1 maart 2010

Compilerbouw

Peephole optimizer

Tim van Deurzen (5742900), Koos van Strien (5783437)

Universiteit van Amsterdam

Informatica instituut

Faculteit voor Natuurkunde, Wiskunde en Informatica

HOOFDSTUK 1

Peephole Optimizer

1.1 Inleiding

Dit document beschrijft het resultaat van het practicum voor het vak Compilerbouw. De opdracht was om een *Peephole Optimizer* te bouwen de overweg kon met **SimpleScalar DLX assembly** code. Hierbij konden verschillende optimalisaties worden uitgevoerd, te bepalen door de studenten.

1.1.1 Vernieuwde algoritmes

In dit document zal de nieuwe versie van de peephole optimizer worden beschouwd. Deze versie verschilt op een aantal punten van de vorige versie. Sinds de vorige versie van deze optimizer, is veel aan de algoritmiek verbeterd. In eerste instantie waren de resultaten niet consistent en konden metingen niet exact worden herhaald. Dit probleem is opgelost door de gebruikte algoritmes opnieuw op te bouwen op een minder complexe wijze.

De nieuwe algoritmen zijn iets makkelijker te begrijpen en zijn minder optimistisch dan de vorige algoritmen. Hierdoor werd het mogelijk de bestaande fouten te verbeteren. Ook is nog een extra algoritme toegevoegd, namelijk het optimaliseren van ‘onhandige branches’. Deze optimalisatie zal worden uitgediept in een volgende sectie.

1.2 Implementatie

De optimizer is geïmplementeerd in Python. Deze programmeertaal (of eigenlijk scripttaal) is gekozen, omdat er snel mee geprogrammeerd kan worden. Ook lenen de ingebouwde datastructuren, zoals lists en dictionaries, zich goed voor een programma zoals dit.
Het programma is vrij modulair opgebouwd. Er zijn aparte klassen voor:

- Het verwerken van gebruikersinput.
- Het indelen in basic blocks.
- Iedere optimalisatie.

Op deze manier is het makkelijk om verschillende optimalisaties buiten beschouwing te houden bij het uitvoeren.

Ten opzichte van de vorige versie van deze optimizer is het indelen in basic blocks verbeterd. Momenteel worden eerst alle 'leaders' opgezocht en worden vervolgens alle operaties vanaf een leader tot, maar *niet* inclusief, de volgende leader in een basic block gezet.

1.2.1 De optimalisaties

Hier worden kort de algoritmen voor de verschillende optimalisaties besproken. Voor de gedetaileerde werking kan de code worden gelezen. Om dit document kort te houden worden alleen de aangepaste algoritmen beschreven. Voor de oude algoritmen kan het vorige verslag worden bekeken.

Copy propagation

Copy propagation is een optimalisatie die onnodig verplaatsen van data optimaliseert. Bijvoorbeeld: het verplaatsen van een variabele *a* naar *b*, om vervolgens direct *b* in het geheugen op te slaan, terwijl *a* onaangepast is. Deze verplaatsing is onnodig en kan worden weggehaald.

Het optimaliseren gaat als volgt:

Iedere operatie in een basic block wordt bekeken. Wanneer de huidige operatie géén 'move' operatie is, stapt het programma gelijk door naar de volgende operatie. Is de operatie wel een move, dan worden stuk voor stuk alle volgende operaties bekeken. Als een van die volgende operaties een aanpassing maakt aan de registers van de huidige *move-operatie* kunnen er twee dingen gebeuren:

- De loop kan worden verbroken en de move operatie wordt niet geoptimaliseerd.
- De huidige operatie wordt zo aangepast dat deze zonder de move correct blijft functioneren.

Als er een "Jump and Link" instructie voorbij komt, wordt de move niet geoptimaliseerd, tenzij dat al door een eerdere instructie wel was gebeurd. Jump en Link instructies kunnen worden gezien als het aanroepen van een functie. In zo'n functie kunnen dingen gebeuren die effect hebben op het optimaliseren.

Redundant loads en stores

Het optimaliseren van redundante *load* of *store* operaties zorgt ervoor dat het geheugen niet meer wordt aangesproken dan nodig is. Zo is het niet nodig een waarde te laden direct volgend op een operatie die deze waarde opslaat.

Het optimaliseren gaat als volgt:

Er wordt over alle operaties binnen het basic block gelopen. Als een operatie een *load* of *store* operatie is, dan worden vervolgens alle volgende operaties nader bekeken. Als er een identieke *load* operatie wordt gevonden, dan zal deze worden verwijderd. En als er een *load* operatie wordt gevonden direct na, of dichtbij, een *store* operatie dan wordt ook deze

operatie verwijderd. Wordt er een ander type operatie gevonden en past deze operatie een van de registers van de huidige *load* of *store* aan, dan wordt de loop doorbroken en kan er geen operatie worden verwijderd.

Ook hier geldt dat als er een "Jump and Link" operatie langs komt de loop wordt doorbroken en er geen operatie wordt vewijderd.

Global branch optimalisatie

Een extra optimalisatie, die is toegevoegd aan de tweede versie van deze optimizer, is een globale optimalisatie voor branches. Er komt in de assembly af en toe de volgende situatie voor:

Een conditionele branch naar een bepaald label, daarna een jump naar een ander label en vervolgens het 'doel-label' van de eerste branch. Deze constructie kan worden ingekort door de branch te 'inverteren' en naar het label van de jump operatie te springen. Dit levert uiteindelijk hetzelfde gedrag op, want als de conditie niet houdt gaat het programma naar het label van de (oude) jump. En in het andere geval, wordt de code onder het label van de branch uitgevoerd.

Deze optimalisatie loopt niet over basic blocks, maar over alle operaties binnen het programma. Wanneer er een branch wordt gevonden, worden direct de volgende twee operaties bekeken. Als de hierboven omschreven situatie wordt geconstateerd, dan zal de code worden aangepast. De brach wordt dan geïnverteerd en de jump verwijderd.

1.3 Resultaten

Helaas leveren niet alle optimalisaties ook echt een versnelling op. Dit komt waarschijnlijk door de opbouw van de verschillende programma's. Het ene algoritme laat zich beter optimaliseren dan het andere.

Hieronder een kleine tabel met versnellings-, dan wel vertragingspercentages:

Voor deze uitvoer zijn alle optimalisaties, behalve globale branch optimalisatie uitgevoerd.

Benchmark	cycles origineel	cycles geoptimaliseerd	versnelling (%):
acron	5138949	5127044	0.23
clinpack	1522838	1835313	-20.51
dhrystone	2884781	2859714	0.87
pi	1709987	1702943	0.41
slalom	2863936	2336204	18.23
whet	2835438	2803240	1.14

De volgende resultaten zijn inclusief de globale branch optimalisatie.

Benchmark	cycles origineel	cycles geoptimaliseerd	versnelling:
acron	5138949	5576533	-8.5
clinpack	1522838	1870891	-22.8
dhrystone	2884781	2824640	2.08
pi	1709987	1702928	0.41
slalom	2863936	2332539	18.55
whet	2835438	2756553	2.78

De globale branch optimalisatie geeft helaas niet overal correcte resultaten. De output van de benchmark 'acron' geeft een aantal extra regels, die niet in de originele output terug te vinden zijn. De regels hebben wel hetzelfde formaat en lijken eigenlijk gewoon extra resultaten te zijn.

1.4 Conclusie

Niet iedere benchmark heeft baat bij het uitvoeren van peephole optimalisaties. Er zitten zelfs benchmarks tussen (clinpack) die er behoorlijk op achteruit gaan. De gekozen algoritmes zijn niet zo heel erg agressief, dus misschien kunnen er nog betere resultaten worden behaald door wat optimistischer naar de operaties te kijken.

Wat heel duidelijk wordt, is dat optimaliseren van assembly behoorlijk complex is. Het vereist een hoop inzicht in de werking van de code en de manier waarop de assembly wordt uitgevoerd.

Een volgende peephole optimizer zou waarschijnlijk op een andere manier worden opgebouwd. Er zou dan beter worden gekeken naar de voordelen van bepaalde datastructuren met betrekking tot het vinden van bepaalde situaties.

Concluderend: het is een complexe taak om een peephole optimizer te bouwen, terwijl de begrippen nog geleerd worden. Maar het is een leuke en inspirerende uitdaging, die snel resultaat geeft.