Profiling voor herprogrammeerbare systemen

Gábor Nándor Hummel

University of Groningen
Department of Computing Science

Begeleider: Ir. S. Achterop

Augustus 2004

Inhoudsopgave

Hoofdstuk 1

Inleiding

Systemen waarin een *general purpose processor* (GPP) zorgt voor de uitvoering van functies, zijn zeer flexibel. Er kunnen namelijk allerlei verschillende functies op worden uitgevoerd. Een probleem hierbij is echter de performance ervan, die soms alleen bij zeer hoge klokfrequenties aanvaardbaar is. Als functies te langzaam uitgevoerd worden op deze processoren, dan wordt al snel gekeken naar een hardware oplossing. Meestal wordt de functie dan gerealiseerd op een *Application Specific Integrated Circuit* (ASIC). Het voordeel hiervan is de efficiëntie ervan, het nadeel echter de inflexibiliteit. Met de intrede van de herprogrammeerbare hardware is deze inflexibiliteit verleden tijd. We kunnen nu de functionaliteit van de hardware wijzigen, zelfs tijdens de executie. Recent onderzoek heeft uitgewezen dat een combinatie van een GPP en een herprogrammeerbare hardware rekeneenheid potentieel kan leiden tot efficiënte oplossingen met hoge performance.
In deze scriptie doen we onderzoek naar de opsplitsing van de functionaliteit van applicaties in hardware en software. De hardware bestaat uit die taken die op de hardware rekeneenheid worden uitgevoerd, de software bevat de tasks die op de GPP worden uitgevoerd. Het belangrijkste deel van het onderzoek bestaat uit het vinden van een efficiënte methode om deze opsplitsing te maken. Hiervoor moet worden gezocht naar geschikte criteria waarmee we kunnen beoordelen of taken wel of niet gerealiseerd moeten worden op de hardware. Door middel van het gebruik van een profiler proberen we eigenschappen belangrijk voor het maken van de opsplitsing uit de applicatie te halen.
In hoofdstuk 2 beginnen met het geven van een overzicht van de context van deze scriptie, waarin de belangrijke concepten worden besproken. Daarna geven we in hoofdstuk 3 een beschrijving van het belangrijke analyse tool: de profiler, toegespitst op software applicaties. In hoofdstuk 4 kijken we hoe we via profiling kunnen komen tot applicaties die we efficiënt kunnen laten uitvoeren op herprogrammeerbare systemen. In hoofdstuk 5 laten we een methode zien waarin we met behulp van profiling kunnen bekijken wat de effecten zijn van bepaalde beslissingen die we in eerdere stappen van de methode hebben genomen. Ten slotte geven we in hoofdstuk 6 een conclusie en bespreken we nog een aantal zaken waar in de toekomst aan gewerkt kan worden.

Hoofdstuk 2

Codesign en herprogrammeerbare systemen

2.1 Inleiding

In dit hoofdstuk geven we een overzicht van een aantal onderwerpen die van belang zijn voor de verduidelijking van de context van dit onderzoek. Als eerste schetsen we een beeld van Hardware/software codesign, een ontwerp methode waarin software en hardware afhankelijk van elkaar worden ontworpen. Daarna beschrijven we een toepassing van codesign in herprogrammeerbare computersystemen. In dergelijke computersystemen kunnen *Field Programmable Gate Array's* (FPGA) een belangrijke rol spelen, daarom geven we een overzicht van een FPGA. Verder noemen we een aantal voorbeelden van bestaande gerelateerde projecten.

2.2 Hardware/software codesign

Tijdens de ontwikkeling van een applicatie werden traditioneel de wegen van hardware en software onafhankelijk van elkaar bewandeld. Tegenwoordig wordt echter steeds vaker een gecombineerde hardware/software aanpak gekozen: hardware/software codesign (HW/SW codesign). Bij HW/SW codesign worden hardware en software tegelijkertijd en afhankelijk van elkaar ontworpen. Codesign maakt het mogelijk de totale ontwerpruimte te verkennen om daarna een goede keuze te maken uit verschillende alternatieven. Het maken van de keuze om een component in hardware of in software te realiseren, wordt pas in een later stadium gedaan. Dit omdat er aan het begin van het ontwerpen van een dergelijk systeem nog niet duidelijk is welke componenten het beste in hardware of software kunnen worden gerealiseerd. Deze opdeling wordt *Hardware/Software partitioning* (in het vervolg van de tekst kortweg *partitioning*) genoemd. In het ideale geval hebben we een design-flow, waarin partitioning kan plaatsvinden op elk moment tijdens het ontwerp. In het begin van het ontwerp, kunnen we op basis van de resultaten van performance analyse van tijdschattingen, al een opdeling maken. Of in latere stappen als we de performance beter kunnen analyseren, bijvoorbeeld tijdens simulaties.
In de volgende secties beschrijven we twee verschillende codesign ontwerp methoden. De eerste methode (sectie 2.2.1) kan worden gezien als een conventionele methode,

waarbij partitioning al vrij vroeg in de design flow aanbod komt. Voor de tweede methode, *OCAPI-xl*[2], geldt dat partitioning overal kan plaatsvinden in het ontwerp. In sectie 2.2.3 evalueren we beide methoden.

2.2.1 Conventionele codesign methode

In figuur 2.1 wordt een overzicht gegeven van een conventionele codesign methode, met daarin de processen en activiteiten die doorgaans voorkomen. De verschillende stappen beschrijven we hierna kort, de volgorde die daarbij gebruikt is hoeft niet per definitie de beste te zijn. Maar het voldoet voor het geven van een duidelijk overzicht van een codesign ontwerp flow.

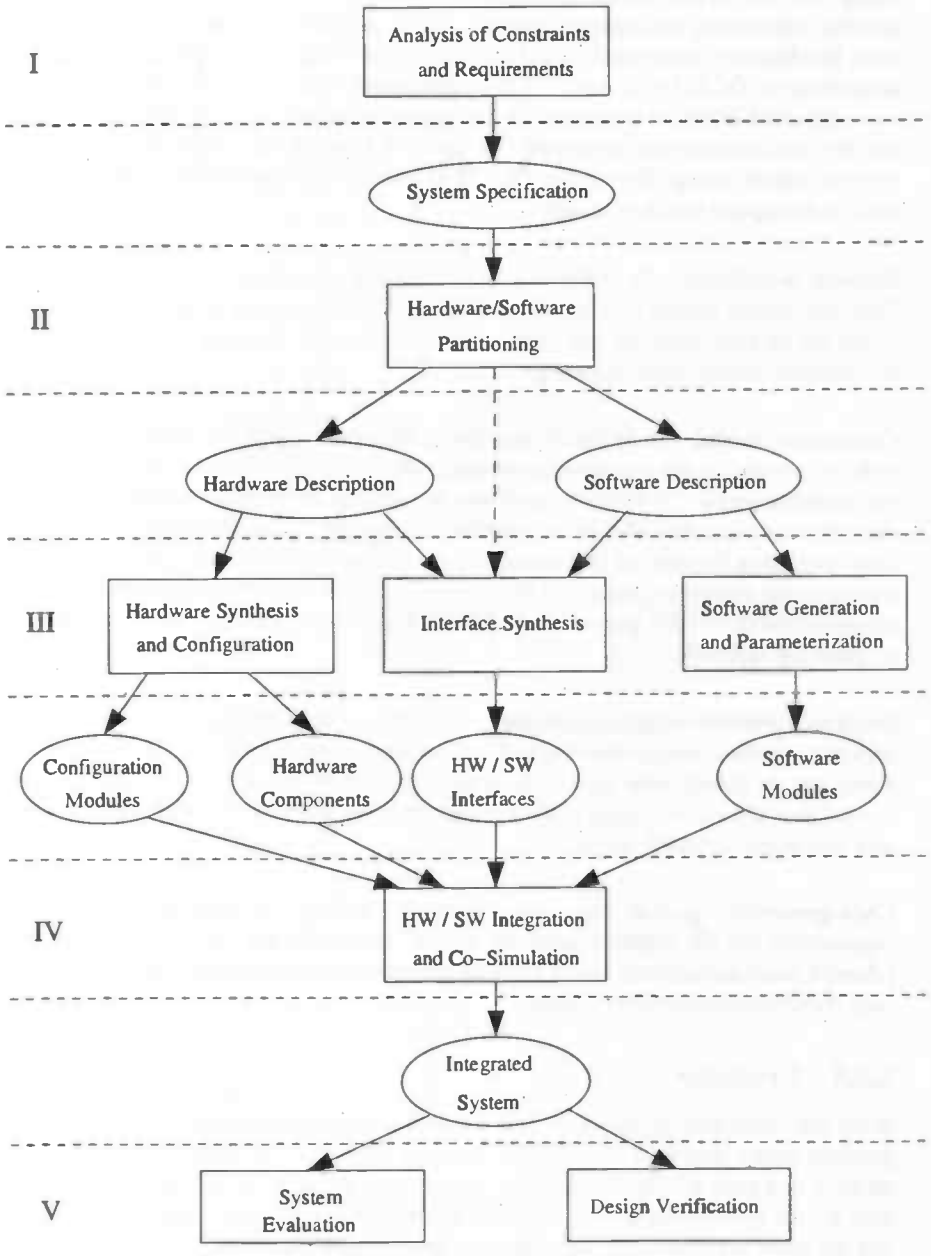
Analyse (I) In deze fase worden de eigenschappen van het systeem gedefinieerd. Deze eigenschappen worden bepaald op basis van de specificaties van de gebruikers en klanten. Voorbeelden van deze specificaties zijn: *performance*, kosten, betrouwbaarheid, *maintainability*, etc. Met behulp van de resultaten van de analyse wordt een geformaliseerde specificatie gemaakt. Vanuit deze specificatie kan de ontwerper algoritmes afleiden en ontwerpen. Via verschillende tools kunnen deze algoritmen worden gesimuleerd, waardoor ermee kan worden geëxperimenteerd. De simulatie is een eerste stap voor de ontwerper van het systeem om zijn ideëen te laten zien aan het management (en andere *stakeholders*). Daardoor krijgt de ontwerper de kans om via feedback zijn ontwerpbeslissingen te heroverwegen.

Hardware/Software Partitioning (II) Partitioning is een belangrijke stap in de codesign methodologie. Hier maakt de ontwerper, zelf of via een tool, een opdeling tussen componenten die worden geïmplementeerd in software en de componenten die worden gerealiseerd in hardware. Vaak worden voor het maken van een geschikte opdeling deterministische, statistische of *profiling* (zie hoofdstuk 3) technieken gebruikt.

Hardware Synthese, Interface Synthese en Software generatie (III) In deze stap wordt via de hardware beschrijving die opgeleverd is door de partitioning, een platform ontworpen waarop de hardware code gaat draaien. Via hardware synthese worden de hardware beschrijvingen gemapped naar fysieke hardware elementen; de functionaliteit wordt gerealiseerd in hardware. De software beschrijving wordt omgezet in software modules. Voor het communiceren en synchroniseren tussen hardware en software moet er een interface tussen beide worden ontworpen. Dat gebeurt ook in deze fase. De interface moet zorgen voor een efficiënte en snelle communicatie tussen de twee onderdelen (hardware en software) van het systeem.

Hardware/Software Integratie en Cosimulatie (IV) In deze fase worden hardware en software geïntegreerd tot een werkend prototype, die echt gerealiseerd is of draait op een simulator. Via cosimulatie kunnen we de onderdelen (hardware, software en interface) van het systeem testen.

Verificatie en Evaluatie (V) Als het geproduceerde systeem klaar is, moet worden gecontroleerd of het voldoet aan de originele systeem specificaties en of aan alle requirements is voldaan. Verificatie kan plaatsvinden door middel van een werkend prototype, maar doorgaans wordt een simulatie van het systeemontwerp gebruikt.



Figuur 2.1: Overzicht van een conventionele codesign methode

2.2.2 Het OCAPI-xl model

In [2] wordt een ontwerp model beschreven voor uniforme Hardware/Software modellering. Het **OCAPI-xl** design model, maakt het mogelijk om op elke plek tijdens het ontwerp partitioning beslissingen te nemen. Dit is mogelijk doordat het model een systeem beschrijving implementeert die architectuur onafhankelijk, implementeerbaar en aanpasbaar is. OCAPI-xl is een C^{++} class library voor het verfijnen en implementeren van embedded HW/SW systemen. In het model wordt een systeem beschreven door een set van concurerende processen. In figuur 2.2 wordt een schematisch overzicht gegeven van de design flow in het OCAPI-xl model. We beschrijven nu kort de verschillende stappen van deze design flow.

Systeem specificatie De methode gaat uit van een uitvoerbare systeem specificatie. Deze specificatie bestaat uit een set van C-functies (of een andere hogere level beschrijving) die de functionaliteit van tasks uitdrukken, zonder rekening te houden met de architectuur, concurrency en timing.

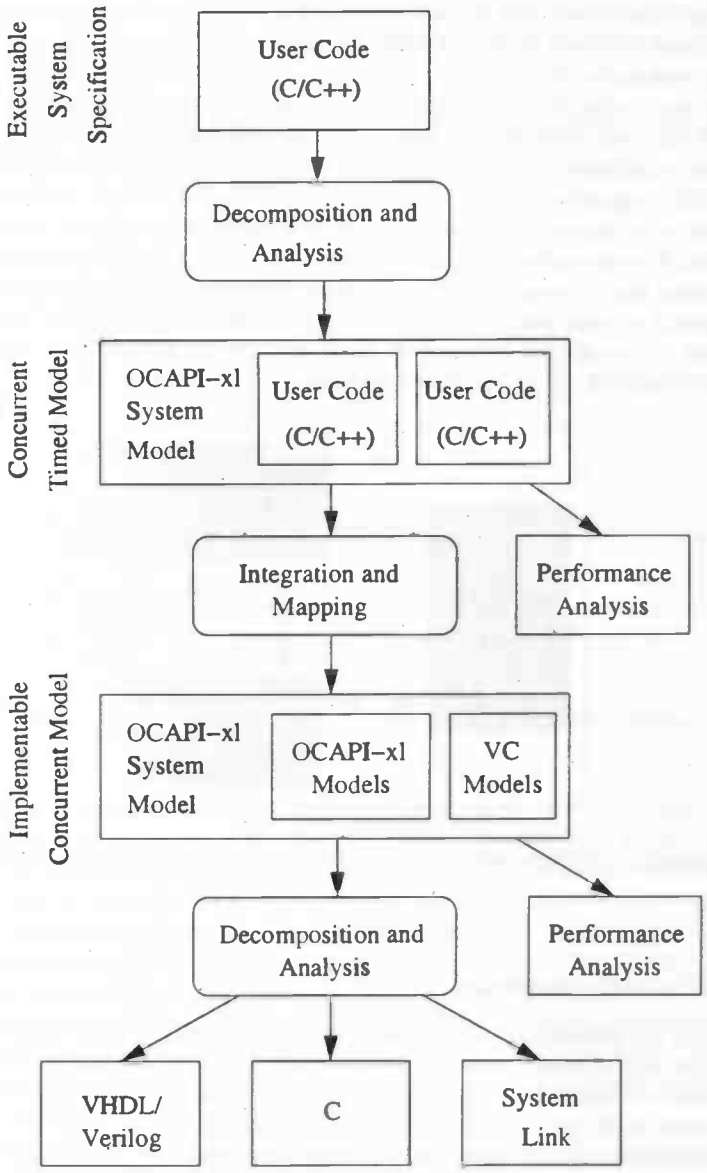
Concurrent model In de eerste stap van de methode, wordt het functionele model gedecomposeerd in een set van concurrerende OCAPI-xl processen die met elkaar kunnen communiceren. De functionaliteit van de tasks uit de systeem specificatie, wordt via *copy and paste* hergebruikt in zogenaamde *Foreign Language Interface*-objecten. Deze processen bestaan uit een aantal acties. Een actie is een stuk code, dat zonder interruptie en zonder interactie met hun omgeving kan draaien. De volgende communicatieprimitieven worden gebruikt in het OCAPI-xl model: *message*, semafoor (binair) en gedeelde variabele.

Implementeerbaar concurrent model Het concurrent model wordt verfijnt naar een implementeerbaar model, door middel van het integreren en mappen van de functionaliteit van de gekopiëerde *user code* naar OCAPI-xl objecten. Dit zijn objecten als *control flow statements*, zoals *loop*-objecten en *ifthen*-objecten en expressies op integers met gespecificeerde lengte.

Code generatie In deze laatste fase, wordt met behulp van code-generatoren code gegenereerd van het implementeerbaar model. Voor software wordt een C-compiler gebruikt, voor de hardware wordt een tool gebruikt voor *technology mapping* van RT-code (beschreven in VHDL/Verilog).

2.2.3 Evaluatie

In het conventionele model was te zien dat al in een vroeg stadium een opdeling wordt gemaakt tussen hardware en software. Dit heeft als nadeel dat omdat er in zo'n vroeg stadium nog geen efficiënte afsplitsing kan worden gemaakt. In het OCAPI-xl model zien we dat partitioning op verschillende momenten in het ontwerp kan plaatsvinden, wat als grote voordeel heeft dat naarmate de ontwikkeling gevorderd is er steeds efficiëntere afwegingen kunnen worden gemaakt, die ook nog gerealiseerd kunnen worden.

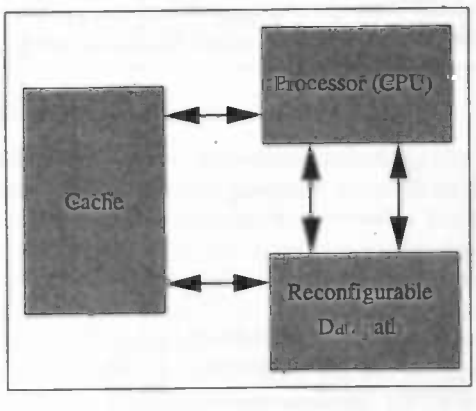


Figuur 2.2: De OCAPI-xl design flow

2.3 Herprogrammeerbare computersystemen

2.3.1 Introductie

Een toepassing van codesign kan worden gevonden bij herprogrammeerbare computersystemen. Deze systemen bestaan doorgaans uit een combinatie van een herprogrammeerbare hardware rekeneenheid en een general-purpose microprocessor (GPP), onderling verbonden door communicatiekanalen (zie figuur 2.3). Hierbij wordt de microprocessor gebruikt voor de niet rekenintensieve taken en systeem management taken. De hardware rekeneenheid wordt gebruikt voor het versnellen van een aantal uitgekozen rekenintensieve taken. Hierdoor krijgen we een architectuur die flexibel is (door zijn programmeerbaarheid) en tegelijkertijd in staat is om snel rekenintensieve taken uit te voeren. Een voorbeeld van zo'n herprogrammeerbare rekeneenheid is een *Field Programmable Gate Array* (FPGA, zie sectie 2.4). Bij herprogrammeerbare hardware kan *at compile-time* of *at runtime* (dynamisch) de functionaliteit die het implementeert worden gewijzigd, door het wijzigen van de configuratie. Bij het programmeren 'at compile-time', wordt de functionaliteit van de hardware vooraf (voor de executie) bepaald, en wijzigt daarna niet meer.



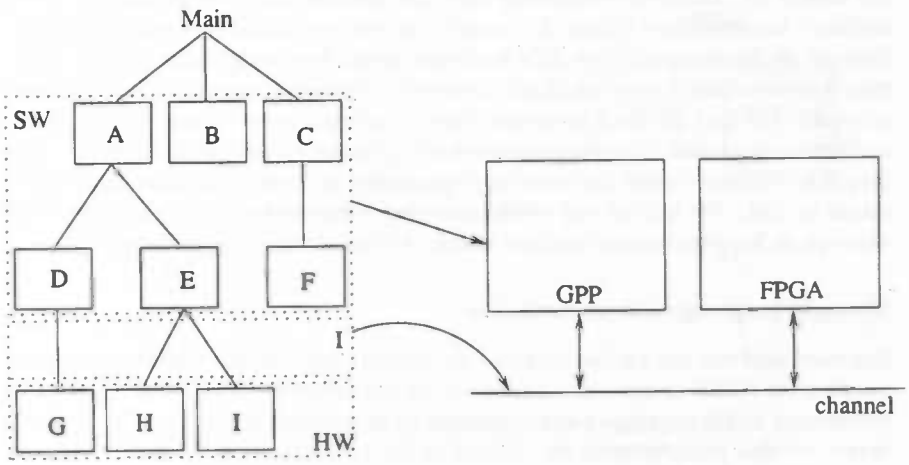
Figuur 2.3: Architectuur van een herprogrammeerbaar computersysteem

2.3.2 Virtuele Hardware

Het concept van **runtime herprogrammeren** is gebasseerd op virtuele hardware. We hebben maar een beperkte capaciteit op de rekeneenheid, die veel kleiner is dan de totale hoeveelheid ruimte die doorgaans nodig is voor het aantal componenten die we uit willen voeren op de hardware rekeneenheid. Doordat we tijdens de executie verschillende configuraties naar de rekeneenheid kunnen schrijven, lijkt het alsof we een veel grotere capaciteit hebben op het hardware device. Hierdoor kan dus ook een groter deel van het programma worden versneld. Om gebruik te maken van runtime herprogrammeren, worden van de taken die moeten worden uitgevoerd op de hardware een aantal configuraties gemaakt. Deze configuraties worden dan sequentieel naar de hardware geschreven en daarna uitgevoerd. Deze *mapping* van taken naar configuraties wordt **temporal partitioning** genoemd. In figuur 2.4 geven we een schets van een applicatie

die op een herprogrammeerbaar systeem draait (waarbij we uitgaan van een FPGA als hardware device). Er zijn drie blokken te onderscheiden:

- SW: Het deel van de applicatie dat op de GPP wordt uitgevoerd.
- I: De interface tussen GPP en FPGA, deze zorgt voor de communicatie tussen het host-systeem en het hardware device. Waarin zaken als herprogrammeren worden geregeld.
- HW: Het deel van de applicatie dat versneld moeten worden en dus wordt uitgevoerd op het hardware deel.



Figuur 2.4: Een applicatie op een herprogrammeerbaar systeem

Er kunnen gevallen voorkomen waarbij slechts een gedeelte van een configuratie hoeft te worden gewijzigd. In dat geval dient de hardware gedeeltelijk te worden geherprogrammeerd (*partial reconfiguration*). Het is duidelijk dat het programmeren van een gedeelte van de hardware minder tijd kost dat het telkens moeten herprogrammeren van het gehele hardware device. In een FPGA die partial reconfiguration ondersteunt, werkt het configuratie geheugen als een **RAM**-device. Via adressering kunnen selectieve doelen worden gespecificeerd die moeten worden aangepast met de configuratie data. Doordat de onaangetaste delen van de hardware blijven functioneren, vindt er een overlap plaats tussen executie op de hardware en herconfiguratie. Via partial-reconfiguration gaat dus minder tijd verloren met het herprogrammeren van het device.

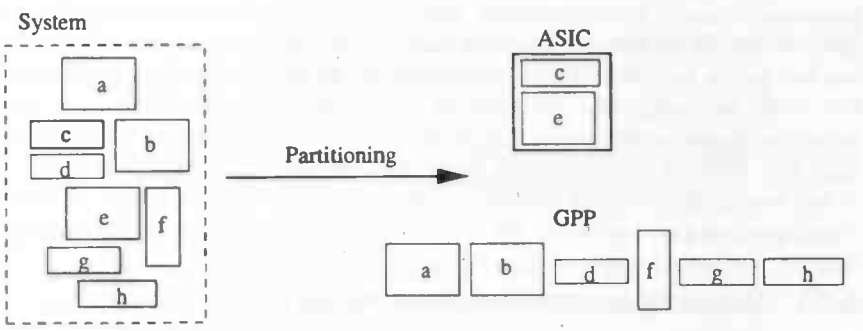
2.3.3 Eénmalig programmeerbaar vs. herprogrammeerbaar

Om een applicatie te versnellen kunnen we bepaalde componenten ervan in hardware realiseren. Hoe meer componenten we in hardware realiseren des te groter is de performance van het systeem. We moeten echter rekening houden met de kosten en grootte van hardware. Doordat een hardware oplossing duur is wordt doorgaans gekozen voor hardware met een beperkte omvang. Het gevolg daarvan is dat slechts een relatief klein deel van de gehele functionaliteit van de applicatie in hardware kan worden gerealiseerd. Vandaar dat we moeten gaan zoeken naar een efficiënte opsplitsing tussen

hardware en software.
Toen we nog geen herprogrammeerbare hardware (zoals FPGA's) hadden, speelde partitioning alleen een rol op het ruimtelijke domein. Een architectuur bestond uit een hardware deel met een bepaalde grootte, zoals een **Application Specific Integrated Circuit** (ASIC), hierdoor kon maar een bepaald deel van de functionaliteit op de hardware worden gerealiseerd. Een ander nadeel was dat als de functionaliteit van die hardware eenmaal was bepaald, het niet meer kon worden gewijzigd. In dergelijke systemen is het doel van partitioning om de totale executietijd te minimaliseren. Hiervoor moeten de verschillende functies uit het systeem worden geanalyseerd, waarna een keus moet worden gemaakt welke functionaliteit in hardware wordt gerealiseerd.
Door de opkomst van herprogrammeerbare hardware speelt de grootte van de hardware een andere rol. Herprogrammeerbare hardware betekent dat de functionaliteit die de hardware implementeert tijdens de executie van een applicatie kan worden gewijzigd. Doordat de functionaliteit van zo'n hardware device kan veranderen, kunnen we nu meer functionaliteit (op verschillende momenten tijdens de executie) op de hardware uitvoeren. Het lijkt dus alsof de omvang van de hardware groter is dan het in werkelijk is. Partitioning speelt in herprogrammeerbare systemen dan ook een andere rol. Voor dergelijke systemen moet partitioning plaatsvinden op twee verschillende aspecten: ruimte en tijd. We hebben nog steeds maar een beperkte hardware capaciteit, maar vanwege de herprogrammeerbaarheid kunnen we meer functionaliteit erop kwijt.

Eénmalig programmeerbare hardware

Een voorbeeld van een hardware device dat slechts eenmalig kan worden geprogrammeerd is de ASIC. In systemen bestaande uit een dergelijk device, moet tijdens de partitioning in het codesign proces duidelijk worden welke functies op het hardware device worden geïmplementeerd. Hierbij is het van belang dat de partitioning (zie figuur 2.5) een zo efficiënt mogelijke oplossing biedt. In figuur 2.5 zien we aan de linkerkant een verzameling van functies van een applicatie. Door middel van partitioning worden bepaalde functies uit die verzameling gefilterd. Hierbij dient rekening gehouden te worden met de omvang van het hardware device, hierdoor kunnen slechts enkele functies op het device worden geïmplementeerd. De partitioner heeft een aan-



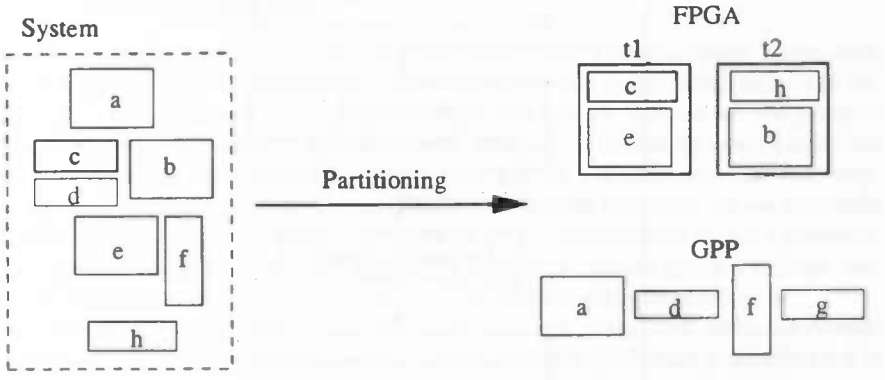
Figuur 2.5: Partitioning bij niet-herprogrammeerbare systemen

tal functies gekozen die het beste in hardware kunnen worden uitgevoerd, echter is de hardware te klein om ze er allemaal op te realiseren. In het voorbeeld heeft de partitioner functies *c* en *e* uitgekozen. De overige functies blijven op de GPP uitgevoerd

worden.

Herprogrammeerbare hardware

In systemen bestaande uit herprogrammeerbare hardware, lijkt het alsof we meer hardware hebben dan werkelijk in het systeem zit. Het hardware device kan immers opnieuw worden geprogrammeerd, en dus andere functionaliteit uitvoeren. Op verschillende tijdstippen tijdens de executie kunnen delen van de huidige configuratie van de hardware worden gewijzigd. Doordat deze configuraties (en dus ook de opsplitsing tussen hardware en software) in de tijd veranderen, wordt dit ook wel *temporal partitioning* genoemd. In figuur 2.6 zien we hetzelfde voorbeeld als in de vorige sectie.



Figuur 2.6: Partitioning bij herprogrammeerbare systemen

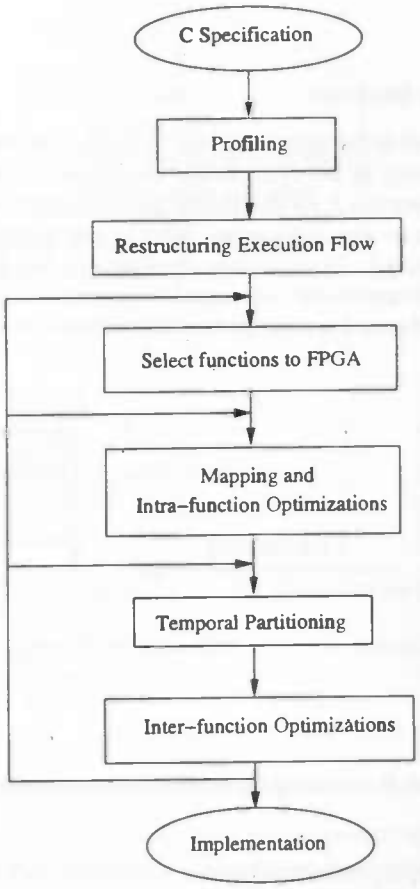
Doordat we kunnen herprogrammeren kunnen we nu meer functies in hardware realiseren, de partitioner kiest hier functies *c* en *e* op configuratie *t1* en functies *b* en *h* op configuratie *t2*.

2.3.4 Ontwerp en optimalisatie van dynamisch herprogrammeerbare embedded systemen

In [3] wordt een ontwerp methode beschreven voor dynamisch herprogrammeerbare systemen, bestaande uit een architectuur met een sterk gekoppelde processor en FPGA. Waarbij de FPGA wordt gebruikt als herprogrammeerbare hardware versneller en de processor de andere taken voor zijn rekening neemt, waaronder het programmeren van de FPGA. De ontwerp methode is gebasseerd op de ontwerptaal C. Er is een software ontwerptaal gekozen om te helpen de ontwikkelinskosten te reduceren. Bovendien wordt doorgaans slechts een klein deel van de code later geïmplementeerd op de FPGA, dus hoeft er weinig C code te worden herschreven. De ontwerpflow die is gebruikt is te zien in figuur 2.7, hieronder beschrijven we de verschillende stappen uit dit proces.

1. Profiling
   Om de functies te identificeren die een grote bijdrage hebben in de totale executie van de applicatie wordt de applicatie geprofiled.

2. Herstructurering van de executie flow
   Doordat het herprogrammeren tijd kost, is het niet efficiënt om een configuratie



Figuur 2.7: Ontwerp flow van een dynamisch herprogrammeerbaar systeem

maar een korte tijd te gebruiken. Daardoor wordt geprobeerd de executie flow zo aan te passen dat configuraties langer worden gebruikt.

3. Kiezen van functies voor de FPGA
   Hier gaat het om de hardware/software partitioning, waarna de uitgekozen functies voor het hardware device worden vertaald naar VHDL code.

4. Intra-function optimalisatie
   Hierin komen zaken als: *Specifying Wordlength* en *Single Instruction Multiple Data* (SIMD) aan bod.

5. Temporal Partitioning
   Hier worden de functies die worden gemapped naar hardware verdeeld in configuraties die sequentieel naar de FPGA worden geschreven en daarna uitgevoerd.

6. Inter-function optimalisatie
   Hier wordt gekeken naar optimalisatie van functies die op dezelfde configuratie aanwezig zijn. Voorbeelden van optimalisaties zijn:

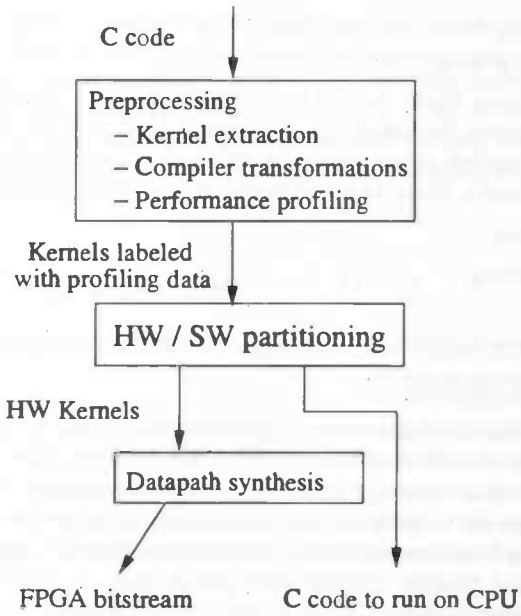
- *Localizing Memory Access*, hierbij wordt memory access verplaatst van het systeem geheugen naar het interne geheugen van de FPGA.
- *Code Reuse*, hierin worden verschillende software functies vergeleken op stukken code die gelijk zijn. Deze stukken kunnen dan zelfstandig op de FPGA worden geïmplementeerd en worden aangeroepen door de betreffende functie. Hiermee wordt ruimte bespaard op de FPGA.
- *Pipelining*
- *Parallelizing*.

2.3.5 Hardware-Software Codesign van herprogrammeerbare embedded systemen

Bij het ontwerpen van een runtime herprogrammeerbaar systeem, speelt partitioning een belangrijke rol. In [4] wordt een hardware/software partitioning algoritme beschreven voor dynamisch herprogrammeerbare systemen die bestaan uit één *general-purpose* processor en een herprogrammeerbaar datapad. In het artikel wordt gesteld dat partitioning bij deze herprogrammeerbare systemen een 2-dimensionale aanpak vergt: op **tijd** en op **layout** (fysieke implementatie van de functionaliteit op verschillende plaatsen op de hardware). Hierbij ligt de focus bij het partitioneren op het tijdsaspect. Dit aspect is belangrijk omdat het herprogrammeerbare datapad op verschillende momenten tijdens de executie van de applicatie kan worden geherprogrammeerd.
Het partitioneringsalgoritme is een onderdeel van een *framework* genaamd *Nimble* (zie figuur 2.8). Het framework compileert automatisch applicaties gespecificeerd in de ontwerptaal C naar uitvoerbare programma's voor dynamische herprogrammeerbare systemen. In een preprocessing stap wordt gezocht naar de loops die de meeste winst op zouden leveren als ze worden gerealiseerd in hardware. Van deze loops (*kernels* genaamd) worden dan meerdere hardware georiënteerde compiler transformaties gemaakt. Deze versies van dezelfde functionaliteit verschillen van elkaar op basis van *omvang* en *delay*. Het is dan de taak van het partitioning algoritme om de totale executietijd van de applicatie te minimaliseren. Als input krijgt de partitioner een beschrijving van het doel architectuur en de set van kernels (waarbij elke kernel een software versie heeft en één of meerdere hardware versies) die uit de applicatie afgeleid zijn. De kernels zijn in de preprocessing stap gelabeld met profiling data. Het algoritme moet nu kiezen of een loop in hardware of in software moet worden gerealiseerd. Bij een keus voor hardware, moet ook worden beslist welke versie gebruikt gaat worden. Bij het partitionen worden een aantal belangrijke aandachtspunten in acht genomen:

- het algoritme moet effectief de kosten van het herprogrammeren bepalen
- in het partitioning proces moeten compiler optimalisaties geïntegreerd worden en ook moet de ontwerp ruimte goed worden geanalyseerd
- partitioning moeten worden begeleid door verschillende vormen van profiling informatie

Het algoritme probeert de totale executietijd te minimaliseren, hiervoor houdt het een kostenfunctie bij voor de totale executietijd van de applicatie. Deze kosten functie omvat de hardware en software executietijd, de vertragingen geïntroduceerd door de hardware entry en exit (met deze kosten wordt bedoeld, de kosten voor het kopiëren van variabelen van en naar het hardware device) en als laatste de hardware herconfiguratie



Figuur 2.8: Nimble Compiler: een overzicht

kosten.
Het algoritme kan worden samengevat in de volgende vier belangrijke stappen:

1. *Loop entry trace profiling*, hierbij wordt elke loop-entry tijdens de executie geregistreerd. Dit is nodig omdat elke keer dat er een nieuwe hardware loop uitgevoerd moet worden, eerst het hardware device moet worden geprogrammeerd. Voor het uitrekenen van de totale kosten voor het herprogrammeren is dus de preciese runtime sequence van alle hardware kandidaten nodig.

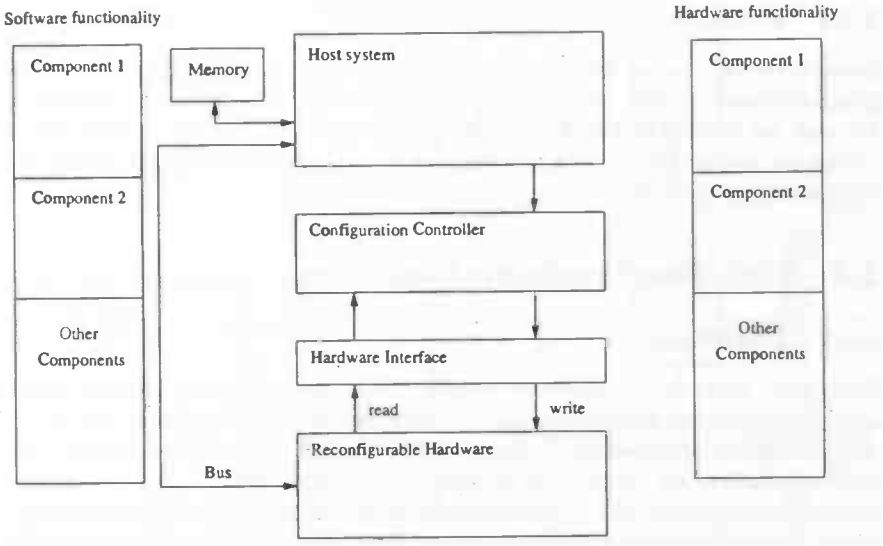
2. *Interesting loop detection*, hier wordt gezocht naar interessante loops. Dat wil zeggen, die loops die een grote bijdrage hebben in de totale executietijd. Hiervoor is een *interesting loop detector* (ILD) geïplementeerd, die de totale bijdrage van een loop aan de totale executietijd van de applicatie bijhoudt.

3. *Intra-loop detection*, hier worden de verschillende hardware versies geëvalueerd en bepaald welke hardware versie van een loop het beste past op de FPGA.

4. *Inter-loop selection*, deze fase bestaat uit twee stappen. In de eerste stap wordt met een clustering techniek de loops verdeeld in kleine clusters. Daarna wordt een optimale partitioning gezocht voor elke individuele loop-cluster.

2.3.6 Een herprogrammeerbare applicatie

Een voorbeeld van een runtime herprogrammeerbaar systeem is het systeem beschreven in [5]. In figuur 2.9, zien we een (vereenvoudigd) overzicht van dat systeem. Hierna



Figuur 2.9: Een runtime herprogrammeerbare applicatie

geven we een overzicht van de belangrijkste onderdelen van de *herprogrammeerbare applicatie*:

- **Host System**
  Hierop draait het software deel van de applicatie, waarbij de applicatie gebruik kan maken van de herprogrammeerbare hardware. Het *Host System* is direct verbonden met de hardware via de systeembus. Met behulp van *in/out* instructies, wordt gecommuniceerd met de hardware.

- **Configuration Controller**
  Deze controller zorgt voor de realisatie van nieuwe functionaliteit op de herprogrammeerbare hardware en wordt aangestuurd door het Host System.

- **Software Functionality**
  Bevat alle software componenten, voor elk software component is er een hardware component met dezelfde functionaliteit. Een software component draait in het geheugen van de Host System.

- **Hardware Functionality**
  Dit bevat alle hardware componenten. Deze componenten bestaan uit een controller, het circuit en de configuratie. Zoals gezegd is een hardware component de equivalent van een software component.

- **Reconfigurable Hardware**
  De herprogrammeerbare hardware, hierop wordt de functionaliteit van de hardware componenten gezet. Er kan slechts één task per keer uitgevoerd worden op de hardware.

2.3.7 Evaluatie

Door de combinatie van flexibiliteit (software) en efficiëntie (hardware) zijn herprogrammeerbare systemen zeer interessant. Het systeem is echter beperkt daar slechts één taak per keer op de FPGA kan worden uitgevoerd. Willen we optimaal gebruik maken van partieel herprogrammeren zouden er meerdere taken tegelijk moeten kunnen draaien op de hardware.

2.4 Field Programmable Gate Array

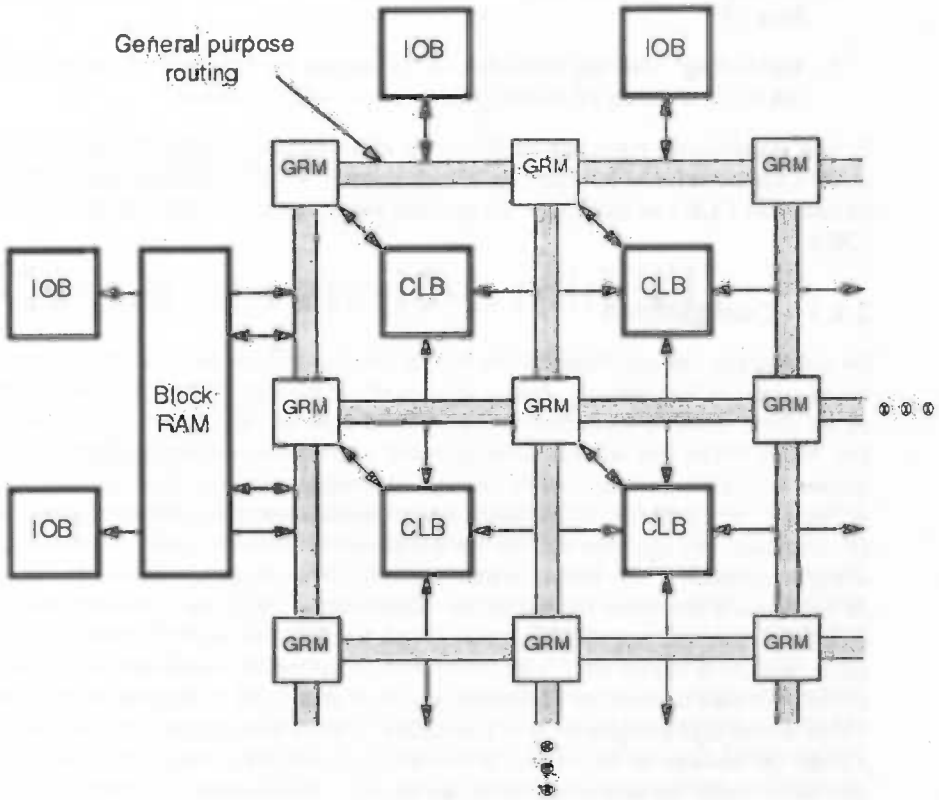
2.4.1 Introductie

Zoals in de vorige sectie naar voren kwam, bestaat een herprogrammeerbaar systeem onder andere uit een herprogrammeerbare rekeneenheid. Een voorbeeld van een herprogrammeerbare rekeneenheid is de *Field Programmable Gate Array*. De FPGA werd geïntroduceerd in het midden van de jaren tachtig. Sindsdien is de populariteit ervan enorm toegenomen en zijn er tegenwoordig verscheidene FPGA architecturen op de markt. Een onderscheid kunnen we maken in *single-context*, *multi-context* en *partially reconfigurable* FPGA's. De eerste FPGA's waren single-context, wat betekende dat de FPGA slechts één configuratie per keer kon bevatten. Multi-context FPGA's kunnen worden gezien als een gemultiplexte set van single-context FPGA's. De nieuwste variant is de partially reconfigurable FPGA. Waarbij meerdere configuraties tegelijk op de FPGA aanwezig zijn en een ervan kan worden geherprogrammeerd terwijl de anderen gewoon blijven functioneren.

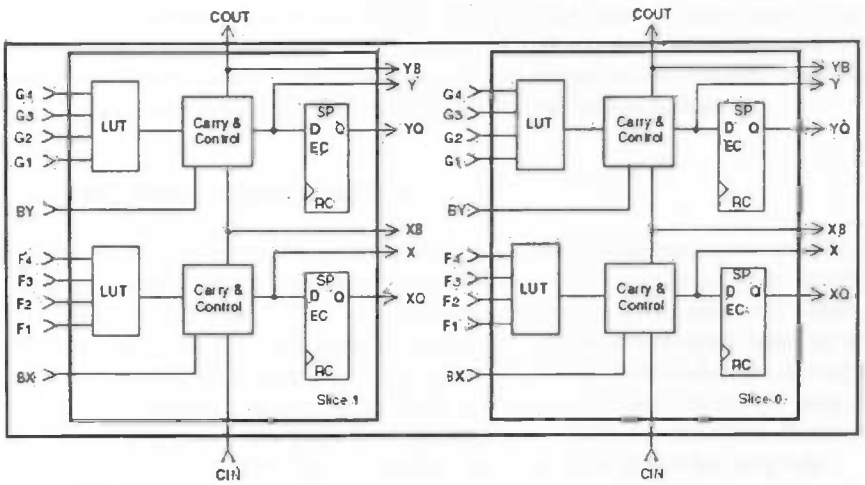
2.4.2 Architectuur

Voor het geven van een overzicht van een FPGA architectuur richten we ons hier op de *Virtex*-familie van Xilinx[1] (zie figuur 2.10 voor een blokdiagram van de Virtex). Deze Virtex FPGA kan oneindig vaak worden geherprogrammeerd (al naargelang de levensduur van het device zelf). Ook kan het device gedeeltelijk worden ge(her)programmeerd. Dit betekent dus dat een deel van de functionaliteit kan worden gewijzigd, terwijl andere delen van het device actief zijn. Hierdoor is de Virtex FPGA zeer geschikt om te worden gebruikt in dynamisch herprogrammeerbare systemen. Een Virtex device is opgebouwd uit twee belangrijke programmeerbare elementen: een array van *configurable logic blocks* (CLBs) die omgeven wordt door *programmable input/output blocks* (IOBs). De CLBs leveren de functionele elementen voor het implementeren van de funcionaliteit van het device. Voor de interface met externe devices zijn er de IOBs. De CLBs worden onderling verbonden door middel van *general routing matrix* (GRM). Deze matrices bestaan uit een array van routing switches, geplaatst op de kruisingen van de horizontale en vertikale routing kanalen. Via deze routing switches is het mogelijk om elke CLB te verbinden met een andere CLB. Hoe meer CLB's met elkaar worden verbonden hoe geavanceerder de functionaliteit van de FPGA wordt. De belangrijkste bouwsteen in een CLB is de *logic cell* (LC), elke CLB heeft er vier van, georganiseerd in twee identieke slices (zie figuur 2.11). Een LC bestaat uit:

1. **Functiegenerators**: Elke LC bevat twee 4-input functie generators. De functie generators zijn geïmplementeerd als 4-input *lookup-tables* (LUT). Met een 4-input LUT, kan elke functie met 4 variabelen worden geïmplementeerd.



Figuur 2.10: Blokdiagram Virtex FPGA



Figuur 2.11: 2-slice Virtex CLB

2. **Geheugenelementen**: Voor het tijdelijk opslaan van waarden bevat de LC *Flip-flops* (FF).

3. **Verbinding**: Voor het verbinden van de outputs van een LUT of andere inputs van de CLB met de FF worden *Multiplexers* (MUX) gebruikt.

Er zijn verschillende types van de Virtex die zich van elkaar onderscheiden door het aantal CLB's, het aantal IO-lijnen en de kloksnelheid. De XCV50 is de kleinste Virtex en bevat 384 CLB's en 168 IOB's. De grootste Virtex bestaat uit 16224 CLB's en 1024 IOB's.

2.4.3 Configureren

De configuratie van een Virtex FPGA bestaat uit drie stappen: eerst wordt het configuratie geheugen leeggemaakt, daarna wordt configuratie data in het geheugen geladen en ten slotte wordt de functionaliteit geactiveerd door middel van een opstart proces. Een Virtex FPGA kan worden geconfigureerd door middel van de *configuration bitstream*, bestaande uit commando's en data. De configuratie data is georganiseerd als words met een 32-bit lengte, bestaande uit een *header* met daarin het doel-adres voor dit commando. Er zijn twee soorten van commando's, namelijk: lees-commando's en schrijf-commando's. De bitstream kan worden gelezen en geschreven door een van de configuratie interfaces van het device. Configuratie informatie kan alleen worden gelezen/geschreven (ook tijdens executie) van/naar de Virtex als in de huidige configuratie bepaald is dat dit mag. Een configuratie interface bestaat uit één of meerdere *device pins* (een elektrische verbinding van de Virtex). De configuratie bits in een Virtex device zijn georganiseerd in kolommen. Deze kolommen zijn opgebouwd uit *frames*, die bestaan uit $18 * (\#CLB\_rows + 2)$ configuratie bits (rechts aangevuld met nullen totdat het past in een 32-bit word). Een frame is de kleinst mogelijke eenheid waarmee er kan worden geschreven en gelezen naar het configuratie geheugen. Het uitvoeren van een configuratie commando, vindt plaats nadat het commando is gelezen/geschreven van/naar het correcte commando register.

Hoofdstuk 3

Profiler als analyse middel

3.1 Inleiding

In het vorige hoofdstuk hebben we een aantal onderwerpen uit het probleemdomein besproken. Daarin staat het herprogrammeerbare systeem centraal. Een systeem dat bestaat uit een software deel en een hardware deel, waarbij het hardware deel wordt gebruikt om een aantal zeer rekenintensieve componenten te versnellen. Het aantal componenten dat kan worden versneld hangt af van de meestal beperkte omvang van de hardware. Meestal is de functionaliteit die we op de hardware willen zetten vele malen groter dan wat er in werkelijkheid op de hardware past. Herprogrammeerbare hardware is hier natuurlijk een geschikte oplossing, omdat we niet beperkt zijn tot één enkele configuratie, maar gedurende de uitvoering van de applicatie de functionaliteit van de hardware kunnen wijzigen. Een uitdaging hierbij is om efficiënte configuraties te maken, die ervoor zorgen dat de totale executietijd van de applicatie wordt geminimaliseerd. Een belangrijke tool hierbij is de zogenaamde profiler, die ons zinvolle informatie kan geven over de applicatie op basis waarvan we ten eerste een opslitsing kunnen maken tussen hardware en software en ten tweede efficiënte hardware configuraties kunnen maken. Profilers worden al lang gebruikt voor het optimaliseren van software applicaties, vandaar dat we dit hoofdstuk beginnen met een beschrijving van die profilers. Daarna geven we een beschrijving van een aantal ontwikkelde profilers, waaronder de profiler *Gprof* [7] die we in ons onderzoek hebben gebruikt.

3.2 Software optimalisatie

Het doel van software profiling is om software applicaties te optimaliseren, waardoor de totale executietijd van de applicatie wordt verkleind. Dit wordt gedaan door tijdens de executie van een applicatie te zoeken naar performance bottlenecks en te kijken naar het gedrag van de applicatie; doet het programma wat we ervan verwachten. Bij de bottlenecks zou je kunnen denken aan bepaalde delen van een functie die veel rekentijd vergen van de processor. In de praktijk blijkt dat het grootste deel van de tijd maar in een klein deel van de functionaliteit wordt doorgebracht. Het is dus zinvol om die delen te vinden en te optimaliseren. Wat een profiler doet is globaal gezien het volgende:

- het houdt bij hoevaak functies, bepaalde blokken code of loops worden uitgevoerd tijdens een bepaalde executie (of een aantal verschillende executies) van het

programma.

- het houdt bij hoe lang een bepaalde functie tijdens het uitvoeren van een programma in gebruik is; de executie-tijd van een functie.
- bovendien brengt het de structuur van een applicatie in kaart.

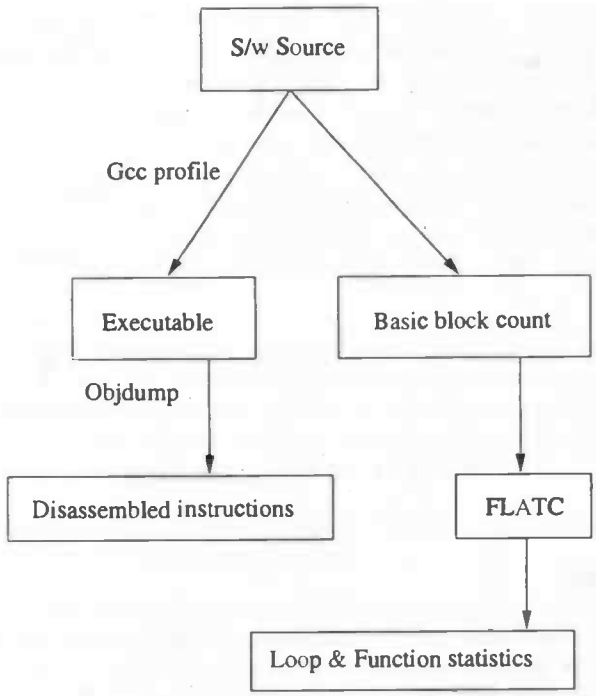
Je zou een profiler dus kunnen zien als een analyse middel op basis waarvan we bepaalde delen in het programma kunnen optimaliseren.

3.2.1 Toevoegen code

Veel van de profiling methoden maken gebruik van het extra toevoegen van code aan applicaties om bepaalde zaken te weten te komen. Zo kan door het toevoegen van een stukje code aan het begin van een functie bijhouden hoevaak die functie tijdens een executie wordt aangeroepen. Dit wordt bijgehouden in het geheugen of weggeschreven naar een file, na de executie kan die data dan worden geanalyseerd en worden bekeken hoevaak functies zijn aangeroepen.
Ook het berekenen van de executietijd van functies kan door middel van het toevoegen van extra code aan de applicatie. Niet moeilijk is in te zien dat door het bijhouden van de tijd aan het begin van de uitvoering van de functie en de tijd na uitvoering van de functie, je kan berekenen wat de executietijd bedraagt. Hier zijn echter ook andere methodes voor zoals *sampling*. In die methode wordt op bepaalde vaste tijdstippen de *program counter* bekeken om te kijken waar het programma zich bevindt. Ook hiermee kan dus de executietijd voor functies worden berekend.
Het toevoegen kan automatisch gebeuren, maar kan ook door de gebruiker zelf worden gedaan. Dit hangt af van de gebruikte profiling methode. Bij sommige methodes is het namelijk mogelijk op bepaalde interessante plekken van het te profilen programma annotaties toe te voegen die dan later door de compiler worden gevonden waarna de compiler extra analyse routines toevoegt. Bij andere profilers is er de mogelijkheid om bepaalde functies uit te kiezen die we willen profilen, of juist niet.

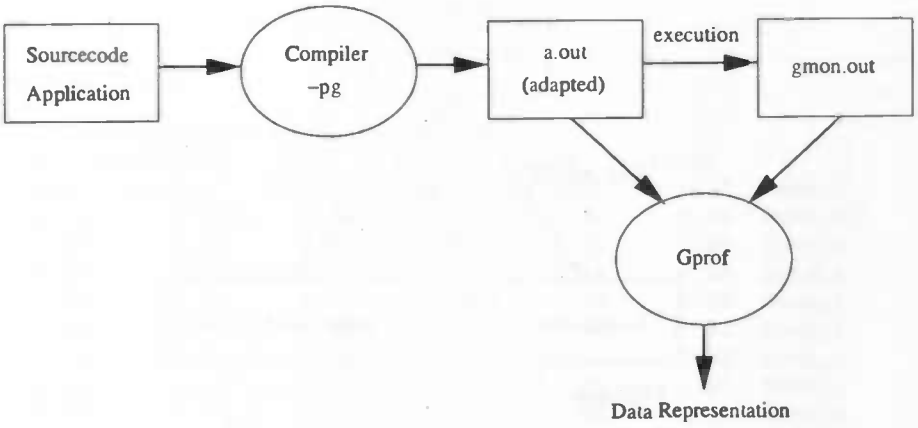
3.3 Verschillende profilers

Er zijn verschillende profilers ontwikkeld, waarbij de mate van detail een rol speelt. Kijken sommige profilers alleen naar functies, andere profilers houden zich bezig met een gedetailleerder niveau, zoals statements, blokken code of loops. Bovendien zijn een aantal profilers speciaal ontworpen voor een bepaalde processor familie.
ATOM[6] geeft de gebruiker de mogelijkheid om via een toolset analyse routines toe te voegen aan een programma op interessante plekken. Tijdens de uitvoering van het aangepaste programma, wordt informatie verzameld via die extra routines. Na de executie, wordt die data opgeslagen in een file. Via een aantal tools, kan die data dan verder worden geanalyseerd.
De Harvard Atom Like Tool (HALT)[8] levert een flexibele manier om analyse routines toe te voegen aan programma's geproduceerd door de SUIF compiler[9]. De gebruiker voegt annotaties toe op interessante plekken in het programma, waarna HALT op basis van de annotaties routines toevoegd. Via de verschillende analyse routines, levert HALT een aantal hardware simulators en houdt het statistieken bij voor *profile-driven* optimalisaties. HALT is gemaakt om te werken met MIPS en ALPHA processoren.



Figuur 3.1: Tool flow van FLATC

Frequent Loop Analysis Tools (FLAT)[10] is ontwikkeld voor de analyse van loop/function level informatie voor een breed scala aan platvormen. FLAT levert executietijden van programma's op basis van loops en functies. Voor het verkrijgen van de loop-profiles zijn twee methodes ontwikkeld. De eerste, FLATC, instrumenteert de compiler zodanig dat het de frequentie van een loop als uitvoer van het programma laat geven. Deze methode wordt gebruikt voor platvormen waarbij gcc wordt gebruikt als compiler. Bij de tweede methode, FLATSIM, wordt een simulator voor een instructie set gebruikt om hetzelfde op te leveren, maar dan voor platvormen als x86, MIPS en SPARC. In figuur 3.1 zien we de tool flow van FLATC. FLATC gebruikt de compiler gcc en de sources voor het verkrijgen van informatie over *basic blocks* (blokken van statements). Via de gedeassembleerde instructies van de executable wordt de informatie over de loopnamen en functieaanroepen verkregen. Met behulp van die data kan FLATC laten zien hoevaak bepaalde loops en functies zijn uitgevoerd en hoeveel executietijd daarbij gebruikt is tijdens het uitvoeren van de applicatie.
Een andere veelgebruikte profiler is Gprof, deze profiler levert informatie over de functies van de applicatie. Door middel van het toevoegen van extra profiling routines en het statistisch sampalen waar het programma zijn tijd doorbrengt, worden performance bottlenecks ontdekt.



Figuur 3.2: De werking van Gprof

3.4 Gprof

Zoals gezegd is Gprof (GNU Profiler) een voorbeeld van een profiler, doordat het veelgebruikt wordt en een *open source*-tool is het zeer geschikte tool om mee te werken. Deze software profiler geeft de programmeur meer inzicht in de werking van de applicatie, waardoor deze in staat is om de applicatie te optimaliseren. In de volgende secties wordt de werking van Gprof beschreven en eindigen we met een uitgewerkt voorbeeld van het gebruik ervan.

3.4.1 Werking Gprof

Via een monitoring routine worden twee zaken in het geheugen bijgehouden. Om te beginnen wordt, om voor elke functie bij te houden hoevaak het tijdens executie wordt aangeroepen, een tabel bijgehouden. In deze tabel (*hashtable*) wordt elke *arc* (de 'boog' tussen aanroepende routine en aangeroepen routine) die tijdens executie wordt tegengekomen geadministreerd, met het bijbehorende aantal keer dat deze is bereikt. Ook wordt een histogram bijgehouden waarin via sampling wordt gekeken waar de *pc* zich op bepaalde momenten tijdens de executie van het programma bevindt. Er wordt dus niet gemeten door de tijd te nemen die is verstreken tussen *routine entry* en *routine exit*. Dit omdat die methode ingewikkeld is op time-sharing systemen, aldus de ontwerpers van Gprof. De Gprof profiling methode omvat drie stappen, zie figuur 3.2. Eerst wordt via de compiler extra code om profiling mogelijk te maken aan het oorspronkelijke programma toegevoegd. Daarna wordt tijdens de executie van het programma profiling data verzameld, die aan het eind van het programma wordt opgeslagen in een file. Ten slotte wordt de profiling data via post-processing verwerkt. Hieronder beschrijven we de aparte stappen:

Compiler

Om een applicatie te kunnen profilen met Gprof worden er extra aanroepen aan het (te profilen) programma toegevoegd in het begin van elke routine. Dit kan door een extra optie (-pg) aan de compiler (voor C, fortran77 en Pascal) mee te geven. Deze

voegt aanroepen toe in de proloog van een routine naar een *monitoring routine*. De aanroep die wordt toegevoegd is *mcount* (of *_mcount*, of *__mcount*, afhankelijk van het Operating System en de compiler). Verder wordt er code toegevoegd die aan het begin van het programma zorgt voor initialisatie van de datastructuren die zorgen voor het verzamelen van de profiling data. Op Linux systemen houdt dat in dat een speciale profiling startup file *gcrt0.o* wordt gebruikt in plaats van de default *crt0.o*. Hierdoor wordt de functie *monstartup* uitgevoerd voordat *main* uit het programma wordt uitgevoerd. Tenslotte wordt extra code aan het eind van het programma toegevoegd, dat ervoor zorgt dat de data wordt opgeslagen in een file. Een aanroep van *mcleanup* zorgt hiervoor.

- *monstartup Subroutine*: Deze routine start en stopt executie profiling. Door een andere startup file aan te roepen, namelijk *gcrt0.o* in plaats van *crt0.o*, wordt *monstartup* aangeroepen voordat de *main* van een functie wordt uitgevoerd. Deze routine roept de *monitor*: routine aan om de data gebieden te initialiseren en profiling te starten.
- *monitor Subroutine*: Deze routine roept de *moncontrol* routine aan om te beginnen met het verzamelen van profiling data.
- *moncontrol Subroutine*: Start en stopt profiling nadat de initialisatie is voltooid. Deze routine roept de *profil* routine aan.
- *profil Subroutine*: Deze routine start en stopt *program address sampling* voor de executie profiling. Dit houdt in dat er schattingen worden bijgehouden voor de hoeveelheid tijd dat het programma in bepaalde delen van zijn adres ruimte doorbrengt. Hiervoor wordt de PC van het programma elke *clock tick* (CLK_TCK keer per seconde, gedefiniëerd in *time.h*), geïnspecteerd.

Executable

Tijdens de executie van de applicatie die we willen profilen wordt de profiling data in het geheugen verzameld. Als de applicatie klaar is, wordt die data naar de file *gmon.out* geschreven.

Post-processing

Door de data in *gmon.out* te analyseren met het programma Gprof, kan de dynamische *call graph* (voor deze executie van het programma) worden opgebouwd. Tijdens deze stap wordt voor elke functie zijn deel aan de totale executietijd berekend, de bijdrage van zijn kinderen hieraan en wordt voor elke functie uitgerekend hoevaak een functie door welke parent ervan is aangeroepen. Bovendien zorgt Gprof ervoor dat de informatie op een duidelijk manier wordt weergeven waardoor het makkelijk te interpreteren is.

3.4.2 Een voorbeeld van het gebruik van Gprof

Hieronder volgt de uitvoer van Gprof na het profilen van een eenvoudig voorbeeld programma bestaande uit een hoofdprogramma $Main$ en tien functies.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 34.75      1.06     1.06     2615     0.41     0.41  func_c
 30.82      2.01     0.94     3805     0.25     0.25  func_d
 18.69      2.58     0.57     1190     0.48     0.48  func_i
 15.08      3.04     0.46     1190     0.39     0.39  func_j
  0.98      3.07     0.03     2615     0.01     0.01  func_f
  0.00      3.07     0.00     2615     0.00     0.65  func_b
  0.00      3.07     0.00     2615     0.00     0.01  func_e
  0.00      3.07     0.00     2615     0.00     0.00  func_g
  0.00      3.07     0.00        5     0.00   348.31  func_a
  0.00      3.07     0.00        5     0.00   265.64  func_h
```

Flat profile

Uitleg van de tabel:

- time: Percentage van de totale tijd doorgebracht in de functie.
- cumulative seconds: Som van het aantal seconden doorgebracht in deze functie en zijn *children*.
- self seconds: Het aantal seconden doorgebracht in de functie.
- calls: Het aantal keren dat de functie werd aangeroepen.
- self ms/call: Milliseconden doorgebracht in de functie per aanroep.
- total ms/call: Milliseconden doorgebracht in de functie en zijn *children* per aanroep.
- name: Naam van de functie.

Als we de uitvoer bekijken zien we dat in functie $func\_c$ de meeste tijd wordt doorgebracht, namelijk bijna 35%. Ook functie $func\_d$ heeft een behoorlijk aandeel. namelijk bijna 31%.

Call graph

granularity: each sample hit covers 2 byte(s) for 0.33% of 3.07 seconds

index	% time	self	children	called	name
					<spontaneous>
[1]	100.0	0.00	3.07		main [1]
		0.00	1.74	5/5	func_a [2]
		0.00	1.33	5/5	func_h [4]
		0.00	1.74	5/5	main [1]
[2]	56.7	0.00	1.74	5	func_a [2]
		0.00	1.71	2615/2615	func_b [3]
		0.00	0.03	2615/2615	func_e [9]
		0.00	1.71	2615/2615	func_a [2]
[3]	55.8	0.00	1.71	2615	func_b [3]
		1.06	0.00	2615/2615	func_c [5]
		0.65	0.00	2615/3805	func_d [6]
		0.00	1.33	5/5	main [1]
[4]	43.3	0.00	1.33	5	func_h [4]
		0.57	0.00	1190/1190	func_i [7]
		0.46	0.00	1190/1190	func_j [8]
		0.29	0.00	1190/3805	func_d [6]
		1.06	0.00	2615/2615	func_b [3]
[5]	34.6	1.06	0.00	2615	func_c [5]
		0.29	0.00	1190/3805	func_h [4]
		0.65	0.00	2615/3805	func_b [3]
[6]	30.7	0.94	0.00	3805	func_d [6]
		0.57	0.00	1190/1190	func_h [4]
[7]	18.6	0.57	0.00	1190	func_i [7]
		0.46	0.00	1190/1190	func_h [4]
[8]	15.0	0.46	0.00	1190	func_j [8]
		0.00	0.03	2615/2615	func_a [2]
[9]	1.0	0.00	0.03	2615	func_e [9]
		0.03	0.00	2615/2615	func_f [10]
		0.00	0.00	2615/2615	func_g [11]
		0.03	0.00	2615/2615	func_e [9]
[10]	1.0	0.03	0.00	2615	func_f [10]
		0.00	0.00	2615/2615	func_e [9]
[11]	0.0	0.00	0.00	2615	func_g [11]

Index by function name

[2] func_a	[9] func_e	[7] func_i
[3] func_b	[10] func_f	[8] func_j
[5] func_c	[11] func_g	
[6] func_d	[4] func_h	

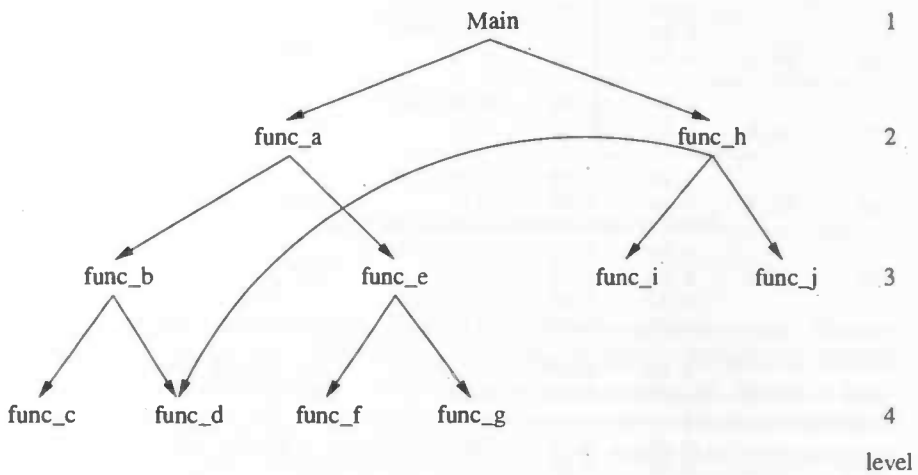
Call graph

De tabel bestaat uit drie delen, elk bestaande uit een aantal rijen. Elke rij met de index (links), geeft aan om welke functie het gaat. De rijen erboven beschrijven de functies die deze functie hebben aangeroepen (parents), die eronder beschrijven de functies die werden aangeroepen door deze functie (children).
Uitleg van de kolommen:

- index: Uniek nummer voor elk element in de tabel
- % time: Het percentage van de totale tijd doorgebracht in deze functie en zijn *children*.
- self: Totale tijd doorgebracht in deze functie.
- children: De tijd doorgegeven aan deze functie door zijn *children*.
- called: Het aantal aanroepen van deze functie.
- name: De naam van de functie, met daarachter de index ervan.

3.4.3 Data evaluatie

Via de data kunnen we nu een aantal dingen zeggen over de applicatie. Ten eerste zien we dat de functie $func\_c$ 34.75% van de totale executietijd in beslag neemt. Hiermee is $func\_c$ de functie waar de meeste tijd wordt doorgebracht.
Bovendien wordt in het Callgraph deel de structuur van de applicatie duidelijk. In figuur 3.3 zien we in grafische vorm dezelfde Callgraph data die we via Gprof van de applicatie hebben afgeleid.



Figuur 3.3: Grafische weergave van de Callgraph van het voorbeeld programma

In het plaatje zien we rechts het niveau van de functie, begint bovenaan bij het hoofdprogramma $Main$ op niveau 1. Daaronder op niveau 2 de functies die door $Main$ zijn aangeroepen (zijn kinderen), hetzelfde geldt voor de onderliggende niveaus.

Hoofdstuk 4

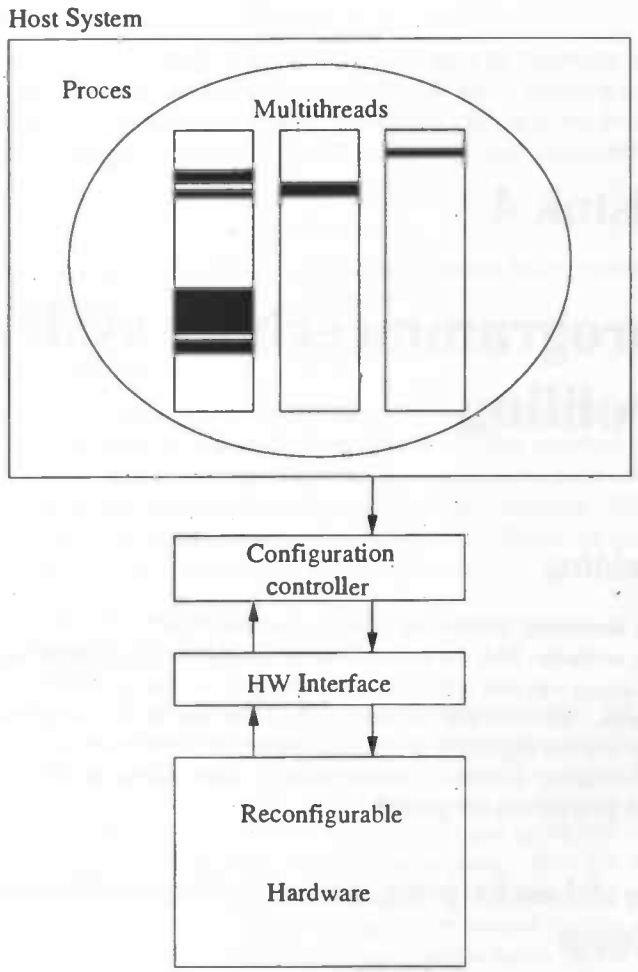
Herprogrammeerbare systemen en profiling

4.1 Inleiding

In het vorige hoofdstuk hebben we gekeken naar de profiler als analyse middel voor applicaties in software. Met die analyse van een applicatie wordt meer inzicht verkregen in de structuur van een applicatie en worden we in staat gesteld om de applicatie te optimaliseren. Het doel van dit onderzoek is om een profiler te gebruiken bij het optimaliseren van een applicatie op een herprogrammeerbaar systeem. Doordat we nu kijken naar een ander systeem, moeten we eerst gaan kijken wat dat voor gevolgen heeft voor het gebruik van een profiler.

4.2 Korte beschrijving van een herprogrammeerbaar systeem

Zoals in eerdere hoofdstukken al is geschetst bestaan herprogrammeerbare applicaties uit een hardware deel en een software deel. Het hardware deel is een extra toevoeging aan het systeem om bepaalde functionaliteit van het systeem te versnellen. Dit hardware deel is (her)programmeerbaar, wat wil zeggen dat de functionaliteit die het implementeert tijdens de executie van de applicatie kan worden gewijzigd. Als uitgangspunt wordt *de herprogrammeerbare applicatie* uit [5] gebruikt. Dat systeem gaat ervan uit dat er voor bepaalde componenten een software en een hardware variant aanwezig zijn. Tijdens de executie van een applicatie, wordt voordat een component wordt aangeroepen door de *configuration controller* gekeken of de hardware genoeg resources beschikbaar heeft om het component in hardware uit te voeren. Zo ja, dan wordt het component op de hardware gerealiseerd en uitgevoerd. Zijn er echter geen resources beschikbaar dan wordt een keus gemaakt of de software variant van de functie wordt uitgevoerd of dat er ruimte op de hardware wordt gemaakt. In figuur 4.1 zien we nogmaals het herprogrammeerbare systeem die we ook in hoofdstuk 2 hebben bekeken. Hierin zien we dat op het Host System een applicatie draait, in dit voorbeeld bestaat de applicatie uit uit meerdere threads. In de threads zijn met zwarte blokken aangegeven welke stukken code moeten worden versneld. Tijdens de executie van de applicatie wordt voordat een dergelijk blok wordt uitgevoerd door de configuration



Figuur 4.1: Overzicht herprogrammeerbaar systeem

controller gekeken of het mogelijk is om het blok in hardware uit te voeren. Het eenvoudigste zou zijn dat alles wat we in hardware willen realiseren ook gewoon allemaal op de hardware past. Dit is echter in de meeste gevallen onmogelijk, omdat de hardware maar een beperkte omvang heeft. Hierdoor moeten we de functionaliteit op de hardware gaan wisselen, wat natuurlijk precies het grote voordeel is bij herprogrammeerbare systemen.

4.2.1 Wisselen functionaliteit

Zoals gezegd heeft de hardware maar een beperkte ruimte. Echter vanwege de herprogrammeerbaarheid van de hardware kunnen we er toch meer functionaliteit op kwijt dan we ruimte hebben. Dit kan doordat we op verschillende momenten tijdens de looptijd van de applicatie, verschillende functionaliteit op de hardware kunnen laten draaien. Nadeel van deze werkwijze is echter dat er extra vertragende kosten een rol

kunnen gaan spelen, namelijk de configuratiekosten. Wat we met profiling willen doen is dan ook om data uit het systeem te halen waarmee we deze configuratiekosten kunnen minimaliseren en natuurlijk het programma zo snel mogelijk te laten worden. Hierbij ligt een belangrijke deel van deze taak bij de configuration controller.

4.2.2 Configuration Controller

De configuration controller is zoals gezegd verantwoordelijk voor het afhandelen van de hardware claims. Hierbij neemt de controller beslissingen die op basis van bepaalde informatie is verkregen. Dit kan informatie zijn die *at compiletime* is bepaald, of data die *at runtime* uit het systeem wordt gehaald. Bovendien zou via verschillende executies van simulaties van de applicatie data onttrokken kunnen worden (dus ook een soort at runtime). Het liefst zouden we vooraf, dus at compiletime al beslissingen maken. Zoals het maken van efficiënte groepen die tegelijk op de hardware aanwezig zijn. Dit is in sommige gevallen echter niet mogelijk. Als we niet op voorhand kunnen bepalen welke functies wanneer moeten worden geladen op de hardware, dan moet de configuration controller at runtime beslissen of een component op de hardware wordt uitgevoerd of niet. Hiervoor kunnen verschillende algoritmes worden gebruikt die op een efficiënte manier zorgen voor het wisselen van componenten op de hardware.
Tijdens de exectutie van een applicatie die op het herprogrammeerbare systeem draait, komen er momenten waarop bepaalde functionaliteit in hardware moet worden uitgevoerd. De configuration controller krijgt dan een aanvraag binnen en handelt die af. Bij dit afhandelen kunnen er een aantal verschillende situaties optreden afhankelijk van de huidige samenstelling van de hardware. We noemen de situaties en het resultaat ervan:

1. **Component is al aanwezig**: de functionaliteit die we op de hardware gerealiseerd willen zien is al aanwezig. De hardware hoeft dus niet opnieuw te worden geprogrammeerd. Is de functionaliteit actief, dan kan de controller beslissen om te wachten, of om toch de software variant uit te voeren. Deze beslissing is afhankelijk van de data die we via analyse hebben verkregen en wordt zodanig gemaakt dat de exectutie zo snel mogelijk is.

2. **Functie plaatsen**: de functionaliteit is nog niet aanwezig op de hardware en er is nog ruimte beschikbaar hiervoor. De controller moet dan dus de functionaliteit op de hardware realiseren.

3. **Bepaald component wisselen**: de functionaliteit is nog niet aanwezig en er is geen ruimte voor nog een functie op de hardware. We moeten dan een bepaald component van de hardware halen en de nieuwe functie erop zetten. Ook hierbij dient de controller een efficiënte beslissing te nemen die gebasseerd is op data uit de analyse.

4. **Gehele context wisselen**: ook in deze situatie is de functionaliteit nog niet aanwezig, de controller besluit echter om een hele context (dus de hele hardware wordt bijgewerkt) te wisselen in plaats van slechts één component.

4.2.3 Herprogrammeerbare hardware

Er zijn verschillende typen herprogrammeerbare hardware, die elk bepaalde effecten hebben op het proces in de ontwikkeling van herprogrammeerbare applicaties.

Single Context

We hebben slechts één configuratie die sequentiëel naar de hardware wordt weggeschreven. Hierbij moet dus de gehele hardware bij elke wissel opnieuw worden geprogrammeerd. Dit is niet efficiënt omdat soms slechts een klein deel van de functionaliteit hoeft worden gewijzigd terwijl er wel een hele herconfiguratie plaatsvindt.

Multi Context

Hierbij hebben we een aantal gemultiplexte single contexten in het systeem. Waardoor er snel kan worden gewisseld tussen configuraties.

Partially Reconfigurable

Hierbij kunnen we een bepaald deel van de functionaliteit die op de hardware is geconfigureerd wijzigen terwijl de andere delen actief blijven. Een voordeel hiervan is dat niet meer een gehele configuratie moet worden geprogrammeerd maar slechts een deel ervan, en dus de kosten van het herconfigureren worden verkleind. Hierdoor wordt deze vorm van herprogrammeren ook gezien als het meest efficiënt en gaan we ook uit van systeem bestaande uit een dergelijke hardware device.

4.3 Optimalisatie applicaties

Het optimaliseren van applicaties op herprogrammeerbare systemen kan globaal worden opgesplitst in twee delen: ten eerste willen we componenten selecteren die we op de hardware willen uitvoeren en ten tweede moeten we van die componenten efficiënte configuraties maken. Deze twee delen samen kunnen we partitioning noemen. Partitioning is namelijk de fase waarin voor onderdelen in een applicatie wordt gekozen waar zé worden uitgevoerd, op de hardware of in software. Voor het selecteren van geschikte functies die in hardware worden gerealiseerd, is het nodig om voldoende inzicht te krijgen in de applicatie. Zoals gezegd willen we een profiler gebruiken om de nodige informatie uit de, te optimaliseren, applicatie te halen. In het vorige hoofdstuk is al naar voren gekomen dat via (software-) profiling bottlenecks kunnen worden geïdentificeerd, deze lijken op het eerste gezicht uitermate geschikt voor realisatie in hardware. In theorie is het meestal zo dat wanneer we een component in hardware uitvoeren zijn executietijd wordt verlaagd in vergelijking met hetzelfde component in software op de GPP. Bij herprogrammeerbare systemen is dit minder triviaal dan het op het eerste gezicht lijkt. Het is namelijk zo dat het (her)configureren (het realiseren van nieuwe functionaliteit) van de hardware relatief veel tijd in beslag neemt. Bij het maken van een afweging of een functie wel of niet in hardware moeten we daar dus rekening mee houden.
We kunnen dus pas efficiënte afwegingen maken na een grondige analyse van de applicatie en de functies waaruit het is opgebouwd. Deze analyse kan worden gedaan met behulp van een profiler. In het vorige hoofdstuk hebben we reeds aandacht besteed aan een profiler voor software applicaties. We willen nu dat profiling die informatie oplevert die voor de partitioner van belang is om adequate configuraties van het systeem te maken. De optimalisatie vindt hier dus plaats door performance bottlenecks in hardware te realiseren. Het vinden van die bottlenecks gaan we doen door in de data van de profiler te zoeken naar belangrijke eigenschappen van de functies. Hierbij kijken we niet alleen naar de functies individueel, maar ook naar hun samenhang in de applicatie.

In de volgende secties proberen we een overzicht te geven van de eigenschappen die van belang zijn voor het partitionen en dus in kaart moeten worden gebracht alvorens we efficiënte afwegingen kunnen maken.

4.3.1 Functie eigenschappen

Als we functies individueel gaan bekijken zijn er een aantal eigenschappen die van belang zijn bij het partitionen. We bedoelen hier de eigenschappen van een functie die ervoor zorgen dat deze functie een grote bijdrage heeft aan de totale executietijd van de applicatie. We kunnen hierbij twee categoriëen onderscheiden:

- **Veel aanroepen**: doordat een functie relatief vaak wordt aangeroepen in een applicatie kan hij zelfs zonder dat de functie erg rekenintensief is, toch een grote bijdrage hebben in de totale executietijd.
- **Hoge rekenintensiteit**: als een functie een grote complexiteit heeft, oftewel, een grote rekenintensiteit heeft, heeft het een relatief hoge executietijd. In tegenstelling tot het vorige geval, kan zelf bij een gering aantal aanroepen de bijdrage aan de totale executietijd al relatief groot zijn.

Uit beide gevallen blijkt dat het zeer belangrijk is om van functies te weten wat hun bijdrage is aan de totale executietijd van de applicatie. Bovendien kan door het bekijken van het aantal aanroepen duidelijk worden of een functie tot de eerste of tweede catagorie hoort.
Bij het maken van afwegingen is het verder van belang om te weten wat het effect is wanneer we een functie niet in software maar in hardware uitvoeren. Zoals gezegd komen er een aantal andere kosten bij, wanneer we een functie in hardware willen uitvoeren. We noemen ze hier:

- **Configureren**: Willen we een functie in hardware uitvoeren, dan zal de hardware voor die specifieke functie (in combinatie met andere functies) eerst moeten worden geprogrammeerd. Hierbij worden de *statements* en de *lokale variabelen* omgezet in hardware *circuits*. Bij grote functies kan dit enige (relatief gezien) tijd duren. Als we hier geen rekening mee houden zou het kunnen voorkomen dat we een functie in hardware uitvoeren, die in software sneller zou kunnen worden uitgevoerd. Dit moet dus worden voorkomen, omdat we de hardware alleen willen gebruiken voor functies die écht winst opleveren.
- **Variabelen**: Verder moeten de variabelen die in de functie worden gemanipuleerd naar de hardware worden gekopieerd. Hierbij gaat het om **globale variabelen** en **parameters** (zowel variable als value parameters). Is de task uitgevoerd, dan moeten (zo nodig) variabelen worden teruggekopieerd naar het *Host systeem*. Hierbij gaat het om **globale variabelen** en **variable parameters**.
- **Result**: Ook moet, indien nodig, het resultaat van de task worden teruggeven aan de *caller* van de functie.

Bovendien speelt er nog een eigenschap een grote rol bij het maken van configuraties.

- **Omvang**: Hiermee bedoelen we de omvang die een functie heeft uitgedrukt in het aantal CLB's die de functie nodig heeft. Bij het afwegen van bepaalde configuraties op de hardware is het van belang om te weten welke componenten samen op de hardware passen. Aangezien we meestal FPGA's als hardware device gebruiken, nemen we het aantal CLB's als maat.

4.3.2 Inter-functie eigenschappen

Zoals eerder genoemd zijn ook inter-functie eigenschappen van belang bij partitioning, met name bij het indelen van functies in configuraties. Hieronder volgt een overzicht van de inter-functie eigenschappen:

- **Concurrentie**: de applicatie bestaat uit verschillende functies die op verschillende tijdstippen gedurende de executie ervan worden aangeroepen. Er bestaat een verband tussen functies in de zin van de volgorde waarin ze tijdens de executie worden uitgevoerd. Deze zogenaamde functie hiërarchie is belangrijk voor de partitioner. In een applicatie zijn er verschillende functies die we willen versnellen, dit doen we door ze uit te laten voeren op de hardware. Er zijn echter een aantal scenario's waarbij veel overhead wordt veroorzaakt door het herprogrammeren van de hardware. Het kan namelijk gebeuren dat twee (of meerdere functies) vlak na elkaar worden aangeroepen in bijvoorbeeld een loop. Als we dan beide functies in hardware willen realiseren (die niet tegelijk op de hardware aanwezig kunnen zijn) moet er zo vaak worden gewisseld van functie op de hardware, dat de overhead de winst van het versnellen overstijgt. Als twee functies kort na elkaar de herprogrammeerbare hardware claimen, zeggen we dat de twee functies met elkaar concurreren.
- **Memory Access**: door bepaalde variabelen, gebasseerd op de set functies van de configuratie, te localiseren op de hardware zou de performance kunnen worden vergroot. Dit speelt alleen een rol bij het optimaliseren van uitgekozen configuraties.
- **Kinderen/subfuncties**: er zijn functies (parents) die weer andere functies, hun zogenaamde kind-functies aanroepen. Voor deze functies nemen we aan dat als een parent-functie op de hardware wordt gezet zijn kind-functies automatisch ook op de hardware worden geplaatst. Dit vereenvoudigt de zaak, omdat er anders steeds gesprongen moet worden tussen hardware en software. Een speciaal geval hier is recursie, waarbij de functie zichzelf aanroept. Ook deze functies kunnen in hardware gerealiseerd worden.

Uit deze eigenschappen blijkt dat de structuur van de applicatie erg belangrijk is bij het efficiënt kiezen van configuraties.

4.4 Eigenschappen modelleren

Nu we de belangrijke eigenschappen hebben gespecificeerd, komen we aan bij de volgende fase. We moeten methodes bedenken waarmee we de door ons gewenste informatie uit een applicatie verkrijgen. Per eigenschap gaan we nu proberen om dit voor elkaar te krijgen door middel van het maken van een model per eigenschap.

4.4.1 Het aantal aanroepen van een functie

Willen we het exacte aantal aanroepen van een functie te weten komen, dan kunnen we extra code toevoegen aan de applicatie die dit administreert. Zo kunnen we als eerste statement van een functie laten afdrukken dat de functie is aangeroepen en dan nadat de applicatie is geeindigd per functie tellen hoe vaak het is aangeroepen. Gelukkig bestaan er tools voor die dit automatisch voor ons doen. Een van deze tools is profiler Gprof, die we in hoofdstuk 3 hebben besproken.

4.4.2 Executietijd per functie

Voor het bepalen van de executietijd van een softwarefunctie kunnen we verschillende methoden gebruiken. Zo kunnen we administreren hoeveel tijd er zit tussen het begin en het eind van een functie. Theoretisch gezien zou dit een exacte tijd moeten zijn. Echter, omdat we de prototypes laten draaien op timesharing-systemen is de executietijd vaak niet zo eenvoudig te berekenen. Een andere methode sampled met een bepaald interval de plek van de programcounter (PC). Hierdoor kan een serieuse schatting worden gemaakt van de executietijd per functie. Bovenstaande methoden zijn echter processor afhankelijk, en door verschillen van test en target processor misschien niet goed genoeg. We kunnen echter ook executietijden afleiden uit een model die de target processor representeerd. Deze laatste methode gaan we ook gebruiken voor de hardware versies van de componenten uit een applicatie. We willen niet eerst een functie in hardware realiseren voordat we er iets mee kunnen. Een model gebasseerd op de hardware die we gaan gebruiken, moet ons accurate informatie geven over de executietijden van de hardware functies.

4.4.3 Kosten Entry-protocol van een functie

Gaan we een functie in hardware realiseren, dan moeten we de kosten van de overhead optellen bij de executietijd van de functie zelf. Het Entry-protocol bestaat zoals eerder is gezegd uit twee delen. Deel 1 bestaat uit het configureren van de hardware voor deze specifieke functie. Hiervoor kunnen we aan de hand van de complexiteit schatten hoeveel tijd het programmeren gaat kosten. Voor deel 2, moeten we per functie bekijken welke variabelen het nodig heeft, en aan de hand daarvan schatten hoeveel tijd het kopiëren ervan kost. We kunnen hiervoor de gebruikte datastructuren voor de variabelen gebruiken.

4.4.4 Kosten Exit-protocol van een functie

Het Exit-protocol bestaat uit het terugkopieren van een aantal gemanipuleerde variabelen en een eventueel resultaat van de functie. Op basis van de gebruikte datastructuren kan bepaald worden hoeveel tijd het kopiëren gaat kosten.

4.4.5 Concurrentie

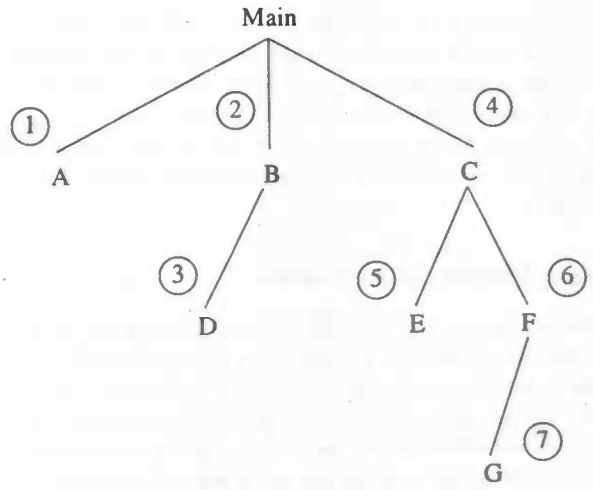
In een applicatie zijn er verschillende functies die we willen versnellen, dit doen we door ze uit te laten voeren op de herprogrammeerbare hardware. Er zijn echter een aantal scenario's waarbij veel overhead wordt veroorzaakt door het herprogrammeren van de hardware. Het kan namelijk gebeuren dat twee (of meerdere functies) vlak na elkaar worden aangeroepen in bijvoorbeeld een loop die duizenden keren wordt aangeroepen. Als we dan beide functies in hardware willen realiseren (die niet tegelijk op de hardware aanwezig kunnen zijn) moet er zo vaak worden gewisseld van functie op de hardware, dat de overhead de winst van het versnellen overstijgt. Als twee functies kort na elkaar de hardware claimen, zeggen we dat de twee functies met elkaar concurreren. Om dit probleem te voorkomen (of minimaliseren) willen we concurrentie zoveel mogelijk vermijden. Hiervoor maken we een model van de applicatie die de concurrentie van de verschillende functies modelleerd. We kunnen namelijk aan de hand van de *callgraph* van de applicatie, voor een deel, bepalen in welke mate functies met elkaar concurreren. Voor single thread applicaties kunnen we in een callgraph al eenvoudig conclusies trekken over concurrency. Bij multithreaded applicaties wordt dit al een stuk ingewikkelder. We zouden bijvoorbeeld per thread een callgraph kunnen

opstellen, echter de concurrentie tussen threads onderling wordt dan lastig te bepalen. In figuur 4.2 zien we een voorbeeld van de callgraph van een eenvoudig single-threaded voorbeeldprogramma. De applicatie bestaat uit zeven functies (A t/m G) en een hoofd-programma (Main). Het programma ziet er als volgt uit (in pseudocode): ___

```
Program Main begin
...;
A;  (functie aanroep van A)
...;
B;
...;
C;

 Function A begin
 ...;
 end

 Function B begin
 ...;
 D;
 ...;
 end

 Function C begin
 ...;
 E;
 ...;
 F;
 ...;
 end

 Function D begin
 ...;
 end

 Function E begin
 ...;
 end

 Function F begin
 ...;
 G;
 ...;
 end

 Function G begin
 ...;
 end

...;
end.
```

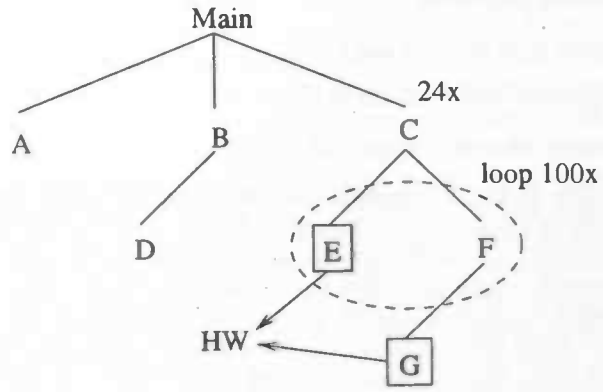
___ *Voorbeeld programma: Main*

In de figuur is met getallen bij de functies aangeven wanneer een functie wordt aangeroepen.



Figuur 4.2: Voorbeeld van een callgraph

Zo kan voor twee verschillende functies worden gekeken in welke mate ze concurreren. Dit kan worden gedaan door de labels van elkaar af te trekken. Zo concurreren functies A en G (7 - 1 = 6) niet (of in mindere mate) met elkaar en B en D (3 - 2 = 1) juist heel veel. Om nu die overhead zo laag mogelijk te houden moeten we zorgen dat functies die met elkaar concurreren zoveel mogelijk in dezelfde configuratie op de programmeerbare hardware gaan draaien, of een keus te maken welke van de functies op de hardware gaat draaien en welke niet.



Figuur 4.3: Loops in de executie

Bekijken we figuur 4.3 dan zien we daarin hetzelfde programma als boven, maar dan gedetailleerder. We zien nu dat functie C zelf 24 keer wordt aangeroepen. Bovendien bevat C een loop die 100 keer E en F na elkaar aanroept. Als we dan kiezen voor functies E en G in hardware, waarbij we ervan uit gaan dat de hardware slechts 1 task per keer kan bevatten moet het systeem dus 24 * 100 keer wisselen van hardware

component. Dit betekent dat de hardware 2 * 24 * 100 keer moet worden geherprogrammeerd. Vinden we dit te vaak, dan kunnen we dit op verschillende manieren oplossen. We zouden kunnen beslissen om slechts één van beide functies in hardware te laten draaien, dan versnellen we dus maar een enkele functie. Echter, als beide functies samen ook op de hardware passen, voegen we ze samen in één configuratie. In dat geval kunnen beide functies toch worden versneld, zonder dat we daar erg veel overhead van krijgen.

4.4.6 Omvang functies in hardware

In deze sectie beschrijven we een methode voor het bepalen van de omvang van functies. Hierbij hebben we als eenheid gekozen voor het aantal CLB's. Bij FPGA's wordt de functionaliteit namelijk gerealiseerd in de CLB's die op de FPGA aanwezig zijn. Voor het bepalen van de ruimte die een bepaalde functie inneemt, gebruiken we dan ook het aantal CLB's die ervoor nodig zijn om de functie te realiseren. Zoals eerder is genoemd is het van belang om te weten hoe groot bepaalde functionaliteit in hardware is, dit omdat de hardware maar een bepaalde omvang heeft.
Allereerst geven we hieronder een overzicht van een aantal operaties die vaak in een stuk code voorkomen:

1. variable declaratie: int a
2. procedure aanroep: proc_name(100, 9)
3. toekenning: a = 100
4. boolean expressie: a != b
5. optelling/aftrekking
6. vermenigvuldiging/deling
7. if statement: if (a) then {b} else {c}
8. while statement: while (test) do {b}
9. for statement: for (a=b; test; c–) {d}

Voor elk van deze statements (operaties) willen we nu een reële schatting maken voor het benodigde aantal CLB's. Belangrijk hierbij is dat we het model aanpasbaar is. Dit kan nodig zijn als bij experimenten blijkt dat een bepaalde schatting niet reëel is. We moeten zorgen dat we het model eenvoudig kunnen aanpassen. In [11] wordt een methode beschreven voor het schatten van de omvang (*area*) van *hardware building blocks*. Hierin wordt voor een aantal veelgebruikte operatoren uitgewerkt hoeveel ruimte ze nodig hebben uitgedrukt in het aantal CLB's. In tabel 4.1 zien we de zogenaamde *Characterisation Vector*. Aan de hand daarvan wordt een vergelijking opgesteld voor de benodigde ruimte.
Voor een functie met operatoren zoals die in bovenstaande tabel voorkomen kan nu als volgt de benodigde oppervlakte worden geschat: (A = #CLB's)

$$A = \frac{C1C2}{2} + 2C3C4^2 + \frac{C5C6C7}{4} + \frac{C8C9C10}{8} + \tag{4.1}$$

$$\frac{C11C12C13}{8} + \frac{C14C15}{2} \tag{4.2}$$

Description		Element
Adders	Number	C1
	Average word length	C2
Array Multipliers	Number	C3
	Average input word length	C4
Multiplexers	Number	C5
	Average fan-in	C6
	Average word length	C7
Bit-wise logic operations	Number	C8
	Average fan-in	C9
	Average word length	C10
LUTs	Number	C11
	Average fan-in	C12
	Average word length	C13

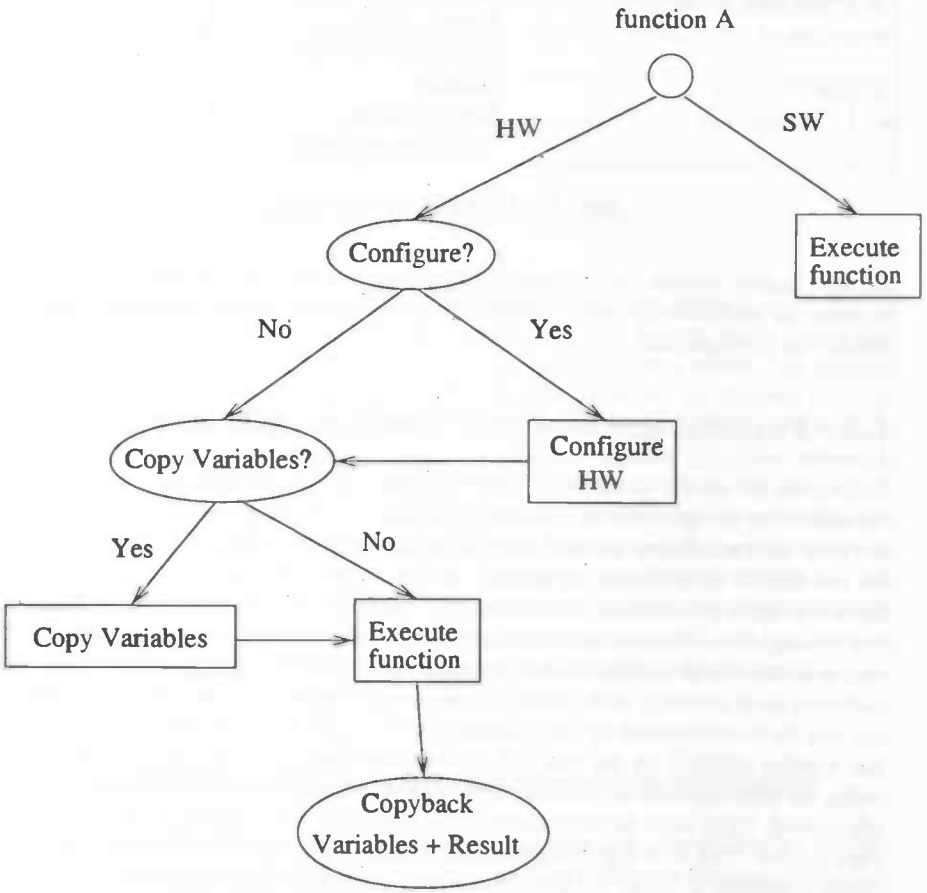
Tabel 4.1: Characterisation Vector

Op deze manier kunnen we dus voor de uitgekozen functies bepalen hoeveel ruimte ze innemen op het hardware device. Deze informatie is, zoals gezegd, belangrijk voor het maken van configuraties.

4.5 Functies geschikt voor hardware uitvoering

De functies die minder executietijd vergen wanneer we ze in hardware uitvoeren, willen we realiseren op het hardware device. Dit lijkt op het eerste gezicht triviaal, maar er zitten toch wat haken en ogen aan. In eerste instantie zouden we kunnen denken dat een functie in hardware per definitie sneller uitgevoerd kan worden dan dezelfde functie in software. Echter, niet alleen de executietijd van de functie in hardware is van belang. Het uitvoeren van een functie in hardware brengt namelijk een overhead met zich mee die er soms voor kan zorgen dat de tijdwinst teniet gedaan wordt. De hardware moet namelijk in de meeste gevallen eerst nog worden geconfigureerd (plaatsen van de functionaliteit op de hardware) alvorens de functionaliteit daadwerkelijk kan worden gebruikt en dat kost tijd. Bovendien moeten na de configuratie, indien nodig, de variabelen die in de functie worden gemanipuleerd naar de hardware worden gekopiëerd. Deze twee factoren noemen we samen het Entry-protocol voor een hardware functie. Ook is er een Exit-protocol, die zorgt dat gewijzigde variabelen worden teruggekopiëerd. In figuur 4.4 is te zien dat een component die in hardware zal worden uitgevoerd heel wat meer organisatie (en dus tijd) vergt dan hetzelfde component in software. Het is belangrijk om die 'overhead' (de extra kosten voor het configureren e.d.) in de gaten te houden bij het maken van efficiënte configuraties. Die overhead kan ervoor zorgen dat een component die in eerste instantie sneller zou kunnen worden uitgevoerd in hardware, toch niet uitgekozen wordt omdat zijn overhead te groot is, waardoor er geen winst meer is.
De volgende opsomming geeft de zaken weer die van belang zijn bij het berekenen van de tijdwinst:

- Executietijd in software (SWcost)
- Executietijd in hardware (HWcost)



Figuur 4.4: Verschil tussen SW flow en HW flow

- Configuratietijd hardware (CFcost)
- Tijd nodig voor het kopiëren van variabelen van en naar de hardware (VARcost)

Om te kijken of een functie echt winst oplevert in hardware kunnen we de volgende berekening maken, waarbij we de *worst case* situatie gebruiken (dwz, de hardware moet nog worden geconfigureerd):

$$winst = SWcost - (HWcost + CFcost + VARcost) \tag{4.3}$$

Het is duidelijk dat een hardware functie de meeste winst oplevert als hij reeds op de hardware aanwezig is, dus er niet meer geconfigureerd moet worden. Het kan echter voorkomen dat de hardware al weer geconfigureerd is voor deze functie, maar dat die functionaliteit nog actief is. Dit kan alleen voorkomen bij multithreaded applicaties, met als gevolg dat alvorens de functie kan worden uitgevoerd er gewacht moet worden totdat de functionaliteit klaar is.

4.6 Efficiënte configuraties

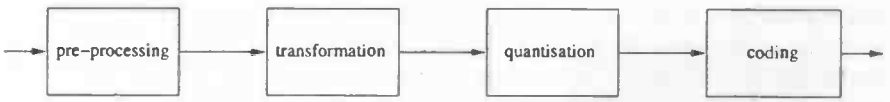
Op basis van bovenstaande eigenschappen moeten we nu gaan bedenken hoe we efficiënte configuraties maken. Hierbij zijn een aantal zaken van belang, de omvang van de componenten in hardware (CLB's) en de onderlinge concurrentie zoals eerder is uitgelegd. Het belangrijkste bij het maken van configuraties is, dat door het op een slimme manier groeperen van functies, de meeste tijdwinst wordt geboekt. Door verschillende sets van configuraties te testen kunnen we kijken welke set het beste is. Verder kunnen we door het grondig analyseren van de sets, optimalisaties toepassen waardoor nog meer tijdwinst wordt gemaakt. Het is duidelijk dat we zoveel mogelijk functies in een configuratie willen stoppen, omdat hierdoor minder vaak geherconfigureerd hoeft te worden. De hardware heeft zoals gezegd echter maar een beperkte omvang, waardoor we dus moeten kiezen welke functies er wel en niet in een configuratie moeten.
We willen zo weinig mogelijk verschillende configuraties hebben, waardoor het aantal keren dat de hardware moeten worden geherprogrammeerd geminimaliseerd wordt. Door configuraties te maken van functies die in elkaars nabijheid (qua tijd) worden uitgevoerd, dus de functies die concurreren, kan dit worden bewerkstelligd.

4.6.1 Omvang

Alvorens we kunnen beginnen met het maken van configuraties moeten we weten hoe groot de hardware is die wordt gebruikt in het systeem. Als we dit weten kunnen we configuraties maken van componenten, zodanig dat de ruimte op de hardware efficiënt benut wordt. Dit kunnen we doen door op een bepaalde manier de callgraph door te lopen en net zolang functies toevoegen tot een configuratie net zolang als dat er functies bij de configuratie passen. Past er geen functie meer in de configuratie, dan beschouwen we de configuratie als vol en beginnen we met een nieuwe configuratie. Net zolang tot we alle hardware kandidaten in configuraties hebben gestopt.

4.6.2 Fasen

Wanneer we configuraties willen gaan maken van componenten uit een applicatie, is het ten eerste belangrijk om te gaan zoeken naar verschillende fasen die in het programma zitten. Hoewel deze fasen misschien niet altijd duidelijk zichtbaar zijn, zijn er altijd wel bepaalde onderdelen te onderscheiden. Het kenmerk van een bepaalde fase is dat daarin bepaalde functionaliteit wordt gebruikt die in een andere fase niet of nauwelijks wordt gebruikt. Bij het indelen van configuraties kunnen we hier handig gebruik van maken, door configuraties te maken op basis van deze verschillende programma fases. Als voorbeeld van een applicatie waarin verschillende fases zitten die duidelijk te onderscheiden zijn nemen we een compressie algoritme, zie figuur 4.5. In de *prepro-*

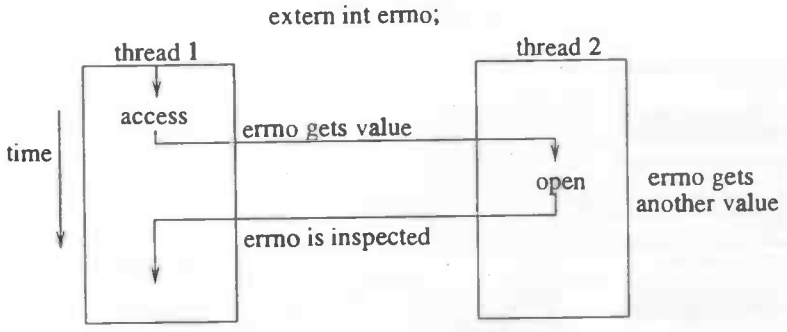


Figuur 4.5: Een compressie algoritme

cessing stap worden de input data geconverteerd naar een ander formaat, bijvoorbeeld van RGB naar YUV. Daarna wordt in de transformatie stap de data uit de vorige fase geschikt gemaakt voor de quantisatie stap. In de quantisatie stap wordt data reductie toegepast, waarna in de *coding* stap de file met data uit de vorige stappen zo compact mogelijk wordt gerepresenteerd. We zien hier dus vier duidelijk van elkaar te onderscheiden programma-fases, doordat er in de fases onderling geen concurrentie is kunnen we configuraties maken per fase.
Om die fasen uit een applicatie te filteren kunnen we de eerder genoemde callgraph gebruiken. Op basis van verschillende algoritmen kunnen we dan aan de hand van de callgraph geschikte configuraties maken. Belangrijk hierbij is dat de omvang van de functies samen niet groter wordt dan de gebruikte omvang van de hardware die in het systeem wordt gebruikt. Door gebruik te maken van verschillende algoritmes en de uitkomsten ervan te analyseren, met een profiler, kan uiteindelijk de beste set van configuraties worden uitgekozen.

4.7 Applicaties op herprogrammeerbare systemen

We gebruiken de herprogrammeerbare hardware om bepaalde rekenintensieve stukken code van applicaties te versnellen. Er zijn echter verschillende soorten applicaties, waarbij het niet altijd eenvoudig is om al vooraf efficiënte keuzes te maken aangaande groeperingen van stukken code voor de hardware. Ten eerste kunnen we twee globale categorieën van programma's onderscheiden: sequentiële programma's en parallelle programma's. Bij sequentiële programma's verandert de flow van het programma niet wanneer we een bepaald deel in hardware uitvoeren, het programma wordt slechts versneld, wat natuurlijk hetgeen is wat willen. Of het programma wordt uitgevoerd in software op de GPP, of een deel wordt uitgevoerd op de hardware, maar nooit tegelijk. Hebben we echter een programma dat bestaat uit meerdere threads die afhankelijk van elkaar zijn, dan kan de flow van het programma wel veranderen als een bepaald deel ineens sneller wordt uitgevoerd. Aangaande threads moeten we een aantal zaken in ons achterhoofd houden. Zo hebben alle threads precies dezelfde adresruimte, ze gebruiken dus dezelfde globale variabelen. Hierbij dient in het ontwerp rekening gehouden te

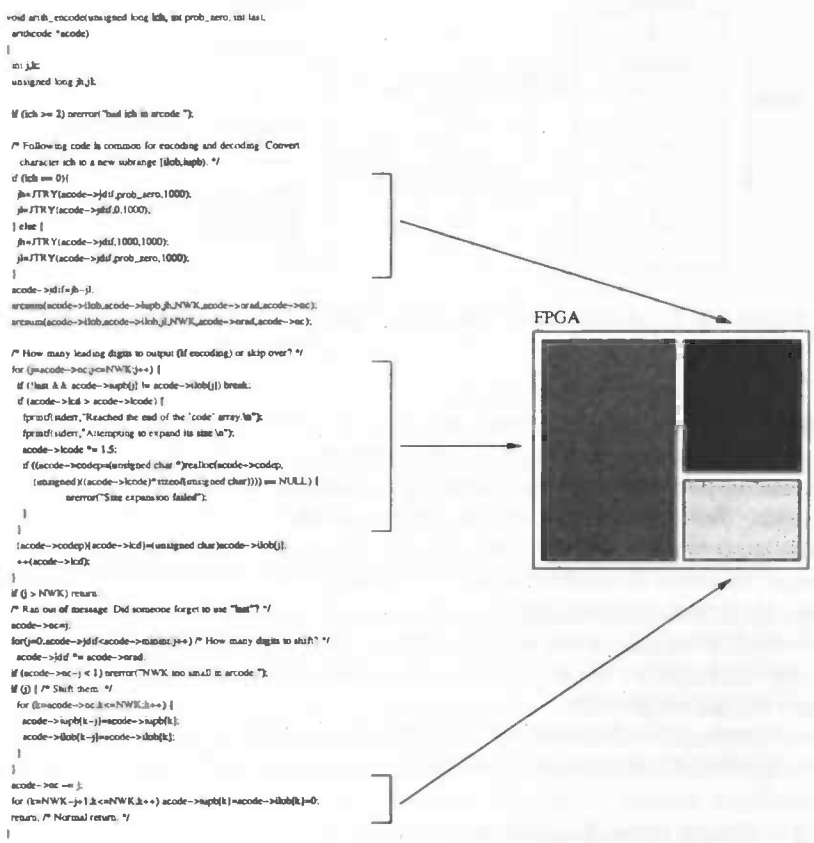


Figuur 4.6: Conflicten tussen threads door gebruik van een globale variabele

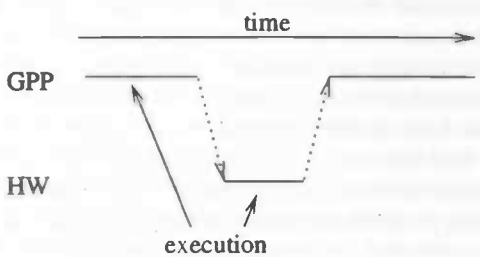
worden, waardoor conflicten worden vermeden. In figuur 4.6 zien we twee threads die allebei via aanroepen van $ACCESS$ (door thread 1) en $OPEN$ (door thread 2) de variabele $errno$ veranderen. Een oplossing zou kunnen zijn om globale variabelen te verbieden. Welke oplossing er dan ook gekozen wordt, het is de taak aan de applicatie programmeur om hier rekening mee te houden. Een andere zaak waar we rekening mee dienen te houden is de toestand waarin de thread zich bevindt. Er zijn vier toestanden: lopend (de thread is actief), geblokkeerd, klaar (de thread is klaar om te worden uitgevoerd) of beëindigd (de thread is afgelopen). Als voorbeeld kunnen we noemen dat de hardware resources die een thread bezit, na beëindiging van de thread aan andere threads worden toegewezen.
In de volgende secties beginnen we met eenvoudige modellen voor single thread applicaties en kijken daarna naar de ingewikkeldere situaties in multithreaded applicaties.

4.7.1 Single thread applicaties

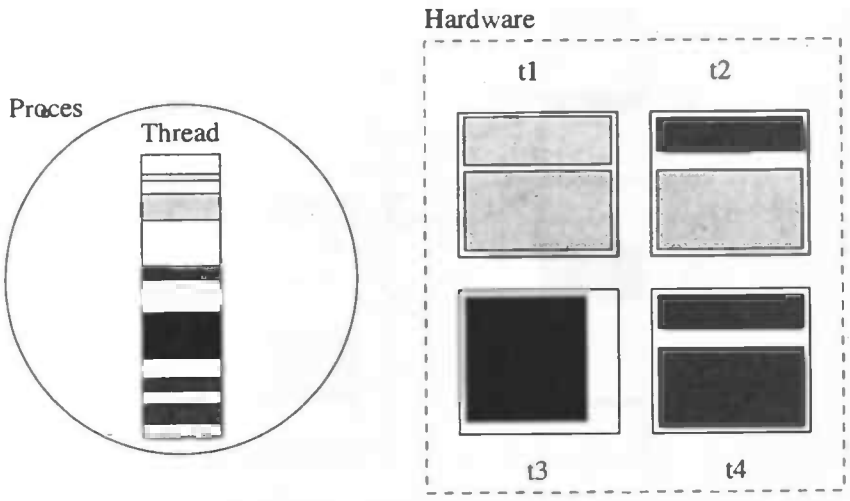
In dit model gaan we uit van een programma bestaande uit één *thread*, waaruit een aantal stukken code moeten worden versneld. Hierbij bestaat de configuratie die naar de hardware wordt geschreven uit een verzameling van één of meer stukken code per keer. Hoeveel stukken code er tegelijk op de hardware passen is afhankelijk van de omvang van het hardware device en de omvang van de stukken code zelf. In figuur 4.7 zien we een deel van de source van een applicatie waarin een aantal delen zijn aangewezen die moeten worden versneld. We kunnen zien dat de ruimte op de hardware volledig is bezet en dit alleen door een klein deel van de applicatie. Willen we nog meer stukken code versnellen, dan zullen we configuraties op de hardware moeten aanpassen of een andere configuratie op de hardware plaatsen. In het uiteindelijke systeem zien we dan dat als een stuk code moet worden versneld, we dan een omschakeling krijgen van executie op de GPP naar executie op de herprogrammeerbare hardware. In figuur 4.8 zien we zo'n omschakeling, eerst wordt een stukje code uitgevoerd op de GPP, daarna wordt na een vertraging (door o.a. configuratiekosten) een stukje code uitgevoerd op de herprogrammeerbare hardware. De vertraging kan minimaal zijn als de functionaliteit al aanwezig is op de hardware, er hoeven in dat geval (indien noodzakelijk) alleen variabelen worden gekopiëerd naar de hardware. Echter, als eerst nog de hardware moet worden geconfigureerd dan wordt de winst verminderd. Hierdoor moeten we ons afvragen of het überhaupt wel zin heeft om dit deel te versnellen, en niet beter de ruimte beschikbaar kunnen stellen aan een functie die wél echt winst oplevert. Uit de



Figuur 4.7: Een aantal stukken code die samen één configuratie vormen op de hardware



Figuur 4.8: Verandering van executie van GPP naar hardware

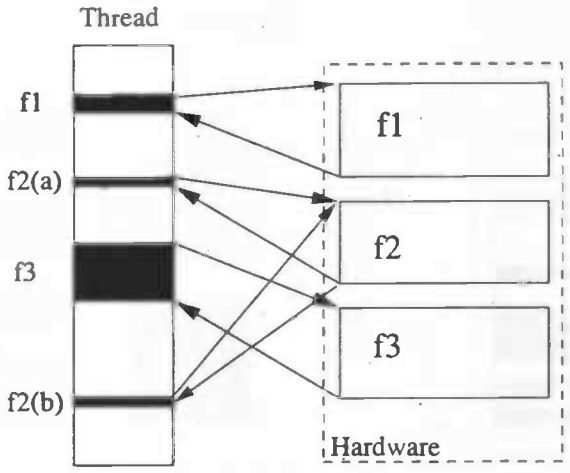


Figuur 4.9: Een proces met één thread, waarbij op de hardware op verschillende tijdstippen verschillende configuraties aanwezig zijn

figuur blijkt dat de flow van de applicatie, ook bij het uitvoeren van een component in hardware, sequentieel blijft. Dus er wordt of een deel op de GPP uitgevoerd of een deel op de hardware, maar niet tegelijk. In figuur 4.9 zien we een voorbeeld met een proces bestaande uit één thread. In de thread zijn een aantal stukken code aangewezen die we uit willen laten voeren op de hardware. Bovendien zijn de stukken code gegroepeerd in een aantal configuraties die op verschillende tijdstippen op de hardware aanwezig zijn. Zoals gezegd willen we zodanig groepen maken dat er zo weinig mogelijk hoeft worden geherprogrammeerd. Hiervoor moeten we de applicatie grondig analyseren, zodat we kunnen inschatten wat goede configuraties zijn. In dit geval maken we gebruik van de sequentiële aard van deze programma's, we kunnen dus vooraf redelijk goed inschatten wanneer bepaalde delen van de code op welk moment van de executie plaatsvinden. Wat we nu willen is dat we configuraties maken van stukken code die in eenzelfde tijdsdeel van de executie plaatsvinden, omdat die componenten elkaar het meest 'dwarszitten' bij het claimen van de hardware. Om te bekijken welke componenten het beste samengevoegd kunnen worden in een configuratie gebruiken we een profiler.

4.7.2 Scenario's

We kunnen nu een aantal scenario's bedenken die voor kunnen komen in dit soort applicaties. Met behulp van figuur 4.10 kunnen we een aantal mogelijkheden bij langsgaan. Ten eerste zien we dat in de thread een aantal blokken zijn aangeven die op de hardware moeten worden uitgevoerd. Verder zien we dat een bepaalde functie (f2) op verschillende plekken in de thread wordt aangeroepen. Dit maakt overigens geen verschil met andere functies, omdat de configuration controller ook in dit specifieke geval moet kijken of het component al op de hardware aanwezig was of dat hij opnieuw moet worden geprogrammeerd. Doordat de thread sequentiëel wordt uitgevoerd, kan het niet gebeuren dat twee hardware claims voor eenzelfde component met elkaar interfereren.



Figuur 4.10: Functies worden vaker gebruikt in de applicatie.

4.7.3 Multithreaded applicaties met onafhankelijke threads

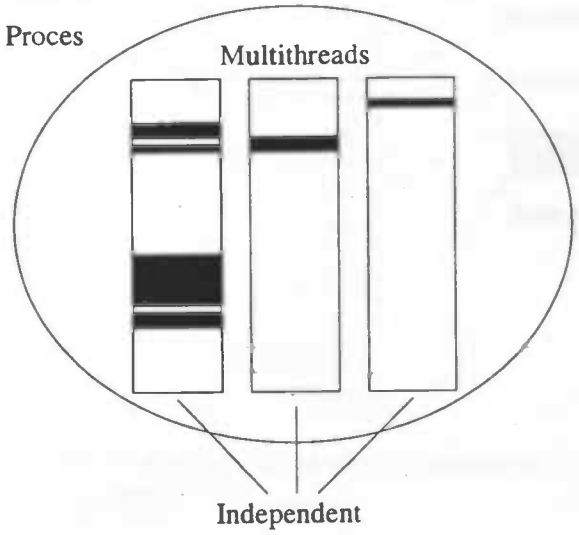
Ingewikkelder wordt het wanneer we een parallel programma hebben waarin uit meerdere threads stukken code versneld moeten worden. We gaan hier uit van een situatie waarin de threads onafhankelijk van elkaar draaien (zie figuur 4.11). We hebben weer hetzelfde doel voor ogen: een zo snel mogelijke applicatie. Hiervoor willen we weer de ruimte op de hardware zo efficiënt mogelijk gaan gebruiken. Dit doen we door weer configuraties te maken van componenten die in eenzelfde tijdsdeel van de applicatie zitten. Hierbij hebben we echter een probleem. De threads draaien onafhankelijk, waardoor het dus erg moeilijk (of onmogelijk) is om te kijken welke stukken code (die versneld moeten worden) in hetzelfde tijdsdeel zitten. We kunnen met dit probleem op een aantal manieren omgaan. In de volgende twee secties bekijken we het probleem op verschillende manieren.

Aparte resources per thread

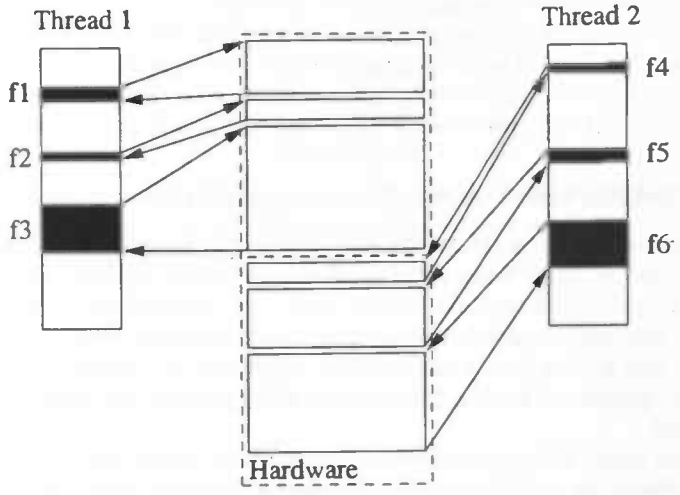
In deze situatie wordt per thread ruimte beschikbaar gesteld op de herprogrammeerbare hardware, zie figuur 4.12. Voordeel hiervan is dat concurrentie tussen verschillende threads geen rol meer speelt bij het claimen van hardware resources. Nadeel is dat er dus per thread minder ruimte eschikbaar is, waardoor grotere configuraties niet meer gaan passen. Bovendien moet bij het beëindigen van threads de gereserveerde ruimte op de hardware worden vrijgegeven aan de andere threads, dit is de taak van de configuration controller.
Een specifiek geval hiervan is om de hardware maar aan één enkele thread toe te wijzen. De situatie die we dan krijgen is dezelfde als die van de single thread applicaties.

Hardware resources worden geshared

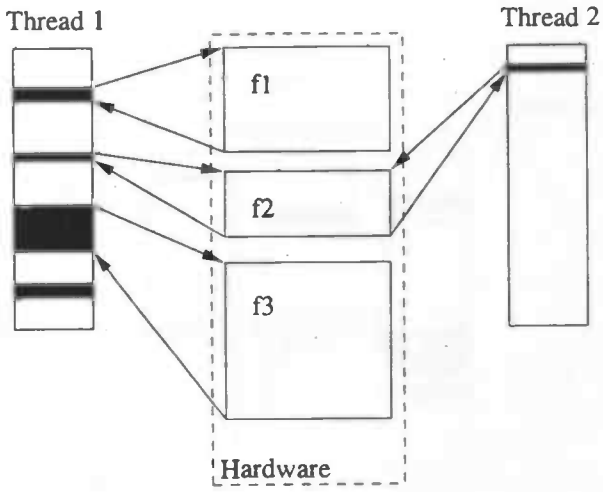
In deze situatie kan elke thread op elk tijdstip van de executie een claim doen op (een deel van) de herprogrammeerbare hardware. Hierbij is het noodzakelijk dat de configuration controller zorgt voor een correcte afhandeling van de claims van de verschillende



Figuur 4.11: Een proces met meerdere onafhankelijke threads, waarbij stukken code uit verschillende threads moeten worden versneld



Figuur 4.12: Resources gereserveerd per thread

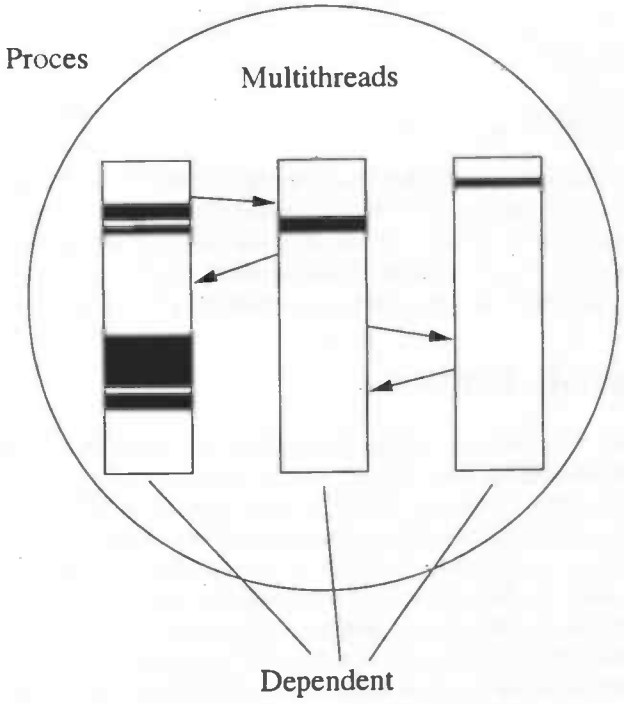


Figuur 4.13: Een component op de hardware wordt door meerdere threads gebruikt

threads. Zonder voorzorgsmaatregelen zou het kunnen voorkomen dat een thread een hardware functie aanroept die niet meer op de hardware aanwezig is, omdat een andere thread deze met een andere claim heeft overschreven. Zoals gezegd is het maken van groepen van stukken code uit verschillende threads niet eenvoudig, we kunnen er dan ook voor kiezen om groepen te maken per thread. We lopen dan het risico dat een configuratie die net op de hardware is gezet wordt overschreven door een andere thread, maar het is de taak van de configuration controller om claims eerlijk af te handelen. Bovendien kan een bepaalde component, die reeds aanwezig is op de hardware, worden geclaimd door meerdere threads (zie figuur 4.13). Dit heeft als gevolg dat het component nog bezig kan zijn met zijn executie als een andere thread dit component ook claimt. Zoals ook al eerder is gezegd moet de configuration controller dan een beslissing nemen of de nieuwe claim moet wachten tot de oude claim afgehandeld is, of dat de claim niet in hardware wordt uitgevoerd maar in software op de processor.

4.7.4 Multithreaded applicaties met afhankelijke threads

In deze situatie hebben we een parallel programma waarbij de threads afhankelijk van elkaar draaien (zie figuur 4.14). Hierbij kan de flow van de applicatie wel veranderen door het versnellen van bepaalde stukken code. De afhankelijkheid van de threads uit zich in messages die ertussen worden verstuurd. Doordat de threads afhankelijk van elkaar zijn, kunnen we via analyse meer zeggen over de samenhang ertussen. Zo zouden we bijvoorbeeld kunnen constateren in welke gevallen een thread wordt gestart of beëindigd.
Ook in deze sitatie kunnen we op eenzelfde manier als beschreven in de vorige sectie, de hardware als *shared resource* gebruiken of bepaalde delen van de hardware reserveren per thread. Als we bepaalde delen van de hardware reserveren per thread, houdt dat in dat het mogelijk wordt om de threads apart te profilen en we dus als het ware dezelfde methode voor het opsplitsen kunnen gebruiken als bij single thread applicaties.



Figuur 4.14: Een proces met meerdere afhankelijke threads, waarbij stukken code uit verschillende threads moeten worden versneld

Hoofdstuk 5

Profiling toegepast

5.1 Inleiding

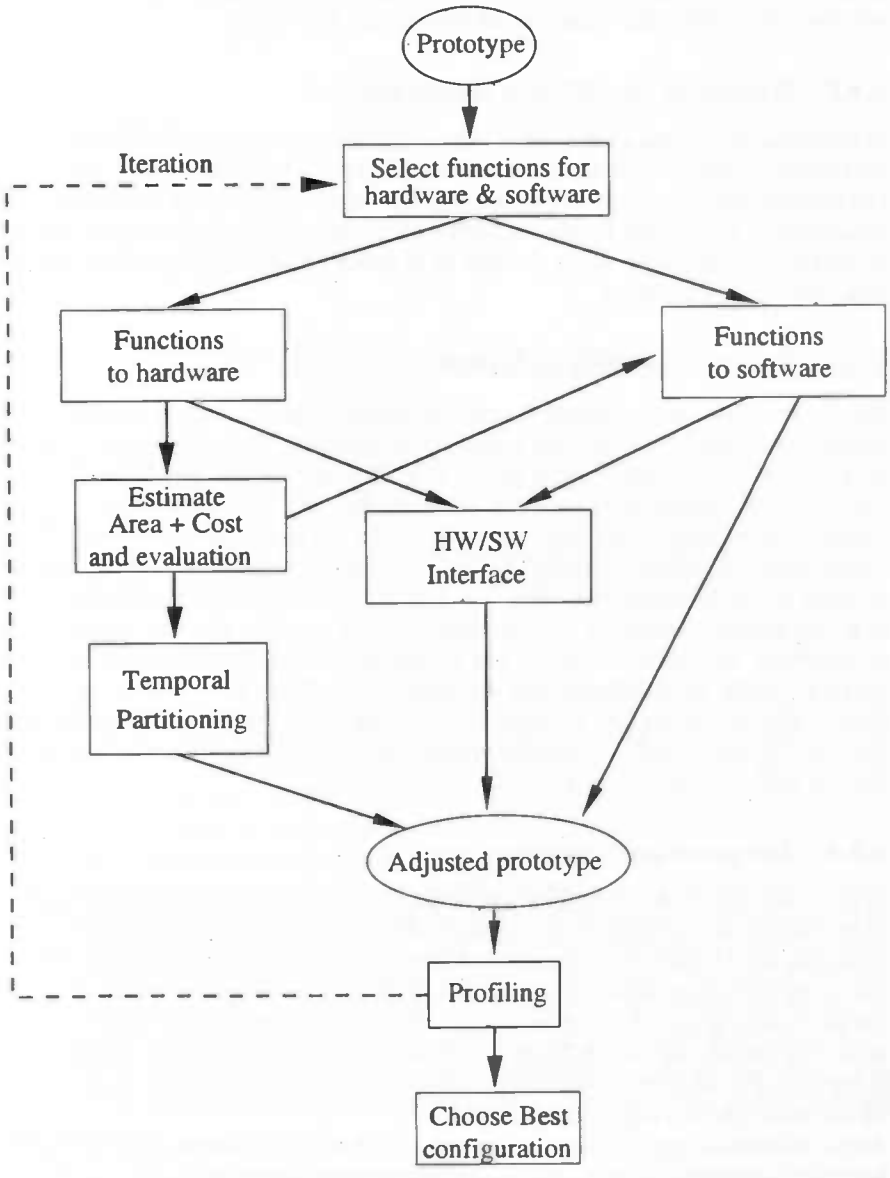
We beginnen dit hoofdstuk met een beschrijving van hoe we het voorgaande kunnen gebruiken om een applicatie op een herprogrammeerbaar systeem zo efficënt mogelijk te kunnen laten draaien. In het tweede deel van dit hoofdstuk gaan we een aantal experimenten doen met een gesimplificeerd systeem. In dat systeem kijken we naar single-thread applicaties die gaan draaien op een herprogrammeerbaar systeem.

5.2 Simulatie hardware

Via de profiler willen we iets kunnen zeggen over een bepaalde configuratie van de applicatie. Belangrijk hierbij is dat we daaruit conclusies gaan trekken waarna het systeem echt wordt gerealiseerd. Die conclusies moeten zo efficiënt mogelijk zijn, zodat we een zo optimaal mogelijk resultaat verkrijgen. Omdat we echter conclusies gaan trekken over een prototype die slechts in software is gerealiseerd, hebben we een goed model nodig die de hardware componenten representeerd. Dit komt erop neer dat we van een aantal zaken van een hardware component willen weten, zonder dat het component daadwerkelijk in hardware wordt of is gerealiseerd. De zaken die we van een component die in hardware wordt gerealiseerd willen weten hebben we al eerder besproken, namelijk: zijn kosten (eigen + overhead) en omvang. Deze eigenschappen moeten bepaald worden met behulp van tools die werken op basis van de methodes besproken in hoofdstuk 4.

5.3 Verschillende fasen

In figuur 5.1 zien we een overzicht van de stappen waarmee we een applicatie kunnen optimaliseren. We kunnen het proces zien als een iteratief proces waarin steeds moet worden beoordeeld of bepaalde beslissingen gunstig of negatief uit pakken. In de volgende subsecties geven we een beschrijving van die fases.



Figuur 5.1: Overzicht van de verschillende stappen

5.3.1 Prototype

Als start nemen we een prototype van de applicatie, die geheel in software is geschreven. Hierdoor kunnen we via een software profiler belangrijke data uit de applicatie halen. In deze fase zal ook duidelijk worden of de applicatie gebruik maakt van één thread of van meerdere threads, die al dan niet afhankelijk van elkaar zijn.

5.3.2 Hardware en software functies kiezen

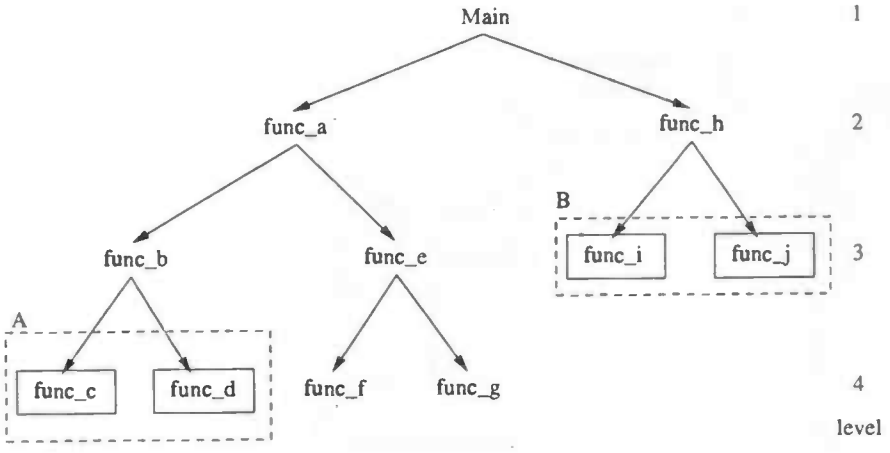
Via profiling kunnen we, zoals gezegd, belangrijke data uit de applicatie halen waarmee we kunnen beslissen welke componenten we het beste in hardware kunnen realiseren. Dat zijn de componenten die de grootste bijdrage vormen tot de totale executietijd van de applicatie. In de eerste iteratie kiezen we alleen functies voor software om dan via de profiler te gaan kijken welke functies er in eerste instantie in aanmerking komen voor uitvoering in hardware.

5.3.3 Schatten omvang en kosten

Van de uitgekozen componenten voor de hardware, is het daarna van belang om te weten hoeveel ruimte die zal gaan kosten op de hardware. Bovendien is het voor het simuleren van de hardware nodig om te weten hoeveel tijd het gaat kosten om het component daadwerkelijk in hardware uit te voeren. Bij deze kosten horen ook de kosten voor het herprogrammeren en het kopiëren van benodigde variabelen van en naar de hardware. Zijn de waarden berekend, dan volgt er een evaluatie van de functies op basis van de berekende waarden. Via deze evaluatie moet worden bekeken of de in eerste instantie uitgekozen functies daadwerkelijk geschikt zijn voor uitvoeren op de hardware. Bij bepaalde functies kan het namelijk mogelijk zijn dat de winst (zie formule 4.3) die wordt behaald door uitvoering op hardware zo minimaal zijn dat de functie beter in software kan worden uitgevoerd, om resources te bewaren voor andere functies. Bovendien zullen er functies tussen zitten die teveel CLB's nodig hebben, en dus ook zullen worden afgewezen.

5.3.4 Temporal partitioning

In deze fase worden de configuraties gemaakt die op verschillende tijdstippen tijdens de executie op de hardware worden geladen. Met behulp van modellen die in het vorige hoofdstuk zijn beschreven moeten in deze stap efficiënte configuraties gemaakt. Doordat we bij deze experimenten Gprof gebruiken als profiler kunnen we via de callgraph die kan worden gereproduceerd door Gprof afleiden welke functies met elkaar concurreren. We kunnen hier verschillende algoritmes gebruiken om de set van configuraties te bepalen. Die we in een latere stap kunnen testen met de profiler om te kijken welke set het beste is en dus uitgekozen zal worden.
Als we kijken naar figuur 5.1 zien we vier functies in een blok weergegeven, dat zijn de hardware kandidaten. Voor het maken van configuraties kunnen we nu het volgende algoritme gebruiken. We beginnen bij het laagste niveau (in dit geval 4) en kijken welke kandidaten op dat niveau dezelfde parent hebben en voegen die samen in een configuratie mits die samen passen anders komen ze in aparte configuraties. Daarna gaan we een niveau hoger (hier 3) en kijken daar ook weer of kandidaten dezelfde parent hebben en voegen die samen in een configuratie. Hetzelfde voor de hogere niveau's. Daarna ga we kijken naar de parent van de parent (de grootouder), hebben configuraties die



Figuur 5.2: Configuraties op basis van callgraph

gemeen, dan proberen we die configuraties samen te voegen, past dat niet dan houden we de configuraties gescheiden en anders maken we er een samengevoegde configuratie van. Zo gaan we net zo lang door tot we bij Main zijn aangekomen en eindigen we met een set van configuraties. In figuur 5.1 zien we twee configuraties, A en B, die vanwege hun omvang niet samengevoegd konden worden.

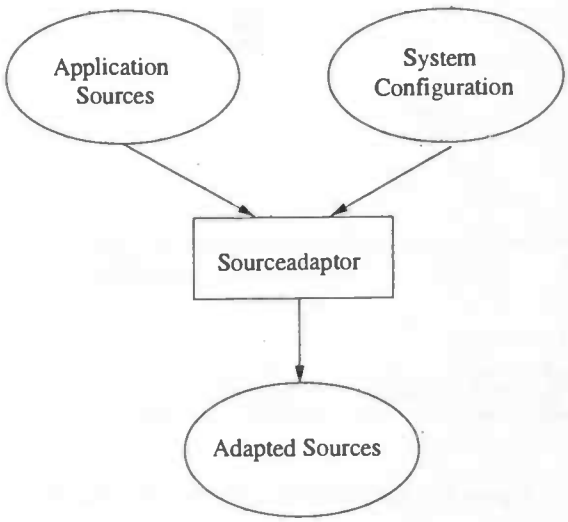
5.3.5 Aanpassing prototype

We gaan nu het prototype op zo'n manier aanpassen dat het lijkt alsof we het laten draaien op een herprogrammeerbaar systeem. De aanpassing wordt gedaan op basis van een bepaalde systeemconfiguratie, die weergeeft welke functies op welk moment in hardware of software worden uitgevoerd. Hiervoor gaan we een aantal elementen aan de source van de applicatie toevoegen. Dit maakt het ons mogelijk om de profiler eenvoudig te gebruiken zonder aan de profiler zelf aanpassingen te hoeven doen. Ten eerste zal er een routine worden toegevoegd die aangeeft dat er gewisseld gaat worden (er komt een nieuwe configuratie om de hardware). Ten tweede voegen we voor elke functie die in hardware uitgevoerd kan gaan worden een hardware variant toe. Deze hardware variant is te herkennen aan een h voor de functienaam. De executietijd voor de hardware functie zal worden bepaald aan de hand van het model die we aan de hand van het schatten ervan hebben opgesteld. Voor het aanpassen kunnen we een tool gebruiken die dit automatisch voor ons doet (zie figuur 5.3). Met als input de sources van de applicatie en een bepaalde systeemconfiguratie kan de tool dan de aanpassingen toevoegen.

5.4 Simplificatie

We hebben ervoor gekozen om een aantal vereenvoudigingen te doen waarmee het mogelijk wordt gemaakt om op eenvoudige wijze een aantal experimenten te kunnen doen zonder dat we al allerlei tools nodig hebben. We noemen ze hier:

1. Ten eerste willen we een bestaande profiler gebruiken, namelijk Gprof. Hierdoor



Figuur 5.3: Automatisch aanpassing van de sourcecode

gaan we uit van applicaties die bestaan uit 1 thread, dit omdat Gprof niet zonder aanpassing met meerdere threads kan omgaan.

2. De tool voor het aanpassen van de sources hebben we nog niet geïmplementeerd, waardoor we de aanpassingen handmatig zullen moeten doen.

3. Het simuleren van hardware functies doen we door handmatig functies toe te voegen die iets sneller kunnen worden uitgevoerd dan de normale software variant. We werken op zo'n manier omdat we liever geen aanpassingen deden aan de code van Gprof. Zouden we daar wel voor kiezen dan zouden we Gprof zodaning moeten aanpassen dat het voor hardware functies bepaalde tijden neemt die het niet zelf uitrekent maar door middel van bijvoorbeeld een configuratie file beschikbaar is. En andere manier is door op basis van de configuratiefile timers bij te houden die ervoor zorgen dat een hardware functie de tijd gebruikt die vooraf is uitgerekend.

4. We houden niet impliciet rekening met de kosten voor het Entry en Exit protocol.

5.5 Het eerste experiment

Voor het eerste experiment gebruiken we het onderstaande zeer eenvoudige test programma.

```
/* example */
#include <stdio.h>

void func_c() {
  int i,x;
  for (i=0; i<10000; i++) {
    x+=x*i%2*i%2;
  }
}
```

```
void func_d() {
  int i,x;
  for (i=0; i<10000; i++) {
    x+=x*i%2;
  }
}

void func_b() {

  func_c();

  func_d();
}

void func_f() {
  int i;
  int x=0;
  for (i=0; i<10000; i++) {
  }
}

void func_g() {
  int i;
  for (i=0; i<1000; i++) {
  }
}

void func_e() {

  func_f();

  func_g();
}

void func_a() {
  int i;
  for (i=0; i<523; i++) {
    func_b();

    func_e();
  }
}

void func_i() {
  int i,x;
  for (i=0; i<10000; i++) {
    x*=x*i%2*i%2;
  }
}

void func_j() {
  int i,x;
  for (i=0; i<10000; i++) {
    x*=x*i%2*i%2;
  }
}

void func_h() {
  int i;

  for (i=0; i<238; i++) {
    func_i();
    func_j();
    func_d();
  }
}

int main ()
{
  int i;
  for (i=0; i<5; i++) {
    func_a();
```

```
    func_h();
  }

  printf("klaar");
  return 0;
}
```

Stap 1

We kiezen dus zoals gezegd in de eerste iteratie slechts functies voor software. Er hoeft dus nog niets aangepast worden aan het prototype.

Stap 2

In deze stap gebruiken we de profiler Gprof om die functies te vinden die in aanmerking komen voor uitvoering in hardware. Hieronder zien we een deel van de uitvoer van Gprof:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.63      1.04     1.04     2615     0.40     0.40  func_c
 26.84      1.87     0.83     3805     0.22     0.22  func_d
 19.72      2.49     0.61     1190     0.51     0.51  func_i
 18.43      3.06     0.57     1190     0.48     0.48  func_j
  1.29      3.10     0.04     2615     0.02     0.02  func_f
  0.32      3.11     0.01     2615     0.00     0.00  func_g
  0.00      3.11     0.00     2615     0.00     0.62  func_b
  0.00      3.11     0.00     2615     0.00     0.02  func_e
  0.00      3.11     0.00        5     0.00   332.85  func_a
  0.00      3.11     0.00        5     0.00   288.58  func_h
```

Uit deze data blijkt dat de eerste vier functies (c, d, i en j) de grootste bijdrage hebben tot de totale executietijd. We kiezen dan ook deze functies voor uitvoering in hardware.

Stap 3

In deze stap moet worden bepaald wat de kosten zijn van de uitgekozen functies in hardware en hoeveel CLB's ervoor nodig zijn om zo'n functie in hardware te realiseren. Het berekenen van de kosten omvat de configuratietijd, de executietijd van de functie en de tijd die het kost dat de benodigde variabelen van en naar de hardware worden gekopiëerd. Aangezien we hier nog geen bruikbare tools voor hebben kiezen we 'met de hand' de waarden hiervoor.

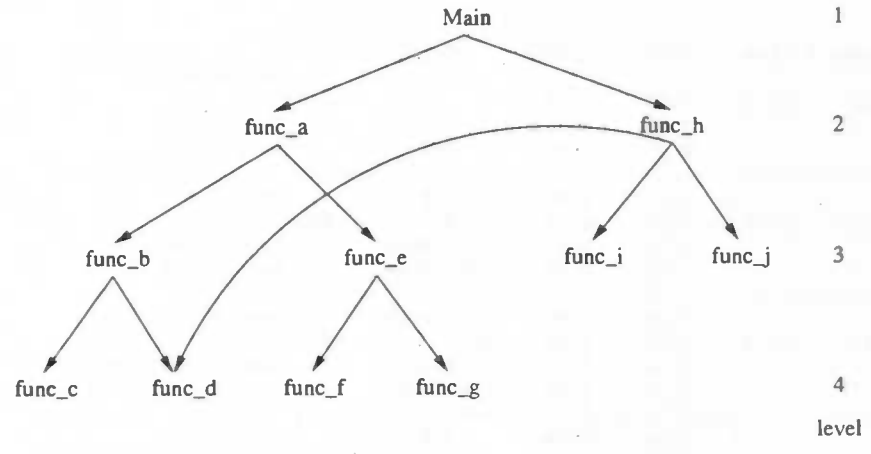
Stap 4

In deze stap maken we de configuraties. Via de callgraph die de eerste profiling iteratie heeft opgeleverd kunnen we kijken hoe we de configuraties indelen. Hieronder zien we de uitvoer van Gprof en in figuur 5.4 de grafische weergave ervan.

```
                     Call graph


granularity: each sample hit covers 2 byte(s) for 0.32% of 3.11 seconds
```

```
index % time    self  children    called     name
                                             <spontaneous>
[1]    100.0    0.00    3.11                 main [1]
                0.00    1.66       5/5           func_a [2]
                0.00    1.44       5/5           func_h [4]
-----------------------------------------------
                0.00    1.66       5/5           main [1]
[2]     53.6    0.00    1.66       5         func_a [2]
                0.00    1.61    2615/2615        func_b [3]
                0.00    0.05    2615/2615        func_e [9]
-----------------------------------------------
                0.00    1.61    2615/2615        func_a [2]
[3]     51.9    0.00    1.61    2615         func_b [3]
                1.04    0.00    2615/2615        func_c [5]
                0.57    0.00    2615/3805        func_d [6]
-----------------------------------------------
                0.00    1.44       5/5           main [1]
[4]     46.4    0.00    1.44       5         func_h [4]
                0.61    0.00    1190/1190        func_i [7]
                0.57    0.00    1190/1190        func_j [8]
                0.26    0.00    1190/3805        func_d [6]
-----------------------------------------------
                1.04    0.00    2615/2615        func_b [3]
[5]     33.5    1.04    0.00    2615         func_c [5]
-----------------------------------------------
                0.26    0.00    1190/3805        func_h [4]
                0.57    0.00    2615/3805        func_b [3]
[6]     26.8    0.83    0.00    3805         func_d [6]
-----------------------------------------------
                0.61    0.00    1190/1190        func_h [4]
[7]     19.7    0.61    0.00    1190         func_i [7]
-----------------------------------------------
                0.57    0.00    1190/1190        func_h [4]
[8]     18.4    0.57    0.00    1190         func_j [8]
-----------------------------------------------
                0.00    0.05    2615/2615        func_a [2]
[9]      1.6    0.00    0.05    2615         func_e [9]
                0.04    0.00    2615/2615        func_f [10]
                0.01    0.00    2615/2615        func_g [11]
-----------------------------------------------
                0.04    0.00    2615/2615        func_e [9]
[10]     1.3    0.04    0.00    2615         func_f [10]
-----------------------------------------------
                0.01    0.00    2615/2615        func_e [9]
[11]     0.3    0.01    0.00    2615         func_g [11]
-----------------------------------------------

Index by function name

   [2] func_a                  [9] func_e                  [7] func_i
   [3] func_b                 [10] func_f                  [8] func_j
   [5] func_c                 [11] func_g
   [6] func_d                  [4] func_h
```

Op basis van bovenstaande gegevens kiezen we voor twee configuraties, namelijk: configuratie 0 met $func_c$ en $func_d$ en configuratie 1 met $func_d$, $func_i$ en $func_j$.



Figuur 5.4: Grafische weergave van de Callgraph van het test-programma

Stap 5

Nu moeten we het prototype gaan aanpassen aan de hand van de beslissingen die we in vorige stappen hebben genomen. Het aangepast test-programma staat hieronder met daarin twee extra functies waarmee we simuleren dat er gewisseld moet worden. Ten eerste wordt *checkconf* gebruikt om te kijken of het nodig is om de hardware aan te passen, is dat het geval dan wordt *configure* aangeroepen. Door middel van een *h* voor een functie wordt aangegeven dat de hardware versie op de hardware aanwezig is en dus moet worden gebruikt.

```
/* example */
#include <stdio.h>

int confstatus = -1;

void configure(int status) {
  int i,x;
  for (i=0; i<10000; i++) {
    x*=x*i%2;
  }
  confstatus = status;
}

void checkconf(int status) {
  if (confstatus != status) configure (status);
}

void func_c() {
  int i,x;
  for (i=0; i<10000; i++) {
    x+=x*i%2*i%2;
  }
}
```

```
void func_d() {
  int i,x;
  for (i=0; i<10000; i++) {
    x+=x*i%2;
  }
}

void hfunc_c() {
  int i,x;
  for (i=0; i<5000; i++) {
    x+=x*i%2*i%2;
  }
}

void hfunc_d() {
  int i,x;
  for (i=0; i<5000; i++) {
    x+=x*i%2;
  }
}

void func_b() {

  checkconf(0); // configure functions c and d on hardware

  hfunc_c();

  hfunc_d();
}

void func_f() {
  int i;
  int x=0;
  for (i=0; i<10000; i++) {
  }
}

void func_g() {
  int i;
  for (i=0; i<1000; i++) {
  }
}

void func_e() {

  func_f();

  func_g();
}

void func_a() {
  int i;
  for (i=0; i<523; i++) {
    func_b();
```

```
    func_e();
  }
}

void func_i() {
  int i,x;
  for (i=0; i<10000; i++) {
    x*=x*i%2*i%2;
  }
}

void func_j() {
  int i,x;
  for (i=0; i<10000; i++) {
    x*=x*i%2*i%2;
  }
}

void hfunc_i() {
  int i,x;
  for (i=0; i<5000; i++) {
    x*=x*i%2*i%2;
  }
}
void hfunc_j() {
  int i,x;
  for (i=0; i<5000; i++) {
    x*=x*i%2*i%2;
  }
}

void func_h() {
  int i;

  checkconf(1); // configure functions i and j and d on hardware

  for (i=0; i<238; i++) {
    hfunc_i();
    hfunc_j();
    hfunc_d();
  }
}

int main ()
{
  int i;
  for (i=0; i<5; i++) {
    func_a();

    func_h();
  }

  printf("klaar");
```

```
  return 0;
}
```

Stap 6

We gaan nu opnieuw profilen maar dan met de gesimuleerde hardware functies. Hieronder zien we opnieuw de uitvoer van Gprof, waaruit we een aantal waarnemingen kunnen doen. Ten eerste zien we dat de functies die op de hardware uitgevoerd worden inderdaad sneller uitgevoerd worden. Logisch, omdat we dit zelf zo gekozen hebben. Bovendien zien we dat de hardware tien keer van functionaliteit is gewisseld.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 31.72      0.50     0.50     3805     0.13     0.13  hfunc_d
 29.81      0.97     0.47     2615     0.18     0.18  hfunc_c
 19.03      1.27     0.30     1190     0.25     0.25  hfunc_j
 15.22      1.51     0.24     1190     0.20     0.20  hfunc_i
  1.90      1.54     0.03     2615     0.01     0.01  func_f
  1.90      1.57     0.03     2615     0.01     0.01  func_g
  0.63      1.58     0.01       10     1.00     1.00  configure
  0.00      1.58     0.00     2620     0.00     0.00  checkconf
  0.00      1.58     0.00     2615     0.00     0.32  func_b
  0.00      1.58     0.00     2615     0.00     0.02  func_e
  0.00      1.58     0.00        5     0.00   177.12  func_a
  0.00      1.58     0.00        5     0.00   139.60  func_h

                     Call graph


granularity: each sample hit covers 2 byte(s) for 0.63% of 1.58 seconds

index % time    self  children    called     name
                                               <spontaneous>
[1]    100.0    0.00    1.58                 main [1]
                0.00    0.89       5/5           func_a [2]
                0.00    0.70       5/5           func_h [4]
-----------------------------------------------
                0.00    0.89       5/5           main [1]
[2]     55.9    0.00    0.89       5         func_a [2]
                0.00    0.83    2615/2615        func_b [3]
                0.00    0.06    2615/2615        func_e [9]
-----------------------------------------------
                0.00    0.83    2615/2615        func_a [2]
[3]     52.1    0.00    0.83    2615         func_b [3]
                0.47    0.00    2615/2615        hfunc_c [6]
                0.34    0.00    2615/3805        hfunc_d [5]
                0.00    0.01    2615/2620        checkconf [12]
-----------------------------------------------
                0.00    0.70       5/5           main [1]
[4]     44.1    0.00    0.70       5         func_h [4]
                0.30    0.00    1190/1190        hfunc_j [7]
                0.24    0.00    1190/1190        hfunc_i [8]
                0.16    0.00    1190/3805        hfunc_d [5]
                0.00    0.00       5/2620        checkconf [12]
-----------------------------------------------
                0.16    0.00    1190/3805        func_h [4]
                0.34    0.00    2615/3805        func_b [3]
```

```
[5]     31.6    0.50    0.00    3805          hfunc_d [5]
-----------------------------------------------
                0.47    0.00    2615/2615         func_b [3]
[6]     29.7    0.47    0.00    2615          hfunc_c [6]
-----------------------------------------------
                0.30    0.00    1190/1190         func_h [4]
[7]     19.0    0.30    0.00    1190          hfunc_j [7]
-----------------------------------------------
                0.24    0.00    1190/1190         func_h [4]
[8]     15.2    0.24    0.00    1190          hfunc_i [8]
-----------------------------------------------
                0.00    0.06    2615/2615         func_a [2]
[9]      3.8    0.00    0.06    2615          func_e [9]
                0.03    0.00    2615/2615         func_f [10]
                0.03    0.00    2615/2615         func_g [11]
-----------------------------------------------
                0.03    0.00    2615/2615         func_e [9]
[10]     1.9    0.03    0.00    2615          func_f [10]
-----------------------------------------------
                0.03    0.00    2615/2615         func_e [9]
[11]     1.9    0.03    0.00    2615          func_g [11]
-----------------------------------------------
                0.00    0.00       5/2620         func_h [4]
                0.00    0.01    2615/2620         func_b [3]
[12]     0.6    0.00    0.01    2620          checkconf [12]
                0.01    0.00      10/10           configure [13]
-----------------------------------------------
                0.01    0.00      10/10           checkconf [12]
[13]     0.6    0.01    0.00      10          configure [13]
-----------------------------------------------

Index by function name

  [12] checkconf              [9] func_e                  [6] hfunc_c
  [13] configure             [10] func_f                  [5] hfunc_d
   [2] func_a                [11] func_g                  [8] hfunc_i
   [3] func_b                 [4] func_h                  [7] hfunc_j
```

5.5.1 Andere configuraties

In tabel 5.1 zien we een overzicht van een aantal verschillende configuraties en het gevolg daarvan voor de executietijd van het programma. Uit de tabel blijkt dat voor

Configuraties	Totale executietijd	Aantal wissels
-	3.11	0
0 = (c, d) 1 = (d, i, j)	1.58	10
0 = (c, d) 1 = (i, j)	1.74	10
0 = (c, d, i, j)	1.56	1

Tabel 5.1: Verschillende configuraties

deze applicatie één configuratie met daarin de vier uitgekozen functies de laagste executietijd heeft. Het is in ieder geval belangrijk om te concluderen dat we via profiling dus inderdaad wel conclusies kunnen trekken over bepaalde configuraties.

5.6 Conclusie

Uit de experimenten blijkt dat door op een eenvoudige manier profiling te gebruiken we al conclusies kunnen trekken over bepaalde configuraties. Echter omdat we een aantal versimpelende aannames hebben gedaan en een aantal tools nog niet tot onze beschikking hebben is het nog moeilijk om werkelijke applicaties via onze methode te testen.

Hoofdstuk 6

Conclusies en werk voor de toekomst

6.1 Conclusies

Uit het onderzoek blijkt dat traditionele software profilers tot een zekere hoogte gebruikt kunnen worden bij het efficiënt implementeren van een applicatie op een herprogrammeerbaar systeem. We hebben in dit onderzoek Gprof gekozen als profiler. Gprof wordt in de literatuur vaak genoemd en in projecten vaak gebruikt. De profiler Gprof wordt in eerste instantie gebruikt om software applicaties te optimaliseren door bepaalde bottlenecks te vinden en die daarna zodanig aan te passen dat deze geoptimaliseerd worden. Wat wij gedaan hebben is om de profiler op een zelfde manier te gebruiken, maar dan met als doel om een applicatie te kunnen opdelen in software en hardware. We zijn er in geslaagd om Gprof zonder aanpassing toe te voegen in een tool flow. Ten eerste kunnen we via het profilen van een programma de functionaliteit lokaliseren waar het programma de meeste tijd doorbrengt, om daarna die functionaliteit te kunnen verplaatsen naar de hardware, waardoor de applicatie wordt versneld. Bovendien kunnen we door de profiler te gebruiken in een eenvoudig tool flow, eenvoudig verschillende sets van configuraties testen en daarna uit de verschillende versies de versie kiezen waarbij de totale executietijd het kleinst is. We hebben een aantal modellen bedacht voor het in kaart brengen van belangrijke eigenschappen van applicaties in het licht van het maken van configuraties.

6.2 Werk voor de toekomst

Deze scriptie is een aanzet voor het ontwikkelen van een profiler voor herprogrammeerbare systemen. In de toekomst kan deze aanzet verder ontwikkeld worden op een aantal punten.

- het implementeren van de in deze scriptie voorgestelde tools. Hierbij denken we aan een tool voor het schatten van de kosten van functies die uitgevoerd worden in hardware, met daarbij een schatting van de kosten die het herprogrammeren ervan kost en het kopiëren van variabelen van en naar de hardware. Bovendien hebben we in de methode een tool nodig voor het uitrekenen van het benodigde aantal CLB's van een functie in hardware.

- het ontwerpen van efficiënte algoritmes voor het maken van de set van configuraties. De in deze scriptie voorgestelde algoritme kan verder worden uitgewerkt en geoptimaliseerd.
- het automatiseren van de in deze scriptie beschreven methode geeft de gebruiker kans om volledig automatisch de efficiëntste set van configuraties te maken. Waarbij het mogelijk moet zijn dat de gebruiker bepaalde instellingen kan wijzigen zoals het kiezen van de gebruikte algoritmen voor het partitionen.
- het uitbreiden van Gprof waardoor het ook mogelijk is om multithreaded applicaties te kunnen profilen. In de scriptie worden al een aantal moeilijkheden genoemd die we tegenkomen als we ook dit soort applicaties willen gaan gebruiken op herprogrammeerbare systemen.

Bibliografie

[1] www.xilinx.com

[2] G. Vanmeerbeeck, P. Schaumont, S. Vernalde, M. Engels, I. Bolsens, "Hardware/Software Partitioning of embedded system in OCAPI-xl."

[3] Bingfeng Mei, Serge Vernalde, Hugo De Man, Rudy Lauwereins, "Design and Optimization of Dynamically Reconfigurable Embedded Systems."

[4] Yangbing Li *et al.*, "Hardware-Software Co-Design of Embedded Reconfigurable Architectures."Proceedings of Design Automation Conf. (DAC), 1999.

[5] Pieter Algra, "Run-time reconfigurable computing".

[6] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools", In ACM conference on Programmaing Language Design and Implementation, Pages 196-205, Orlando, FL, June 1994

[7] S.L. Graham, P.B. Kessler, and M.K. McKusick, "Gprof: A Call Graph Execution Profiler,"Proceedings of the SIGPLAN '82 Symposium on Compiler Construction (June 1982), pp. 120-126.

[8] C. Young, The Harvard Atom Like Tool Manual (HALT), http://citeseer.nj.nec.com/121315.html

[9] H. Kienle and U. Hölzle, "Introduction to the SUIF 2.0 Compiler System

[10] D.C. Suresh, W.A. Najjar, F. Vahid, J.R. Villarreal, G. Stitt, "Profiling Tools for Hardware/Software Partitioning of Embedded Applications, Proc. of the 2003 ACM SIGPLAN Conf. on Languages, Compilers and Tools for Emedded Systems, San Diego, CA June 2003

[11] Rolf Enzler, Tobias Jeger, Didier Cottet, and Gerhard Tröster, "High-Level Area and Performance Estimation of Hardware Building Blocks on FPGAs."