Voorwoord

Dit rapport is geschreven in het kader van het bachelor Elektrotechniek eindproject. Dit project, uitgevoerd door twee studenten van de opleiding elektrotechniek, werd uitgevoerd in opdracht van een werkgroep van de faculteit EWI. Wij hebben gewerkt aan een opdracht uit de groep Computer Engineering, waarbij we onder begeleiding stonden van Koen Bertels en Roel Meeuws.

Twee maanden lang zijn we bezig geweest met het verbeteren van zogenaamde code profiling, het extraheren van kenmerken uit c-code, om daarmee een schatting te kunnen geven over de complexiteit van een hardware implementatie van die code. Dit werd gedaan in het kader van het Delft Workbench project, een groot project om efficiënt snelle hardware te kunnen maken.

Onze dank gaat uit naar onze 2 begeleiders voor hun ondersteuning en naar onze studentcollega's, voor de gezelligheid.

Bij het schrijven van dit verslag is er vanuit gegaan dat de lezer enige kennis heeft van hardware en c-code.

Samenvatting

Vanuit een groot aantal bedrijven is er een toenemende vraag naar snellere hardware. Het MOLEN-platform voorziet in die behoefte, doordat de architectuur bestaat uit een normale processor en een herprogrammeerbare chip. Hierdoor is een grote versnelling te behalen. Het omzetten van programma's naar deze architectuur is echter lastig en tijdrovend. De Delft Workbench automatiseert dit proces, waardoor het omzetten net zo eenvoudig wordt als het compilen van een programma. Bovendien is deze softwaretool te gebruiken voor verschillende soorten processoren en herprogrammeerbare chips.
Een van de eerste zaken die gebeurt, is het analyseren van de programma's die binnenkomen.

In dit rapport komt de analyse van c-code aan de orde, het zogenaamde profilen. In deze analyse wordt een schatting gemaakt van het mogelijk oppervlak dat een programma inneemt op een herprogrammeerbare chip. Ook wordt gekeken in hoeverre een programma is te parallelliseren. Deze waardes worden niet direct bepaald, maar worden geschat aan de hand van bepaalde eigenschappen, zogenaamde metrics.

Er wordt gekeken naar verschillende metrics die een schatting kunnen geven voor de mogelijke grootte op de herprogrammeerbare chip. Er is gekeken naar het aantal aritmetische operaties (rekenkundige, bitwise of logische operaties) om een maat voor de oppervlakte te krijgen. Daarnaast wordt gekeken naar de bit-breedte van de input van een functie. Hieraan is te zien hoeveel exchangeregisters er nodig zullen zijn tussen de normale processor en de herprogrammeerbare chip. Dit is geeft ook een maat voor de oppervlakte. Ook wordt bekeken hoeveel variabelen er in een functie zitten en wat daarvan de bit-breedte is. Dit geeft aan hoeveel registers er ongeveer nodig zullen zijn.

Er wordt ook gekeken naar metrics die een schatting kunnen geven voor de mate waarin een programma is te parallelliseren. Basic blocks is een aantal achtereenvolgende operaties die uitgevoerd worden zonder te verspringen. Zo'n verspringing treedt bijvoorbeeld op bij een jump.

Daarnaast wordt er een schatting gemaakt van de algemene complexiteit van de code. Dit kan zowel bruikbaar zijn voor een oppervlakte schatting als voor een schatting over de mogelijke te halen versnelling.

De metrics worden gebruikt voor het maken van een schatting van eigenschappen op een fysieke implementie op herconfigureerbare hardware. Hierbij wordt een schatting gemaakt van het aantal states, flipflops, slices en LUT's.

Code die in de profiler komt, moet aan een paar eisen voldoen. In de c-code mogen zich geen function-calls, geen pointers en geen arrays bevinden. Als aan deze eisen wordt voldaan, levert het programma op basis van testen correcte resultaten op.

Ondersteuning van arrays zou het toepassingsgebied van de profiler aanzienlijk vergroten. Daarom moet daar verder onderzoek naar worden gedaan. Schattingen voor de grootte van een array zou kunnen worden bereikt door het dynamisch gedrag van de code te bepalen. Dynamisch gedrag kan worden bepaald door de code vaak te runnen met verschillende parameters.

Inhoudsopgave

1 Inleiding

Vanuit een groot aantal bedrijven is er een toenemende vraag naar snellere hardware. Het MOLEN-platform voorziet in die behoefte, doordat de architectuur bestaat uit een normale processor en een herprogrammeerbare chip. Hierdoor is een grote versnelling te behalen. Het omzetten van programma's naar deze architectuur is echter lastig en tijdrovend. De Delft Workbench automatiseert dit proces, waardoor het omzetten net zo eenvoudig wordt als het compilen van een programma. Bovendien is deze softwaretool te gebruiken voor verschillende soorten processoren en herprogrammeerbare chips.

Een van de eerste zaken die gebeurt, is het analyseren van de programma's die binnenkomen. In deze analyse worden eigenschappen van de code geextraheerd die relevant zijn voor de verdere omzetting.

Doel van dit rapport is het presenteren van onze oplossing om deze eigenschappen te extraheren uit de code. Er wordt ook aandacht besteed aan de keuze van de eigenschappen. Er wordt beschreven hoe de extrahering is geïmplementeerd in de elsa parser.

In hoofdstuk 2 komt de totale Delft Workbench aan de orde. In hoofdstuk 3 worden de eisen die aan het product zijn gesteld genoemd. In hoofdstuk 4 wordt het concept van het programma besproken. In hoofstuk 5 wordt de daadwerkelijke implementatie besproken. In hoofdstuk 6 zijn de resultaten te vinden. Hoofdstuk 7 bevat de conclusies en aanbevelingen.

2 Delft Work Bench

2.1.1 Inleiding DWB

De Delft Workbench (DWB) is een applicatie om snel software geschikt te maken voor herconfigureerbare hardware. Hierbij wordt gebruik gemaakt van de MOLEN architectuur. MOLEN is een aan de TU Delft ontwikkelde architectuur bestaande uit een General Purpurse Processor (GPP) en een FPGA. De architectuur in combinatie met een speciale compiler zorgt voor de communicatie tussen beide chips.

Op deze manier is het mogelijk om rekenintensieve taken enorm te versnellen. Dit platform is hardware onafhankelijk, wat grote verscheidenheid aan implementaties mogelijk maakt. Zo is het bijvoorbeeld zelfs mogelijk de GPP te simuleren op de FPGA.

De keuze welke functie er geimplementeerd gaat worden en de bijbehorende VHDL-code die nodig is om de FPGA te programmeren kan door de DWB bijzonder snel geconstrueerd worden. Hierdoor is er veel tijdwinst te behalen met alle voordelen van dien. Zeker in sterk dynamische markten zoals de telecom sector waar men een korte time to market heeft en grote concurrentie, kan dit een enorm voordeel opleveren.

2.1.2 Doelstelling DWB

De doelstelling van de DWB-groep is om een softwarepakket te maken dat (uiteindelijk) commercieel bruikbaar is. Met het programma moet iedereen, ook mensen zonder uitgebreide hardware kennis, in staat zijn code versneld te laten draaien op het MOLEN-platform. Door de hardware onafhankelijkheid van MOLEN is het softwarepakket breed inzetbaar en kan het toegepast worden in zeer veel verschillende markten.

2.2 Onderdelen DWB

De DWB bestaat uit een viertal hoofdonderdelen. Samen vormen zij de totale ketting van c-code tot een hybride applicatie die gedraaid kan worden op het MOLEN-platform. Om de profiler in context te kunnen plaatsen worden de vier onderdelen kort toegelicht.

2.2.1 Profiling

Allereerst wordt de code door de profiler gehaald. Hier wordt gekeken welke stukken code geschikt zijn om te implementeren op de FPGA. Hierbij wordt bijvoorbeeld gekeken of de functie kan worden geimplementeerd, zo mogen er geen pointers en objecten in voorkomen. Daarnaast is het belangrijk dat de functie veel snelheidwinst oplevert, en dat hij past op het FPGA oppervlakte. Om deze eigenschappen te achterhalen wordt er gekeken naar een verzameling zogenaamde 'metrics' uit de c-code. Door deze metrics door een model te halen kan je een uitspraak doen over de eigenschappen van de FPGA implementatie. In dit verslag wordt uitgebreid ingegaan op code profiling.

2.2.2 Graph Transformation

De functies die door de profiling worden aangewezen als potentieel bruikbare functies worden doorgegeven voor de 'graph transformation'. Hier wordt geprobeerd de c-code, welke normaal gesproken geoptimaliseerd is voor een GPP, om te vormen naar een paralelle vorm, geschikt voor implementatie in hardware. Om dit te bereiken wordt er gebruik gemaakt van twee technieken.

- Graph restructuring:

Hier wordt gekeken naar groepen 'oude' instructies, om deze samen te voegen naar een nieuwe instructies in de hardware.

- Loop parallelization:

Hier wordt gekeken naar loops die de nieuwe instructies bevatten. Bij deze loops wordt geprobeerd of ze getransformeerd kunnen worden naar een paralelle vorm. Dit kan door middel van bijvoorbeeld loop unrolling, software pipeling en loop tiling.

2.2.3 Retargetable Compilation

Op dit moment is er een keuze gemaakt voor een functie die geimplementeerd gaat worden op de FPGA. Hiervoor moet de originele code worden aangepast. De functie wordt weggehaald en vervangen door een call naar de FPGA. Daarnaast moet er gedacht worden aan het klaar zetten van data voor de FPGA en het ophalen van de gegevens achteraf. Hier moet rekening gehouden worden met verschillende factoren zoals timing en architectuur.

2.2.4 VHDL Generation

De functies die op de FPGA geimplementeerd gaan worden moeten van c-code omgezet worden naar een hardware description language zoals Verilog of VHDL. Om hier te komen zijn er 3 verschillende mogelijkheden.
Voor zeer kritieke stukken code moet er met de hand VHDL of Verilog code geschreven worden.
Als de code 'standaard' is, kan er gebruik gemaakt worden van een libary. Hier kan gedacht worden aan bijvoorbeeld een MPEG decoder.
De laatste manier is het automatisch genereren van code. De kwaliteit van deze code is nog inferieur aan zelfgemaakte code, maar het is zeer geschikt voor het snel kunnen testen van ontwerpen en om een idee te krijgen van de mogelijkheden. Achteraf kunnen er met de hand nog dingen aangepast worden om hogere kwaliteit te krijgen.

3 Programma van eisen Profiling

3.1 Inleiding

De eerste stap die de Delft Workbench maakt is profiling. Profiling is het analyseren van de c-code, om daar bepaalde eigenschappen uit te halen, zogenaamde metrics. Deze metrics worden verderop in de DWB gebruikt om te bepalen welke stukken code in aanmerking komen voor omzetting naar de herconfigureerbare hardware. Doel van dit project is profilen van c-code.

In paragraaf 2 is de opdrachtomschrijving te vinden. In paragraaf 3 bevinden zich de randvoorwaarden waaraan het programma moet voldoen.

3.2 Opdrachtomschrijving

Doel van dit project is het extraheren van eigenschappen uit c-code die relevant zijn voor verdere omzetting naar herconfigureerbare hardware. Een stuk code moet aan bepaalde eisen voldoen. Zo kunnen pointers niet omgezet worden naar hardware. Deze en andere voorwaarden staan beschreven in de randvoorwaarden.

Als een stuk code voldoet aan deze voorwaarden, dienen er eigenschappen te worden geextraheerd die belangrijk zijn als de code wordt omgezet naar een hardwarebeschrijving voor de herconfigureerbare hardware.

Ten eerste dient de mogelijke hardwarebeschrijving die uit de code volgt, te passen op de herconfigureerbare hardware. De hardwarebeschrijving is nog niet bekend, dus vanuit de code dient een schatting te worden gemaakt van het mogelijke oppervlak. Het is nutteloos om een supersnelle oplossing voor herconfigureerbare hardware te ontwerpen, als het er vervolgens niet fysiek op past. Daarom wordt er gezocht naar eigenschappen van de c-code die een maat geven voor de mogelijke grootte van de hardware.

Ten tweede is er een grote versnelling mogelijk door de sequentieele code om te zetten naar parallelle hardware. In plaats van alles stapje voor stapje achter elkaar uit te voeren op de normale processor, kunnen er op de herconfigureerbare hardware stukken code parallel uitgevoerd worden.

Als er een conditional zoals een if-statement optreedt, wordt op een normale processor mogelijk naar een ander stuk code gesprongen. Het is vrijwel onmogelijk om dit op een efficiente manier op parallelle hardware te doen. Daarom wordt gezocht naar eigenschappen van de c-code die een maat geven voor in hoe verre de code parallel is uit te voeren.

Ten slotte worden metrics van c-code gebruikt voor het maken van schattingen die dicht bij de fysieke implementatie van een hardwarebeschrijvingen op herconfigureerbare hardware.

3.3 Randvoorwaarden

De code die de profiler inkomt, moet aan drie voorwaarden voldoen. Deze voorwaarden worden grotendeels opgelegd door de omzetting naar hardware.

Function-calls zijn ingewikkeld om uit te voeren in hardware. Op dit moment is het niet mogelijk om code met function-calls om te zetten naar hardware.

Pointers worden alleen in software gebruikt. In hardware bestaat niet zoiets als pointers. Op dit moment is het niet mogelijk om code met pointers om te zetten naar hardware.

Arrays zijn lastig om te maken in hardware. Als niet gedefinieerd is hoe groot een array is, is het ook ongedefinieerd hoeveel registers er bijvoorbeeld nodig zijn. Dit bemoeilijkt de omzetting naar hardware.

In de c-code mogen zich dus geen function-calls, geen pointers en geen arrays bevinden.

4 Concept

4.1 Inleiding

In dit hoofdstuk wordt het concept voor ons programma besproken. Hierbij wordt nog niet ingegaan op de implementatie. Door nog niet aan implementatie te denken, wordt bepaald wat het programma daadwerkelijk moet gaan doen. In paragraaf 2 zijn omschrijvingen te vinden van de metrics. In paragraaf 3 is een omschrijving te vinden van de schattingen die dicht bij de fysieke implementatie van hardwarebeschrijvingen op herconfigureerbare hardware staan. In paragraaf 4 is beschreven hoe de output van ons programma is gedefinieerd.

4.2 Te bepalen metrics

Een van de doelen is om een schatting te maken van de oppervlakte die ongeveer nodig is op de herconfigureerbare hardware. Het aantal aritmetische operaties (zoals optellingen) en het totale aantal bits dat gebruikt wordt in een stuk programma geeft daar een maat voor. Aritmetische eenheden zoals adders en multipliers nemen meestal een aanzienlijk deel in van het chipoppervlak. Het totale aantal variabele bits geeft aan hoeveel registers er ongeveer nodig zullen zijn. Deze registers nemen ook veel chipoppervlak in. Los daarvan is het ook nuttig om het aantal parameter bits van de input te weten. Het aantal inputs en outputs bepalen hoeveel exchange-registers er ongeveer nodig zijn.

Naast deze eigenschappen voor het chipoppervlak, willen we ook de mate van versnelling weten. De sequentiële code moet kunnen worden omgezet naar parallelle hardware. Dat is alleen mogelijk als er geen conditionals, zoals if- en for statements optreden Stukken code waarin geen conditionals optreden kunnen in het algemeen parallel worden uitgevoerd. Zulke stukken code noemen we basic blocks.

Als laatste wordt er gekeken naar de algemene complexiteit van de code. Hieruit kan zowel een maat voor de versnelling als een maat voor de oppervlakte worden gehaald.

Verdere details over de metrics worden beschreven in de volgende subparagrafen.

4.2.1 Omschrijving metric 1: Arithmetic operaties

Aritmetische eenheden zoals adders en multipliers nemen meestal een aanzienlijk deel in van het chipoppervlak. In de code doen aritmetische operaties zich voor als enkelvoudige en tweevoudige operaties. Enkelvoudige, of unary, operaties zijn rekenkundige operaties die maar een input hebben. Denk hierbij aan bijvoorbeeld i = i + 1. In tweevoudige, of binary, operaties zijn er twee inputs. Een voorbeeld hiervan is c = a + b. We maken dit onderscheid, omdat

enkelvoudige operaties minder chipoppervlak innemen dan tweevoudige operaties.

4.2.2 Omschrijving metric 2: Bit-breedte van programma

Het totale aantal bits geeft aan hoeveel registers er ongeveer nodig zullen zijn. Deze registers nemen veel chipoppervlak in. Bovendien neemt een arithmetic operatie van 2x32bit getallen meer ruimte in dan een paar van 8bit.

Zoals eerder vermeldt, wordt aangenomen dat er geen pointers en arrays voorkomen. Structs en enums echter wel, en deze dienen ook ondersteund te worden.

4.2.3 Omschrijving metric 3: Bit-breedte van de input

Het aantal bits wat aan een functie wordt meegegeven is een maat voor hoeveel data er tussen de FPGA en de GPP moet worden verstuurt. Doorgeven van data zal gebeuren door zogenoemde exchange registers. De FPGA zet daar het resultaat ook weer in terug, zodat de GPP het kan uitlezen en gebruiken.

4.2.4 Omschrijving metric 4: Basic blocks

Een basic block is een aaneenschakeling van code welk altijd sequentieel wordt uitgevoerd. Zoals beschreven is een basic block geschikt om te paralleliseren. Om dit goed te doen is het nodig een paar eigenschappen te ontrekken. Zo wordt er bijvoorbeeld gekeken naar hoeveel basic blocks er in de code voorkomen, hoe lang de langste is en naar wat verhoudingen tussen het aantal expressies, statements en basic blocks. Met deze informatie kan een schatting worden gemaakt in hoever een functie geparalleliseerd kan worden. Dit hangt van nog veel meer factoren af zoals data afhankelijkheid, welke we hier niet zullen bespreken. Een kort voorbeeld van 2 basic blocks is hieronder weergegeven.

```
int a,b;
a = 0;
b = 0;
for (int i=0; i<10; i++) {
        a = a + i;
        b = b - i;
}
```

4.2.5 Omschrijving metric 5: Code complexiteit

In een stuk code komen vaak dezelfde variabelen, berekeningen en constructies voor. De mate van herhaling kan een maat zijn voor hoe complex het programma is. Een grote complexiteit zou kunnen duiden op een groot chipoppervlakte of het moeilijker kunnen halen van versnelling. Deze manier van het bepalen van complexiteit staat beschreven in Kokel et al(2000).
Om tot deze complexiteit te komen wordt gekeken naar de autocorrelatie van een zogenaamde walk door de code. Dit komt neer op dat je kijkt in hoever

sommige aspecten van de code zichzelf herhalen. Naar welke aspecten gekeken wordt kan verschillen. We hebben naar 4 verschillende manieren gekeken en gekozen voor de 'char methode'. Er is voor deze methode gekozen omdat het relatief eenvoudig te implementeren is.
Uiteindelijk is de richtingcoefficient van de autocorrelatiefunctie een maat voor de complexiteit van de code.

4.3 Omschrijving Model

De metrics geven niet meteen een duidelijk inzicht hoe de code zich zal gaan gedragen in hardware. Een schatting van het aantal flipflops, lookup tables (LUT), slices en states geeft veel meer inzicht in de uiteindelijke hardware. Een slice is een blok logica dat in een keer te programmeren is. Een slice bevat LUT's, flipflops en andere logica.

De schattingen worden op de volgende manier bepaald. Van iedere metric zijn coefficienten bepaald. Deze coefficienten zijn verkregen bij Roel Meeuws, begeleider van dit project. Iedere metric wordt vermenigvuldigd met zijn coefficient. De som van deze vermenigvuldigingen is de uiteindelijke schatting. Voor iedere schatting zijn andere coefficienten.

Meer details over het model zijn te vinden in Meeuws (2007).

4.4 Omschrijving XML-bestand

De informatie die het programma oplevert moet overzichtelijk worden opgeslagen in een XML-bestand. XML is opmaaktaal om informatie overzichtelijk en hierarchies op te slaan. Dit kan worden uitgelezen door machines, maar is ook goed leesbaar voor mensen.
XML-bestanden uitlezen is relatief eenvoudig omdat er veel gratis parsers beschikbaar zijn om te gebruiken. Dit is omdat XML een open standaard is.
Om onze resultaten op een eenvoudige manier door te geven aan de rest van het project is er besloten een XML-bestand aan te maken met daarin de gevonden resultaten.

Het bestand is opgedeeld in de geanalyseerde functies. In elke functie zitten 5 groepen met metrics, te weten 'Basic blocks', 'Memory usage', 'Operators', 'Output Model' en 'Misc'. In deze groepen zitten alle gerelateerde metrics.

Bovenin het bestand staat een CRC-code (cyclic redundancy check). Dit is gedaan om in één oogopslag te kunnen zien of 2 bestanden identiek zijn. Bij het testen van allerhande aanpassingen kan zo makkelijk bekeken worden of de geleverde metrics identiek zijn. In de volgende paragraaf is hier meer over te vinden.

Een voorbeeld van het XML-bestand staat hieronder.

```
<file filename="myfile.c">
        <checksum>1304456934</checksum>
        <function name="function1">
                <group name="Basic blocks">
                        <metric name="BBavgExpPerBlock">12.25</metric>
                </group>
                <group name="Memory usage">
                </group>
        </function>
        <function name="function2">
        </function>
</file>
```

4.5 Omschrijving CRC check

Om in één oogopslag te zien of twee XML-bestanden identiek zijn, gebruiken we een checksum. In plaats van alle waardes een voor een te controleren op veranderingen, kan met de checksum in een keer gezien worden of het bestand verschilt van de andere. Gebruikers van het XML-bestand en programma's die het XML-bestand gebruiken kunnen hierdoor snel controleren of het juiste bestand wordt gebruikt.

De checksum wordt als volgt berekend. CRC deelt de binaire string door een vastgelegd getal. CRC blijft dit doen tot verder delen niet meer mogelijk is. De rest is het checksum getal. Een wiskundige beschrijving van dit proces is (de binaire representatie van de file) mod (vastgelegd getal) = CRC. In Van Mieghem (2006:55) is meer te lezen over de werking van CRC checksums.

5 Implementatie profiling

5.1 Inleiding

In dit hoofstuk wordt ingegaan op de implementatie van de profiling. Voor de implementatie wordt gebruik gemaakt van een parser. Zo'n parser ontleedt de c-code (net als in normale taal). Door de elsa parser aan te passen zijn er verschillende metrics geimplementeerd.

In paragraaf 2 wordt ingegaan op de werking van de elsa parser. In paragraaf 3 wordt uitgelegd hoe wij de verschillende metrics hebben geimplementeerd in de elsa parser. In paragraaf 4 is de implementatie van het model te vinden. In paragraaf 5 staat de implementatie van de XML–output beschreven.

5.2 Omschrijving elsa parser

De Elsa parser is een zogenaamde front-end van een compiler. Het is een programma waarin een c-bestand wordt ontleed tot een Abstract Syntax Tree. Deze boom representeert het c-programma. Het c-bestand hoeft dus niet meer letter voor letter te worden ontleed, want dat heeft de parser al gedaan. Meer informatie over de elsa parser is te vinden op de site Elkhound en Elsa.

De AST kan eenvoudig worden doorlopen. Iedere soort node heeft een eigen visit() functie. Deze wordt aangeroepen zodra de node wordt doorlopen. Zo begint het doorlopen van een c-functie met de visitFunction() aanroep, gevolgt door enkele visitDeclaration() etc. Een volledig overzicht van alle type nodes en bijbehorende functies is te vinden in bijlage 1.
Elke visit() functie krijgt een object mee. Zo krijgt de visitDeclaration() een Declaration* object mee, en een visitExpression een Expression* object. Uit deze objecten is de meeste nuttige informatie te verkrijgen.

Meer informatie over parsers is te vinden in Grune et al.(2000:H2).

5.3 Implementatie metrics

De bovenstaande elsa parser is aangepast om verschillende metrics van c-code te bepalen. In de volgende paragrafen wordt besproken hoe de verschillende metrics zijn geimplementeerd in de elsa parser. De code van ieder programma is te vinden op de bijgevoegde cd.

Elke klasse die een metric implementeert erft alle visit() functies over van bepaalde superklassen uit Elsa. Zo hoeven we ons enkel druk te maken over wat er moet gebeuren als node X wordt bezocht, en niet over hoe die node bezocht wordt.
Alle gevonden metrics worden opgeslagen in een metricbag, zodat ze aan het einde van het programma uitgelezen kunnen worden.

5.3.1 Implementatie metric 1: Arithmische operaties

Arithmische operaties kunnen alleen optreden binnen expressies. Van de expressies hoeft dus alleen te worden bekeken wat voor soort expressie het is. Als zo'n expressie een arithmische operatie is, wordt de teller voor het type arithmische operatie verhoogd.

Het programma vindt dus een expressie en kijkt of het bijvoorbeeld een vermenigvuldiging is. Als het een vermenigvuldiging is, wordt de vermenigvuldigingsteller met 1 verhoogd. Deze vermenigvuldingsteller wordt vervolgens in een metricsbag gestopt. Dit wordt zo voor ieder type arithmische operatie gedaan.

Sommige arithmische operaties worden bij elkaar geteld. Dit kan als ze in hardware een soorgelijke implementatie hebben. De implementatie van deze metric is verder zo triviaal dat er geen ontwerpkeuzes gemaakt moesten worden.

5.3.2 Implementatie metric 2: Bit-breedte van een functie

Bit-breedte van een functie representeert het totaal aantal bits dat een functie als geheugen nodig heeft. In C worden variabelen altijd eerst gedeclareerd om ze te kunnen gebruiken. Van deze declaraties is met behulp van de reprSize() functie het aantal bits op te vragen.

Er wordt niet gekeken naar arrays en pointers. Ook hoeven members niet mee te worden geteld. Members zijn subvariabelen van een Struct. Struct is een datatype dat meerdere variabelen bevat. Als ergens Struct wordt aangeroepen, wordt het totaal aantal bits van de Struct meegeteld bij de bit-breedte. Er zou dubbel worden geteld als ook de membervariabelen worden meegeteld. Naast de bit-breedte wordt ook het aantal variabelen geteld.

De implementatie van deze metric heeft de nodige ontwikkeling gehad. Er zijn een drietal fases te onderscheiden.
Allereerst was het ons niet gelukt om uit het meegekregen visit() object de goede reprSize() functie te vinden, en zodoende er achter komen wat voor variabele er gedeclareerd werd. Om alsnog aan de goede gegevens te komen hadden we

besloten de debugPrint() te gebruiken. Hiermee is het mogelijk om een enorme tekst uit te printen, waar ook de nodige informatie in staat. Met behulp van wat string functies konden we 'normale' declaraties als 'int a' goed afhandelen. Helaas werd het met structures dusdanig ingewikkeld dat deze methode moest worden vervangen.
Hierna vonden we een functie waarmee we het type van de declaratie konden opvragen. Hiermee werd de code al een stukje makkelijker maar we moesten nog steeds de grote van elk type bijhouden. Voor normale declaraties is dat niet zo lastig, voor structures wel. Deze methode werkte uiteindelijk helemaal goed, maar was niet heel netjes.
Met de reprSize() functie kan in 1 keer de grote van de declaratie worden opgevraagd. Hiermee werd de code en heel makkelijke en netjes.

5.3.3 Implementatie metric 3: Bit-breedte van de input

Bit-breedte van de input van een functie representeert het totaal aantal bits van de variabelen die worden meegegeven een functieaanroep. Net als bij de bit-breedte van een functie wordt gebruik gemaakt van de reprSize() functie.

De functie visitFunction() heeft in zijn childs (de nodes verder onderaan de tak) ook de inputvariabelen. De inputvariabelen doen zich dus pas voor nadat visitFunction() wordt aangeroepen. De input eindigt bij de eerste accolade ("{") die zich voordoet. Een accolade is in dit geval een Compound-statement. Er hoeft dus niet verder naar variabelen te worden gekeken als zich zo'n compound-statement voordoet. Binnen deze grenzen wordt de bit-breedte en het aantal variabelen bepaald. Vervolgens worden deze ook in de metricsbag gedaan. Omdat de implementatie hiervan sterk lijkt op de vorige metric, hebben we geen ontwerp keuzes hoeven maken.

5.3.4 Implementatie metric 4: Basic blocks

Basic blocks zijn stukken code die achter elkaar kunnen worden uitgevoerd zonder jumps en daardoor in aanmerking komen voor paralellisatie in herconfigureerbare hardware. Na zo'n jump wordt opnieuw begonnen met het tellen van het daarop volgende blok. Zo'n basic block wordt onder twee voorwaarden beindigd:
Een compound-statement die wordt afgesloten ("}")
Een conditional optreedt.
Er wordt dus bijghouden wanneer een postVisitStatement optreedt van het type compoundstatement. Ook wordt bijgehouden wanneer een visitCondition optreedt. In deze twee gevallen wordt gestopt met het tellen van het aantal expressies en statement. De voorlopige maxima worden opgeslagen.

In het geval van een visitCondtion wordt pas weer begonnen met het tellen van statements en expressies als een postVisitcondtion optreedt. Dit om het aantal expressies en statements binnen een condition niet te tellen. Ook wordt het aantal expressies binnen een statement bijgehouden.

De volgende outputparameters worden op deze manier bepaald:

- Totaal aantal statements binnen een functie.
- Totaal aantal basic blocks binnen een functie
- Maximale aantal expressies binnen een basic block
- Maximale aantal expressies binnen een statement
- Maximale aantal statements binnen een basic block
- Gemiddeld aantal expressies per basic block
- Gemiddeld aantal statements per basic block
- Gemiddeld aantal expressies per statement

Deze parameters worden in de metricsbag opgeslagen.

5.3.4 Implementatie metric 5: Code complexiteit

Om een analyse te kunnen maken van de complexiteit van de code, hebben we allereerst de code zelf nodig. Omdat Elsa ons alleen maar kan voorzien van de AST, lezen we zelf de tekst uit het code bestand. De code wordt vervolgens ondaan van spaties en omgezet in een bitstring van ascii codes.

Van deze bitstring wordt een walk gemaakt. Dit houdt in dat er een functie wordt gemaakt welke 1 punt toeneemt als er een '1' in de bitstring zit, en een punt daalt als er een '0' voorbij komt. Van deze functie wordt vervolgens de autocorrelatie uitgerekend.

Het uitrekenen van de autocorrelatie gebeurt met de 'long range power law correlation' methode zoals beschreven in Kokel et al(2000) . Dit is een variatie op een standaard autocorrelatie functie. Er wordt hier voor elk teken uitgerekend in hoever hij en zijn omgeving op de rest van de code lijkt. De resulterende functie heeft intressante eigenschappen. Geplot op een dubbel logaritmische schaal verschijnt een vrijwel rechte lijn. Door middel van een linear square fit model wordt er gezocht naar de beste richtingscoefficient voor een eerste orde benardering.

Een stuk tekst zonder enige structuur (random getallen bijvoorbeeld) levert een richtingscoefficient van ongeveer 0.5. Elke correlatie in de tekst maakt dit getal wat groter. Zichzelf sterk herhalende code (blowfish algoritme bijvoorbeeld) komt uit op 0.85. De gevonden waarde wordt in de metricsbag opgeslagen.

Deze metric was wat lastiger om te implementeren. Allereerst moesten we ons verdiepen in de theorie, wat door onze ervaring met autocorrelatiefuncties geen groot probleem was. Het genoemde algoritme implementeren was lastiger.

Na implementatie van het 'long range correlation' algoritme bleken de resultaten nog niet gelijk bruikbaar te zijn. Door de aard van het algoritme werden de waarden naar het einde toe vanzelf lager. Hier hebben we zelf een schalingfactor overheen gezet. Op die manier zijn alle waarden met elkaar te vergelijken.

Een tweede moeilijkheid was het implementeren van het lineair square fit algoritme. We hadden gekozen voor de meest eenvoudige methode waarbij gebruik werd gemaakt van matrix vermenigvuldigingen, transponaties en inverteren. Na een grote zoektocht naar bruikbare functies bleek er zo snel geen

code beschikbaar die we makkelijk konden inpassen in het project. Dit heeft ons doen besluiten zelf de benodigde functies te maken.

5.4 Implementatie model

Om een schatting te maken van het aantal flipflops, states, slices en lookuptables, moeten eerst de coefficienten van de metrics worden uitgelezen. Deze coefficienten zijn aangeleverd in een space separated value bestand. Dit is een variant op comma separated value bestanden.

Op de eerste regel staan alle metrics, gescheiden door spaties. Deze metrics staan alfabetisch geordend. De regels daaronder staat het eerste kenmerk waarvan een schatting moet worden gemaakt, gevolgt door de coefficienten die bij het kenmerk horen. Voor de verdere kenmerken herhaalt zich dit op de daaropvolgende regels.

In het programma wordt de eerste regel niet gelezen. De naam van het kenmerk wordt toegevoegd aan een kopie van de MetricsBag. De eerste waarde bevat de offset. Vervolgens worden alle waardes van de metrics vermenigvuldigd met de coefficienten.

De som van deze vermenigvuldigingen en de offset wordt weggeschreven in het kopie MetricsBag. Deze MetricsBag wordt in een kopie van de MetricsMap geschreven. Het bovenstaande wordt herhaald voor alle functies die geanalyseerd worden. Het kopie van de MetricsMap overschrijft tenslotte de originele MetricsMap.

5.5 Implementatie XML-bestand

De XML output generator is gebaseerd op de bestaande printfunctie van de Elsa parser. Er zijn vier wijzigingen tenopzichte van de printfunctie.

Ten eerste wordt de output in verschillende groepen verdeeld. Deze groepen zijn basicblocks, memory usage, operators, output Model en misc (overige). De metrics worden uit de metricsbag gehaald. Aan de hand van hun naam worden ze ingedeeld in de verschillende groepen.

Ten tweede is de syntax van XML toegevoegd. In plaats van alleen de function naam printen, wordt de syntax <function name="function1"> gebruikt.

Ten derde wordt de output weggeschreven in een bestand. Dit wordt gedaan in de map waarin gewerkt wordt.

Ten vierde wordt een CRC code toegevoegd. De output van zonder bestandsnaam en CRC code wordt door de CRC generator gebruikt. De gegenereerde CRC code en bestandsnaam worden vervolgens toegevoegd. Meer over de implementatie van CRC bevindt zich in de volgende paragraaf.

5.6 Implementatie CRC check

Voor de implementatie van de CRC check hebben we gebruik gemaakt van een bestaande CRC generator. Er is gebruikt gemaakt van de 32 bit Cyclic Redundancy Check van Richard A. Ellingson. Deze kon zonder aanpassing worden gebruikt in het programma.

6 Resultaten

Alle metrics worden met succes bepaald. Verificatie is gedaan door de metrics te bepalen van 10 voorbeeldprogramma's. Deze voorbeeldprogramma's zijn te vinden op de bijgevoegde CD. Vervolgens zijn de resultaten met de hand gecontroleerd. Deze metrics kunnen nu verder gebruikt worden in een model voor schatting van grootte en versnelling.
De resultaten van het model zijn niet geverifieerd.

7 Conclusies en aanbevelingen

Conclusies

Het bepalen van verschillende eigenschappen van c-code voor omzetting naar een hardwarebeschrijving levert correcte resultaten op. Op redelijk eenvoudige wijze is het gelukt om eigenschappen van c-code te bepalen door de elsa parser aan te passen.

De implementatie van schattingen voor verschillende fysieke eigenschappen voor herconfigureerbare hardware levert een werkend programma op. Hierbij is een schatting gemaakt van het aantal LUT's, slices, flipflops en states, wanneer een programma wordt omgezet naar herconfigureerbare hardware. Deze schattingen worden gemaakt met behulp van de eigenschappen die we uit de c-code hebben gehaald. Verificatie van deze schattingen is niet uitgevoerd.

De resultaten van het programma worden opgeslagen in een XML-bestand.

Aanbevelingen

De voorwaarden waaraan code moet voldoen om te kunnen worden gebruikt door het programma zijn vrij streng. In c-code komen vaak pointers, arrays en functioncalls voor. Deze datatypes en functioncalls worden niet ondersteund door het programma. Er zou verder onderzoek moeten worden gedaan zodat dit wel ondersteund kan worden.

Op dit moment wordt alleen gekeken naar de code van een programma, om te kijken of het in aanmerking komt voor omzetting naar hardware. Het programma wordt als het ware ontleed. Hierbij wordt niet gekeken naar hoe vaak een functie daadwerkelijk gerund wordt. Door naar het dynamische gedrag van functies te kijken zijn meer gegevens te verzamelen die een nauwkeurigere schatting zouden opleveren. Dynamisch gedrag kan worden bepaald door het programma vaak te runnen en te meten hoevaak een functie wordt aangeroepen. Ook zou kunnen worden bekeken hoevaak bepaalde jumps worden gemaakt.

Tijdens het bepalen van het dynamische gedrag zou ook de grootte van de arrays kunnen worden bepaald. Hierdoor zou een schatting kunnen worden gemaakt van het aantal registers.

Bovendien dienen onze schattingen voor de fysieke implementatie op herconfigureerbare hardware nog geverifieerd te worden.

Referentielijst

Elkhound en Elsa. http://www.cs.berkeley.edu/~smcpeak/elkhound/ Geraadpleegd op 20 mei 2007

Ellingson, R.A. 2 bit Cyclic Redundancy Check Source Code for C++ http://www.createwindow.com/programming/crc32/index.htm Geraadpleegd op 15 juni 2007

Grune, D., Bal, H.E., Jacobs, C.J.H. & Langendoen, K.G.(2000) Modern Compiler Design. West Sussex: John Wiley & Sons, Ltd

Kokel, P., Brest, J., Zumer, V. (2000) Long range correlations in computer programs Slovenie

Meeuws, R.J. (2007) A Quantitative Model for Hardware/Software Partitioning Delft: MSc Thesis

Mieghem, P. Van (2006) Data Communication Networking Delft : Techne Press

Bijlage 1: Soorten nodes in elsa parser

```
visitType(Type *obj);postvisitType(Type *obj);

visitFunctionType_params(SObjList<Variable> &params);
postvisitFunctionType_params(SObjList<Variable> &params);
visitFunctionType_params_item(Variable *param);
postvisitFunctionType_params_item(Variable *param);

visitVariable(Variable *var);
postvisitVariable(Variable *var);

visitAtomicType(AtomicType *obj);
postvisitAtomicType(AtomicType *obj);

visitEnumType_Value(void /*EnumType::Value*/ *obj);
postvisitEnumType_Value(void /*EnumType::Value*/ *obj);

visitScope(Scope *obj);
postvisitScope(Scope *obj);

visitScope_variables(StringRefMap<Variable> &variables);
postvisitScope_variables(StringRefMap<Variable> &variables);
visitScope_variables_entry(StringRef name, Variable *var);
postvisitScope_variables_entry(StringRef name, Variable *var);

visitScope_typeTags(StringRefMap<Variable> &typeTags);
postvisitScope_typeTags(StringRefMap<Variable> &typeTags);
visitScope_typeTags_entry(StringRef name, Variable *var);
postvisitScope_typeTags_entry(StringRef name, Variable *var);

visitScope_templateParams(SObjList<Variable> &templateParams);
postvisitScope_templateParams(SObjList<Variable> &templateParams);
visitScope_templateParams_item(Variable *var);
postvisitScope_templateParams_item(Variable *var);

visitBaseClass(BaseClass *bc);
postvisitBaseClass(BaseClass *bc);

visitBaseClassSubobj(BaseClassSubobj *bc);
postvisitBaseClassSubobj(BaseClassSubobj *bc);

visitBaseClassSubobj_parents(SObjList<BaseClassSubobj> &parents);
postvisitBaseClassSubobj_parents(SObjList<BaseClassSubobj> &parents);
visitBaseClassSubobj_parents_item(BaseClassSubobj *parent);
postvisitBaseClassSubobj_parents_item(BaseClassSubobj *parent);

visitSTemplateArgument(STemplateArgument *obj);
postvisitSTemplateArgument(STemplateArgument *obj);

visitPseudoInstantiation_args(ObjList<STemplateArgument> &args);
postvisitPseudoInstantiation_args(ObjList<STemplateArgument> &args);
visitPseudoInstantiation_args_item(STemplateArgument *arg);
postvisitPseudoInstantiation_args_item(STemplateArgument *arg);

visitDependentQTypePQTArgsList(ObjList<STemplateArgument> &list);
postvisitDependentQTypePQTArgsList(ObjList<STemplateArgument> &list);
visitDependentQTypePQTArgsList_item(STemplateArgument *sta);
postvisitDependentQTypePQTArgsList_item(STemplateArgument *sta);

visitExpression(Expression *obj);
postvisitExpression(Expression *obj);
```

Bijlage 2: XML-file Sobel

```
<file filename="/home/praks661/profilingobj/examples/sobel_c_preprocessed.c">
	<checksum>3517171115</checksum>
	<function name="hw_sobel">
		<group name="Basic blocks">
			<metric name="BBavgExpPerBlock">12.25</metric>
			<metric name="BBavgExpPerStatement">7</metric>
			<metric name="BBavgStaPerBlock">1.75</metric>
			<metric name="BBcurMaxExpression">56</metric>
			<metric name="BBcurMaxStatement">9</metric>
			<metric name="BBmaxExpPerStatement">28</metric>
			<metric name="BBnumBlocks">12</metric>
			<metric name="BBtotalExpressions">147</metric>
			<metric name="BBtotalStatements">21</metric>
		</group>
		<group name="Memory usage">
			<metric name="VSArgMemCount">8</metric>
			<metric name="VSArgVarCount">2</metric>
			<metric name="VSAvgMemPerVar">4</metric>
			<metric name="VSMemCount">48</metric>
			<metric name="VSVarCount">12</metric>
		</group>
		<group name="Operators">
			<metric name="BINBitLogic">0</metric>
			<metric name="BINDivisions">0</metric>
			<metric name="BINLogic">3</metric>
			<metric name="BINMod">0</metric>
			<metric name="BINMultiplications">7</metric>
			<metric name="BINShift">0</metric>
			<metric name="PlusMinus">30</metric>
			<metric name="UNYBitNot">0</metric>
			<metric name="UNYNot">2</metric>
		</group>
		<group name="OutputModel">
			<metric name="RAWFlip-Flops">2279.13</metric>
			<metric name="RAWLUTs">158.547</metric>
			<metric name="RAWSlices">717.421</metric>
			<metric name="RAWStates">63.2663</metric>
		</group>
		<group name="Misc">
			<metric name="AICC">3.16838</metric>
			<metric name="Average Nesting Depth">5.2</metric>
			<metric name="Average Path Length">19.2273</metric>
			<metric name="Basili-Hutchens">60.5414</metric>
			<metric name="Cumulative Nesting Depth">26</metric>
			<metric name="Cyclomatic">10</metric>
			<metric name="Gong and Schmidt">10.8889</metric>
			<metric name="Loads">81</metric>
			<metric name="Loop Area">1.331</metric>
			<metric name="Maximum Nesting Depth">6</metric>
			<metric name="Maximum Path Length">30</metric>
			<metric name="NPATH">394</metric>
			<metric name="Oviedo DU pairs">108</metric>
			<metric name="Piwowarski">32</metric>
			<metric name="Prather's mu">230</metric>
			<metric name="Scope Number">30</metric>
			<metric name="Statements">42</metric>
			<metric name="Stores">25</metric>
			<metric name="Tai DU pairs">74</metric>
			<metric name="Variable Declarations">7</metric>
			<metric name="charMetric">0.559378</metric>
			<metric name="nOperands">113</metric>
			<metric name="nOperator">92</metric>
			<metric name="nUOperands">19</metric>
			<metric name="nUOperator">18</metric>
		</group>
	</function>
</file>
```