Transformation

Dennis Ogiermann, Kristina Petrow

Einleitung:

Die Hauptaufgabe der Transformation besteht darin, die Transformationen bzw. Verschleierungen (Obfuscation Passes) des Codes, der von Frontend in die LLVM IR (Intermediate Representation) geliftet wurde, durchzuführen.

Die Aufgabe des Transformationsteams war mehrere Analysen und Transformationen (sogenannte Passes) zu implementieren, sowie 2 Test Cases zu schreiben.

Es folgt vorerst kurze Beschreibung des Programmablaufs, die genaueren Angaben zur einzelnen Funktionen sind ab der nächsten Seite zu finden.

Als Erstes wurden 2 Test Cases ausgesucht, die mehrere Funktionen und somit mehrere Basic Blöcke beinhalten, um möglichst große Testabdeckung zu gewährleisten.

Das erste Test Case behandelt Mean shift, ein nicht parametrisches Verfahren zum iterativen Finden von Maxima in Dichtefunktionen. Es wird ein Startfenster festgelegt, deren Verschiebung mithilfe des Gradienten zum Punkt der größten Verteilungsdichte erfolgt. Das zweite Test Case implementiert das Newton Verfahren, welches zur Annäherung einer Nullstelle einer Gleichung mithilfe der Tangente dient.

Die Transformation selbst beginnt mit dem Einlesen der Inputdatei, welche die Frontendphase bereits durchlaufen hat.

Zuerst müssen die Metadaten eingelesen und analysiert werden. Wenn die Analyse erfolgreich war, wird die Bytegröße jedes Basic Blocks berechnet.

Als nächstes werden die Blöcke so gespaltet, dass die Bytegröße jedes Basic Blocks nicht größer als der Mittelwert ist.

Trotz diesem Schritt gibt es Basic Blöcke, die kleiner sind als der Mittelwert und dementsprechend müssen diese Basic Blöcke an die Bytegröße des Mittelwertes angepasst werden. Das Aufpumpen von Basic Blöcken wird von Backendphase übernommen, jedoch ist die Transformationsphase für die Übermittlung dieser Informationen zuständig.

Abschließend wird die Liveness Analyse durchgeführt, um den Nutzungsstatus einzelner Register zu betrachten, damit das Backend weiß, welche Register es zum Auffüllen der Basic Blöcke nutzen darf. Über die Metadaten kann man sehen, welche Register in der aktuellen Instruktion geschrieben und welche gelesen werden.

Einblick in den Size Analysis Pass:

Verantwortliche: Dennis Ogiermann, Kristina Petrow

Die Aufgabe des „Size Analysis“ Passes ist es die übergebenen Metadaten zu analysieren und den Mittelwert der Bytegröße der Basic Blöcke zu berechnen.

Als Erstes durchläuft jeder Basic Block wird die Funktion „*analyzeBlockSize(…)*“, die für den aktuellen Basic Block die Bytegröße berechnet. Für jede Instruktion des aktuellen Basic Blocks wird die Funktion „*getInstructionSize(…)*“ aufgerufen. Diese Funktion liest die Metadaten, die von Frontend zur Verfügung gestellt werden, ein und summiert alle Instruktionsgrößen, die im aktuellen Basic Block vorkommen.

Sollten die Metadaten keine oder unvollständige Daten (z.B. Instruktionsgröße fehlt) enthalten, dann wird das Programm abgebrochen, da an diesem Punkt davon ausgegangen werden kann, dass die Daten beschädigt sind oder ein vorausgehender Fehler nicht abgefangen wurde.

Nachdem die beiden Funktionen durchlaufen sind, wird der Mittelwert ermittelt.

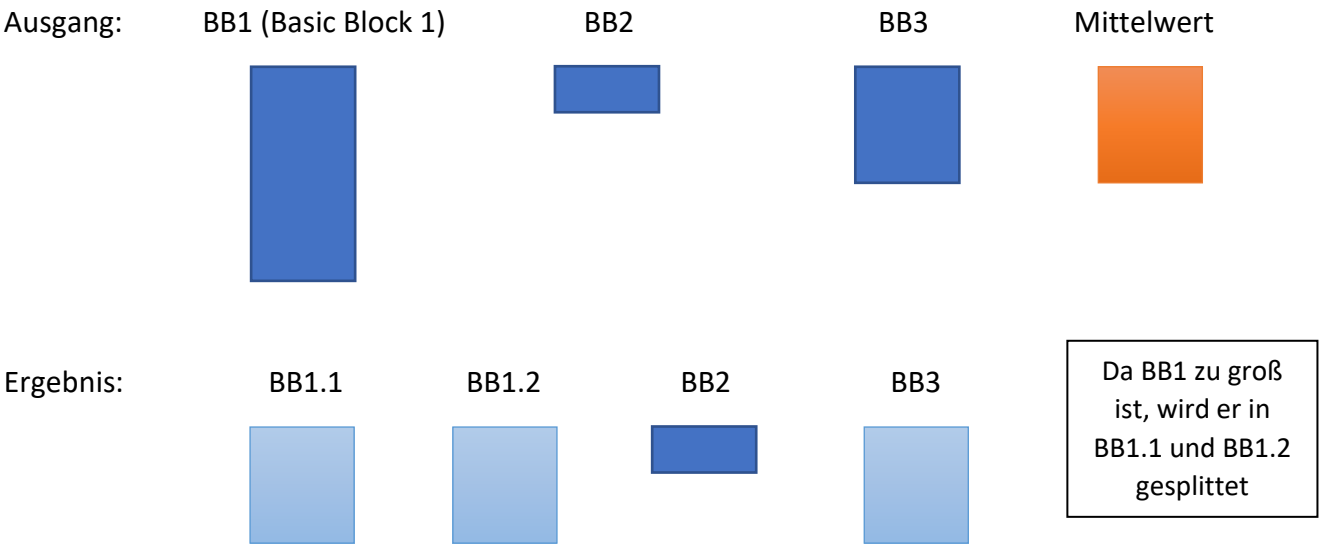
Einblick in den Splitting Pass:

Verantwortliche: Dennis Ogiermann, Kristina Petrow

Die Aufgabe dieses Passes ist es die Basic Blöcke, die größer als der Mittelwert sind, zu splitten, so dass nach dem Durchlauf dieser Funktion die Basic Blöcke entweder exakt die gleiche Größe wie der Mittelwert haben oder kleiner sind.

Für jeden Basic Block wird die Funktion *„continuousSplitting(…)“* gestartet. Zunächst läuft erneut eine Analyse („*analyzeBlockSize(…)*“ und folglich „*getInstructionSize(…)*“) durch. Gleichzeitig werden die Instruktionsgrößen (Info: eine Instruktion kann entweder 1 oder 2 Byte groß sein) aufaddiert, bis die Mittelwertgröße überschritten wird. Basic Blöcke werden mithilfe der Funktion *„splitBasicBlock(…)“*, die vom LLVM Framework zur Verfügung gestellt wird, vor der aktuellen Instruktion gesplittet. Der Rest des Basic Blocks wird zwischengespeichert, damit die Spaltung, falls notwendig, rekursiv fortgesetzt werden kann (rekursiver Aufruf der Funktion *„splitBasicBlock(…)“*). Bei der Spaltung wird eine Branch Instruktion eingefügt, der direkt an dieser Stelle die relevanten Metadaten angehängt werden.

Beispiel (vor und nachdem Splitting Pass):



Einblick in den Padding Pass:

Verantwortliche: Dennis Ogiermann, Kristina Petrow

Die Aufgabe des Padding Passes ist es die Basic Blöcke, deren Größe kleiner als der Mittelwert ist, zu finden und die fehlende Bytegröße zu ermitteln, die später dem Backend über die Metadaten mitgeteilt wird.

Zuerst läuft die Suche nach solchen Basic Blöcken. Wenn so ein Basic Block gefunden wurde, wird die Differenz zwischen der aktuellen Größe des Basic Blocks und die vom Mittelwert ermittelt.

Danach wird am Anfang diesen Basic Blocks eine NOP Instruktion eingefügt und über die Metadaten mittels der Funktion *„appendMetadata(…)*“ unter dem Parameter *„instructionSize“* die fehlende Bytegröße angehängt. So weiß das Backend um wie viele Byte das aktuelle Basic Block aufgepumpt werden soll.

Einblick in den Def-Use Analysis Pass:

Verantwortliche: Dennis Ogiermann, Kristina Petrow

In diesem Pass wird Liveness aller Register zu jedem Zeitpunkt analysiert und über Metadaten dem Backend übermittelt.

Liveness Analysis der Register muss ebenfalls durchgeführt werden, da MSP430 Architektur nur 16 Register unterstützt und LLVM IR dagegen unendliche Anzahl an virtuellen Registern bietet.

Über Metadaten einzelner Instruktionen, die von Fronend übergeben werden, werden noch zusätzlich zwei Vektoren übergeben. Das eine Array sagt, welche Register in der aktuellen Instruktion geschrieben und welche gelesen werden. Diese zwei Vektoren sind identisch aufgebaut. Die Zuordnung der Felder dieser zwei Vektoren zu den Registern erfolgt folgendermaßen:

Z.B.: Vektor „write“ {x, <wert>, x, …} → Register {R0, R1, R2, …} („x“→ kein Wert hinterlegt)

- Dieser Vektor sagt, dass in der aktuellen Instruktion ein Wert in Register R1 geschrieben wird

Z.B.: Vektor „read“ {x, x, <wert>, …} → Register {R0, R1, R2, …}

- Der Vektor „read“ sagt, dass in der aktuellen Instruktion ein Wert in Register R1 geschrieben wird

In diesem Pass werden 2 Vektoren *„UpExp“* und *„VarKill“* erzeugt (UpExp =Upwards Exposed Reads und VarKill = Written or killed variables). Zuerst folgt die Auslesung der einzelnen Arrayfelder aus den Metadaten (für „write“ und „read“ Variablen getrennt).

<u>Aufbau</u>: UpExp[b][c] und VarKill[b][c]

Diese beiden Vektoren sind zweidimensional, das erste Feld [b] gibt die Nummer des Basic Blocks an und das zweite Feld [c] gibt die Nummer des Registers(s. Bsp. oben) an, wobei b ein Integer und c ein Boolean ist. In der Zeit, wenn die Metadaten ausgelesen werden, wird an allen Werten eines Registers, das in den Instruktionen im aktuellen Basic Block vorkommt, ein ODER-Operator angewendet.

Beispiel*:* UpExp[2][1] ist auf „true“ gesetzt → bedeutet, dass im Basic Block 2 Register 1 gelesen wird

Im Endeffekt erhalten wir zu jeden Basic Block ein Array, das uns sagt, welche Register geschrieben oder gelesen werden.