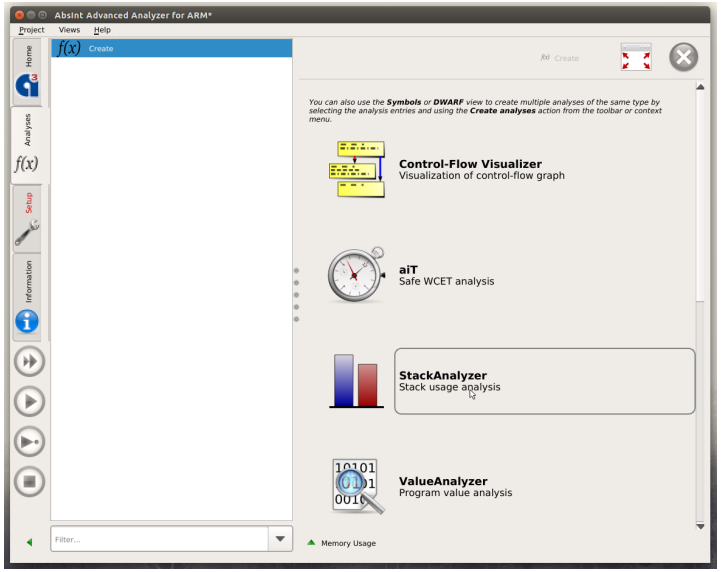
a^3 Analyzer – Compiler-Annotation Anlegen



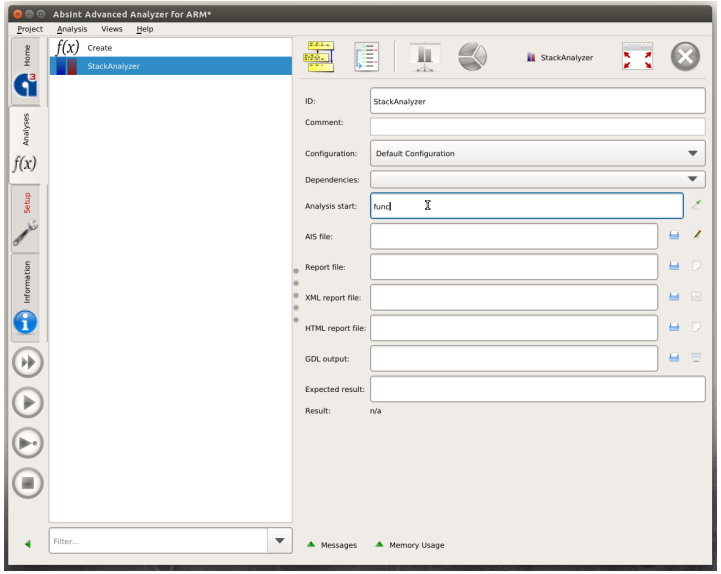
a^3 Analyzer – arm-gcc Angeben



a^3 Analyzer – Stack-Analyse Selektieren



a^3 Analyzer – Stack-Analyse Starten



Überblick

Aufgabenstellung

- Existierende Implementierung: Array-Datenstruktur
- Vorgegebene Funktionen: Sortieren, Maximumssuche, ...
- Aufgaben

1. Dynamische Analyse
   - 1.1 Thread erstellen
   - 1.2 Stack initialisieren
   - 1.3 Programm (mit Eingabedaten) ausführen
   - 1.4 Stackverbrauch messen
2. Statische Analyse
   - 2.1 ILP aus Aufrufgraph aufstellen
   - 2.2 Mittels `lp_solve` lösen
   - 2.3 Optional: Verwendung a^3 Stack-Analyzer

Fragen?