AUFGABE 6: STACKVERBRAUCHSANALYSE

In dieser Aufgabe werden Sie lernen, wie Sie nichtfunktionale 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. Abschließend werden Sie sich mit dem entsprechenden Werkzeug `StackAnalyzer` der Firma AbsInt 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, um die Aufrufe und ihre Parameter in der Datei `stackanalyzer.c` zu verstehen.

☞ `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 Wasserstandsmethode. Hierzu gehören hauptsächlich Funktionen zum Initialisieren des Stapelspeichers und zur Auswertung des bisher maximal genutzten Stapelspeichers.

☞ `make stackanalyzer_x86`

☞ `./ stackanalyzer_x86`

Aufgabe 3 *Visualisierung*

Um sich von der Korrektheit Ihrer Lösung zu überzeugen, ist es vorteilhaft, eine graphische Ausgabe des Stapelzustandes zu besitzen. In der Funktion `visualize()` finden sie eine Implementierungsskizze einer solchen Funktion, welche den Stapel dabei kompakt auf der Standardausgabe ausgibt. Dabei bezeichnet jedes ausgegebene Zeichen, wie viele Bytes des Wasserstandsmusters innerhalb von 32 Zeichen noch intakt geblieben sind. Machen Sie sich mit der Implementierung vertraut, erweitern Sie diese durch eine Erkennung Ihres Musters aus der vorherigen Aufgabe und rufen Sie die Visualisierung an geeigneter Stelle (oder auch Stellen) in Ihrem Programm auf. *Welche Regionen erkennen Sie in der Visualisierung? Erklären Sie insbesondere größere Blöcke, soweit möglich.*

Antwort:

Aufgabe 4 *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 höchsten Speicherverbräuche für die einzelnen Funktionen? Wie hoch ist dann der schlimmstmögliche Gesamtspeicherbedarf?*

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 5 *Problemstellen*

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

☞ `make stackgraph_arrays`

Hinweis: Zur Anzeige des Analysegraphen im Target `make stackgraph_arrays` wird ferner eine laufende graphische Oberfläche benötigt (Arbeit vor Ort im CIP).
Antwort:

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

Aufgabe 6 *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.

☞ `make stackgraph_arrays`

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. *Begründen Sie Ihren Lösungsweg, besonders bezüglich Konstanten in Ihrem Gleichungssystem.* 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. ☞ GET_TOS
Hinweis: Mit korrekten Eingaben sollte die Differenz zwischen statischem und dynamischem Wert nicht mehr als 200 Byte betragen. Greifen Sie bei Bedarf nochmals auf ihre Stapel-Visualisierung und den Aufrufgraphen zurück, um fehlerhafte Resultate zu identifizieren.
Antwort:

Aufgabe 7 *Übersetzeroptimierungen*

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

Statische Analyse mit der AbsInt-Werkzeugkette

Um einen Einblick in die statische Stapelverbrauchsanalyse „in der Praxis“ zu erhalten, können Sie in dieser Aufgabe die oben vorgestellte Anwendung auch mittels der in der Industrie verbreiteten Programme `StackAnalyzer` der Firma AbsInt untersuchen.

Aufgabe 8 *Annotationen*

Machen Sie sich mittels der zur Verfügung gestellten Folien sowie dem Handbuch mit der grundlegenden Bedienung des Programmes vertraut. Verschaffen Sie sich einen groben Überblick über die verfügbaren Annotationen.

Aufgabe 9 *Konfiguration*

Starten Sie die Werkzeug-Suite mit folgendem Befehl:

```
/proj/i4ezs/tools/a3_x86/bin/a3x86
```

Die Lizenz beziehen Sie dabei über unseren Lizenzserver. Nutzername und Passwort entsprechen dabei Ihren Anmeldedaten aus der initialen Mail. Kontaktieren Sie uns bitte, falls hierbei Probleme auftreten. Konfigurieren Sie die Analyse entsprechend der Vorgaben aus den Folien (CPU-Variante, Binärdatei, Analysestart, Annotationsextraktion aus Quelltextdateien, etc.). Speichern Sie Ihr Projekt in Ihrem git-Repository ab und stellen Sie es unter Versionsverwaltung.

☞ i4alm.cs.fau.de
Port: 42426

Hinweis: Die Warnungen der Analyse "ELF file is not an executable[...]" und "ELF file is not a statically linked executable[...]" sind durch die Natur der Anwendung (dynamisch gebundene Desktopanwendung) verursacht und dürfen durch Sie ignoriert werden. Alle anderen Warnungen/Fehler der Analyse sind jedoch durch Sie selbst geeignet zu adressieren.

Aufgabe 10 *Analyse*

Analysieren Sie nun den maximalen Stackverbrauch der Funktion `run()`. Optional können Sie gerne auch mit der Analyse weiterer Funktionen/Einstiegspunkte wie etwa `find_max()` experimentieren. *Woran scheitern die Analysen zunächst? Wie können Sie den Werkzeugen helfen, das Programm trotzdem zu analysieren? Notieren Sie sich die nötigen Anpassungen der Implementierung oder Annotationen. Wie verhalten sich diese Werte im Bezug zu Ihrer Messung, sowie Ihrer eigenen statischen Analyse?*

Antwort:

Erweiterte Übung

Warteschlange

Aufgabe 11

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 12

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

Optionale Zusatzaufgabe: Zeitanalyse mit `aiT`

Aufgabe 13 *Compiler*

Uns steht nicht nur der StackAnalyzer sondern auch der aiT für die statische Bestimmung der Worst-Case-Execution-Time(WCET) zur Verfügung. Das WCET-Analsysewerkzeug aiT operiert dabei 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 14 *aiT*

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 und analysieren sie die schlimmstmögliche Ausführungszeit der Methode `run()`.

Hinweise

- Bearbeitung: Gruppenarbeit
- Abgabefrist: 17.01.2025
- Fragen bitte an `i4ezs@lists.cs.fau.de`