Compilers Practicum 4

Yasser Deceukelier Titouan Vervack

19 April 2016

1 Statische bounds-check

In de pass die we geïmplementeerd hebben, wordt `getelementptr` geanalyseerd en worden de argumenten van de instructies gecontroleerd. Indien de indexering van een array met een constante gebeurt, dan kan de compiler zelf controleren of de index binnen de grenzen van de array ligt. In dit geval kan deze check gebeuren tijdens het compileren, en hoeven er geen extra instructies uitgevoerd worden tijdens de uitvoering van het programma. Dus, een statische check heeft geen invloed op de uitvoeringstijd van de resulterende executable.

2 Dynamische bounds-check

2.1 Bounds-checks

In het geval dat de waarde voor de index niet gekend is bij het compileren zal extra code nodig zijn om tijdens het uitvoeren van het programma, te controleren of de index binnen de grenzen van de array liggen. Hiervoor moeten drie extra basic blocks worden toegevoegd aan het programma: één om te controleren of de index kleiner is dan de grootte van de array, en één om te checken of de index groter of gelijk is aan nul. Het derde basic block bevat de instructies om een foutmelding te geven en de uitvoering stop te zetten. Indien de check in één van de eerste twee basic blocks faalt, wordt naar het laatste basic block gesprongen en wordt de uitvoering met een foutmelding stopgezet. In het geval van een geldige index zal het basic block met de eerste check springen naar het basic block met de tweede check en dit zal op zijn beurt doorspringen naar het basic block dat de rest van de originele code bevat.

2.2 Invloed op intermediare code

Voor het meten van de impact van deze extra instructies gaan we er vanuit dat de index geldig is, aangezien de uitvoering wordt stopgezet indien dit niet het geval zou zijn. Als we deze veronderstelling maken, kunnen we uit de intermediare code afleiden dat er slechts vier extra instructies zijn tijdens de uitvoering: een vergelijking van de index met een constante, en een conditionele sprong gebaseerd op het resultaat daarvan. Dit gebeurt dus voor twee constanten: de grootte van de array en 0.

2.3 Metingen

Om de impact van de bounds-checks te meten hebben we gebruik gemaakt van twee programmas: `fib.c` en `nestedloop.c`. Het eerste programma berekent de eerste 100.000 Fibonacci-getallen met behulp van een array. De berekening van één Fibonacci-getal gebeurt met drie geheugentoegangen: het lezen van de twee vorige getallen en het schrijven van het nieuwberekende getal. Bij deze geheugentoegangen wordt telkens een bounds-check uitgevoerd, wat dit een geschikte kandidaat maakt om de overhead van deze checks mee te bepalen. We laten het berekenen van de eerste 100.000 Fibonacci-getallen 10.000 keer gebeuren zodat de uitvoeringstijd significant genoeg is.

Het tweede programma voert een geneste lus uit. De binnenste lus doet telkens een geheugentoegang naar een array en telt de waarde op bij een variable uit de buitenste lus. De buitenste lus assigneert deze variable dan aan een positie in de array. De metingen van beide programma's, zowel met als zonder bounds-check kunnen in tabel 1 gevonden worden. Elk programma werd getimed met behulp van het time commando en werd 100 keer uitgevoerd om een stabiele gemiddelde uitvoeringstijd te bekomen.

Programma	Geheugentoegangen	Uitvoeringstijd (s)		Overhead	
		Origineel	Bounds-checked	Tijd (ms)	Procentueel
nestedloop	$\frac{n*(n+1)}{2} = 5.000.150.000$	$4,58$	$4,62$	40	$0,8\%$
fib	$3 * 10^9$	$2,42$	$2,60$	180	$7,4\%$

Tabel 1: Metingen (gemiddelde uitvoeringstijden)

2.4 Minimaliseren van de impact

De impact van deze extra instructies wordt beperkt door enkele hardware features, waarvan speculatieve uitvoering en branch prediction de meeste invloed hebben. De twee checks zullen tijdens de uitvoering telkens hetzelfde resultaat hebben; indien dit niet zo zou zijn, zou de uitvoering stoppen. Doordat deze checks hetzelfde resultaat hebben, zal de branch predictor deze uiteindelijk leren, en zal de sprong correct voorspeld worden. De speculatieve uitvoering zal dan de instructies in dit pad beginnen uitvoeren in de pipeline, en zo hoeft er niet gewacht te worden op het resultaat van de check, waardoor de overhead beperkt wordt. De impact van de extra instructies is dus beperkt tot de uitvoering van de instructies zelf en heeft geen extra overhead zoals bijvoorbeeld geheugenoperaties of *nop's* in de pipeline.

Daarnaast kan de invloed nog beperkt worden door de layout van de basic blocks: basic blocks die in een correcte uitvoering na elkaar zullen uitgevoerd worden, worden best ook dicht bij elkaar geplaatst in de binary, zodat de instructie cache ook optimaal wordt gebruikt.

3 Conclusie

We concluderen dat de impact niet al te groot is, in ons eerste test geval is de overhead verwaarloosbaar klein. In het tweede geval is het slechts 7.4%. We zien ook dat deze code niet veel verder valt te verbeteren en dat de overhead rechtstreeks komt uit de verhooging van het aantal instructies. Door extra optimalisaties uit te voeren zou de overhead echter nog verder kunnen verkleind worden. Zo denken we aan invullen van variabelen als hun waarden al gekent is waardoor de boundscheck een statische i.p.v. een dynamische check kan worden. Een andere optimalisatie is het liften van checks, indien we bvb. `list[i] = list[i-1] + list[i-1]` hebben, een instructie waar dezelfde check meerdere keren wordt uitgevoerd, dan zouden we deze gewoon één keer khunnen uitvoeren.