AUFGABE 6: STACKVERBRAUCHSANALSYSE

In dieser Aufgabe werden Sie lernen, wie Sie Eigenschaften wie Stackverbrauch sowohl dynamisch als auch statisch analysieren können. Zunächst werden Sie mit Hilfe der *Wasserstands*-Technik den Stapelspeicherverbrauch einer von uns gegebenen Bibliothek messen. Anschließend werden Sie eine einfache statische Analyse für eine Testanwendung durchführen um den maximalen Stackverbrauch auf eine sichere Art und Weise zu bestimmen.

Auf freiwilliger Basis können Sie sich zusätzlich mit den Werkzeugen StackAnalyzer und aiT der Firma AbsInt für Worst-Case-Stack-Consumption- und Wort-Case-Execution-Time(WCET)-Analyse vertraut machen.

Grundlegende Übung

Wasserstand

Aufgabe 1 *Vorgabe*

Machen Sie sich mit unserer Vorgabe in `src/arrays.c`, `include/arrays.h` und `src/stackanalyzer.c` vertraut. *Was wurde hier jeweils implementiert? Was leisten die Funktionen* `fill()`*,* `find_max()`*,* `array_cpy()` *und* `sort()` *sowie das Makro* `GET_TOS()`*? Wo liegt der Stapelspeicher für den Faden, der mit der Funktion* `run()` *gestartet wird?*
Antwort:

Wir haben versucht, Ihnen so viel Ärger wie möglich mit der `pthreads`-Bibliothek zu ersparen. Dennoch ist es eine gute Idee, sich mit den Handbuchseiten von `pthread_create` und `pthread_attr_setstack` auseinanderzusetzen.

☞ `man 3p pthread_create`

☞ `man 3p pthread_attr_setstack`

Aufgabe 2 *Messaufbau*

Implementieren Sie die noch fehlenden Komponenten für die Messung des Speicherverbrauchs gemäß der in der Vorlesung und Übung vorgestellten Methode. Hierzu gehören hauptsächlich Funktionen zum Initialisieren des Stapelspeichers und zur Auswertung des bisher maximal genutzten Stapelspeichers.

Aufgabe 3 *Messung*

Messen Sie den Stack-Verbrauch der von uns vorgegebenen `run`-Funktion und der transitiv aufgerufenen Funktionen mit den von uns vorgegebenen Daten durch eine konkrete Ausführung des Programms.
Antwort:

☞ `make stackanalyzer_x86`

☞ `./stackanalyzer_x86`

Überlegen Sie sich eigene Eingabewerte für die Datenstruktur, die den Speicherverbrauch maximieren. *Für welche Eingabesequenzen ergeben sich die höchste Speicherverbräuche für die einzelnen Funktionen?*

Antwort:

Statische Stackanalyse

In dieser Teilaufgabe soll nun der Stapelverbrauch einzelner Funktionen durch statische Analyse bestimmt werden. Dazu wird das Flussproblem des maximalen Stapelverbrauchs wie in den Übungen gezeigt in ein mathematisches Optimierungsproblem umgewandelt und durch den Löser `lpsolve` ermittelt. Analysieren Sie hier den maximalen Stapelverbrauch der `run()` Funktion.

Aufgabe 4 *Problemstellen*

Verschaffen Sie sich zunächst einen Überblick über das Analyseproblem *und identifizieren Sie dort etwaige Problemstellen für die Analyse*.
Antwort:

☞ `make stackgraph_arrays`

Wie können Sie diese Probleme ausräumen oder in Ihrer Analyse berücksichtigen? Notieren Sie sich die nötigen Anpassungen der Implementierung.
Antwort:

Aufgabe 5 *Analyse*

Bestimmen Sie nun den maximalen Stapelverbrauch. Kodieren Sie dazu die relevanten Flussbedingungen als Nebenbedingungen für ein numerisches Maximierungsproblem für `lpsolve`. Als Ausgangspunkt können Sie die Datei `arrays.skel.lp` nutzen, welche automatisch im `build`-Verzeichnis erzeugt wird. Der Stapelbedarf kann nun mittels `./stackusage/lp_solvepp arrays.lp` ermittelt werden. **Achtung:** Die Datei `build/arrays.skel.lp` wird durch erneute Aufrufe von `make stackgraph_arrays` überschrieben, kopieren Sie sich deshalb die Datei und halten Sie diese Kopie nach dieser Änderung manuell aktuell.
Antwort:

☞ `make stackgraph_arrays`

Vergleichen Sie insbesondere den analysierten Stackverbrauch mit dem von Ihnen gemessenen. *Wie verhalten sich die Ergebnisse mit Bezug zu Ihrer Messung? Wie lassen sich etwaige Abweichungen erklären? Verbessern Sie bei Bedarf die Implementierung Ihrer Messung.* Gegebenenfalls kann es sich als hilfreich erweisen, eine einfache Visualisierung der Verteilung des Prüfmusters auf dem Stapel zu implementieren.
Antwort:

☞ `GET_TOS`

Aufgabe 6 *Übersetzeroptimierungen*

Bestimmen und vergleichen Sie nun die Ergebnisse für unterschiedliche Compileroptimierungen. Gerade `-O0` *und* `-Os` *sollten hier interessant sein.*
Antwort:

☞ `src/CMakeLists.txt`

Erweiterte Übung

Warteschlange

Aufgabe 7

Kopieren Sie Ihre *Prioritätswarteschlangenimplementierung aus der vorherigen Aufgabe* in die dafür vorgesehenen Dateien. Ändern Sie ihre Implementierung dahingehend ab, dass sich Prioritäten mehrfach einfügen lassen. Nutzen Sie Ihre

☞ `queue.h`
☞ `queue.c`

Warteschlangenimplementierung aus der Datei `stackanalyzer_ext.c`, um die selben Daten wie in der grundlegenden Übung abzuspeichern. Iterieren Sie einmal über die ganze Warteschlange und suchen Sie auch hier nach dem größten Wert. Bestimmen Sie den Stapelverbrauch dieser Operation durch Messung.
Antwort:

☞ g_data

Aufgabe 8

Minimieren Sie nun den Stapelbedarf dieser Operation. *Welche Änderungen haben Sie vorgenommen? Wie wirken diese sich auf den Stapelverbrauch aus?*
Antwort:

Optionale Zusatzaufgabe

Statische Analyse mit `aiT` und `StackAnalyzer`

Um einen Einblick in die statische Stapelverbrauchs- und Laufzeitanalyse „in der Praxis“ zu erhalten, können Sie in dieser Aufgabe auf freiwilliger Basis die oben vorgestellten Anwendungen optional auch mittels der in der Industrie verbreiteten Programme `StackAnalyzer` und `aiT` der Firma AbsInt untersuchen.

Aufgabe 9 *Compiler*

Das Stackverbrauchsanalsysewerkzeug StackAnalyzer operiert direkt auf ausführbaren Dateien. Leider wird dieses Werkzeug nicht für die Intel-Architektur sondern nur für eingebettete Architekturen entwickelt. Deshalb müssen Sie für diese Aufgabe einen Cross-Compiler für die unterstütze ARM-Cortex-M4-Architektur verwenden. Dies erreichen Sie, indem Sie `cmake` wie folgt aufrufen:

```
cmake -DCMAKE_TOOLCHAIN_FILE=../cmake/armM4.toolchain ..
```

Beim Aufruf von `make` wird aus der pthreads-freien Vorgabe für ARM die Datei stackanalyzer_arm erzeugt, die nun weiter statisch analysiert werden kann.

☞ src/stackanalyzer_arm.c

Aufgabe 10 *Annotationen*

Machen Sie sich mit dem zur Verfügung gestellten Handbuch vertraut. Verschaffen Sie sich einen groben Überblick über die verfügbaren Annotationen.

Aufgabe 11 *Konfiguration*

Starten Sie die Werkzeug-Suite mit folgendem Befehl:

```
/proj/i4ezs/tools/a3_arm/bin/a3arm
```

Falls Sie nach einer Lizenzdatei gefragt werden, geben Sie folgenden Pfad an:

```
/proj/i4ezs/tools/a3_arm/license.dat
```

Machen Sie sich mit den verfügbaren Konfigurationsoptionen vertraut und wenden Sie diejenigen an, von denen Sie glauben, dass Sie notwendig sind oder Ihr Analyseergebnis positiv beeinflussen. *Welche sind das?*
Antwort:

Beachten Sie insbesondere, dass `ais`-Annotationen im Quellcode standardmäßig nicht von der Analyse beachtet werden. Die kann durch entsprechende Konfiguration aber geändert werden. Speichern Sie Ihr Projekt in Ihrem git-Repository ab und stellen Sie es unter Versionsverwaltung.

Aufgabe 12 *Analyse*

Analysieren Sie nun den maximalen Stackverbrauch der `run()` Funktion und interessanter einzelner Module wie z.B. `find_max()`. *Woran scheitern die Analysen zunächst?*
Antwort:

Wie können Sie den Werkzeugen helfen, das Programm trotzdem zu analysieren? Notieren Sie sich die nötigen Anpassungen der Implementierung oder Annotationen.
Antwort:

Vergleichen Sie die Ergebnisse für unterschiedliche Compileroptimierungen. Gerade `-O0` *und* `-Os` *sollten hier interessant sein.* Vergleichen Sie insbesondere den analysierten Stackverbrauch mit dem von Ihnen gemessenen. *Sind beide Werte aufgrund der unterschiedlichen Architekturen vergleichbar?*
Antwort:

Wie verhalten sich diese Werte im Bezug zu Ihrer Messung?
Antwort:

Aufgabe 13 *aiT*

Uns steht nicht nur der StackAnalyzer sondern auch der aiT für die statische Bestimmung der Worst-Case-Execution-Time(WCET) zur Verfügung. In dieser optionalen Aufgabe, können Sie zusätzlich auch eine statische Laufzeitanalyse des Programms durchzuführen. Die Nutzung des aiT ist der des StackAnalyzers sehr ähnlich. Arbeiten Sie sich auf die gleiche Art und Weise in seine Benutzung ein.

Hinweise

- Bearbeitung: Gruppenarbeit
- Abgabefrist: 09.07.2019
- Fragen bitte an `i4ezs@lists.cs.fau.de`