Communicatiebewuste plaatsing van data in een gedistribueerd rekensysteem

Peter Bertels

Promotor: prof. dr. ir. Dirk Stroobandt
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. Jan Van Campenhout
Faculteit Ingenieurswetenschappen
Academiejaar 2009 - 2010

UNIVERSITEIT
GENT

Dankwoord

Dit proefschrift is tot stand gekomen dankzij vele bijdragen groot of klein, rechtstreeks of onrechtstreeks, van heel wat mensen. Ik wil iedereen van harte bedanken voor de goede raad en voor de verfrissende ideeën waarmee zij dit werk inhoudelijk vooruit hebben gestuwd. Cruciaal was ook het schouderklopje dat jullie me op de juiste momenten hebben gegeven om door te zetten wanneer dit onderzoek even minder goed vlotte.

Een prominente plaats in dit dankwoord is voor mijn promotor, Dirk Stroobandt. Binnen de onderzoeksgroep, heeft hij het kader gecreëerd waarbinnen mijn collega's en ikzelf hun onderzoek in alle vrijheid en zonder zorgen konden aanvatten.

Een eerste aanzet tot de ideeën die ik verder heb uitgewerkt in dit proefschrift, is ontstaan bij het onderzoek van Philippe Faes. Ruim twee jaar heb ik het genoegen gehad om een bureau met Philippe te mogen delen en veel van hem op te steken. In de beginjaren van mijn onderzoek kon ik ook altijd rekenen op Mark Christiaens die de kunst verstaat om steeds de juiste vragen te blijven stellen. Daarnaast heeft Mark me geleerd hoe een vleugje ironie de universitaire bureaucratie een stuk draaglijker kan maken.

In de tweede fase van dit onderzoek heb ik bijzonder veel gehad aan enkele korte, maar zeer diepgaande gesprekken met Erik D'Hollander en Jan Van Campenhout. Met de steun van Wim Heirman heb ik de resultaten van die gesprekken kunnen vertalen in meerdere publicaties die later van kapitaal belang zouden blijken voor het geheel van dit proefschrift.

Mijn dank gaat ook uit naar de andere collega's binnen de onderzoeksgroep die mij rechtstreeks of onrechtstreeks geholpen hebben bij de realisatie van dit werk. In het bijzonder wil ik hier Dries, Harald, Hendrik, Kristof en Michiel vermelden.

Naast het pure onderzoekswerk, heb ik binnen de universiteit

mogen meewerken aan heel wat andere, mooie projecten die me de kans gaven om mezelf verder te ontplooien en om nieuwe, boeiende mensen te leren kennen.

In de eerste plaats denk ik aan de honderden studenten die de practica van het vak Ontwerpmethodologie voor Complexe Systemen (OCS) volgden. Elke week opnieuw maakten zij met hun enthousiasme en betrokkenheid van woensdagnamiddag een hoogtepunt in mijn werkweek waar ik veel energie uit kon putten. Ik kan het vak OCS niet vermelden zonder expliciet Tom, Michiel en Tom te bedanken. Samen hebben we dit vak uit de grond gestampt en we hebben daar zelfs internationaal erkenning voor gekregen.

Aan de universiteit raakte ik bevriend met Karel, Francis, Michiel, Sean en Wim. In april 2009 hebben we samen Dwengo vzw opgericht om experimenteren met microcontrollers te promoten. Het veelzijdige Dwengo-experimenteerbord dat we daarvoor ontwikkelden, is ondertussen op weg om de wereld te veroveren. Bedankt dat ik hieraan mocht en nog steeds mag meewerken.

Ik bedank ook uitdrukkelijk alle mensen die voor of achter de schermen hebben meegewerkt aan de Werkgroep Elektronica, de drie voorbije robotcompetities, de IEEE Student Branch Gent en CenEka voor de vele fijne momenten die ze me bezorgden.

Als dit doctoraat even minder goed vlotte, kon ik steeds terugvallen op mijn chirovrienden bij Chiro WAWW in Brugge, in het Verbond West-Vlaanderen of in de werkgroep Informatica op Kipdorp.

De vrienden van het eerste uur mogen uiteraard niet in dit dankwoord ontbreken: Christophe en Inge, Jeroen, Koen, Karel, Wim en Wouter. Bedankt voor jullie onvoorwaardelijke steun.

Kenmerkend voor een dankwoord is wellicht dat er altijd mensen zijn die over het hoofd worden gezien. Dat heeft waarschijnlijk te maken met het feit dat een dankwoord vaak helemaal op het einde geschreven wordt, wanneer de deadline verraderlijk dichtbij komt. Deze mensen wil ik oprecht danken voor alles wat ze voor mij gedaan hebben. Ik wil hen in het bijzonder danken voor het feit dat ze het me niet kwalijk nemen dat ik hen hier over het hoofd heb gezien.

Tot slot wil ik ook mijn ouders en mijn naaste familie bedanken. Ze steunen me al jaren in alles wat ik doe. Zonder hen was van dit doctoraat wellicht niets in huis gekomen.

Peter Bertels
Gent, 19 mei 2010

Examencommissie

prof. Bart Dhoedt
Vakgroep Informatietechnologie (INTEC)
Faculteit Ingenieurswetenschappen
Universiteit Gent

prof. Daniël De Zutter, voorzitter
Vakgroep Informatietechnologie (INTEC)
Faculteit Ingenieurswetenschappen
Universiteit Gent

prof. Dirk Stroobandt, promotor
Vakgroep Elektronica en Informatiesystemen (ELIS)
Faculteit Ingenieurswetenschappen
Universiteit Gent

prof. Erik D'Hollander, secretaris
Vakgroep Elektronica en Informatiesystemen (ELIS)
Faculteit Ingenieurswetenschappen
Universiteit Gent

prof. Gerard Smit
Computer Architecture for Embedded Systems (CAES)
Faculteit Elektrotechniek, Wiskunde en Informatica
Universiteit Twente

dr. ir. Hugo Kuyken
Algemeen directeur
Dekimo NV

Samenvatting

Communicatie is cruciaal. Het uitgangspunt van dit doctoraatsproefschrift is de vaststelling dat communicatie cruciaal is in gedistribueerde rekensystemen. Door de evolutie naar een steeds groeiend aantal rekenkernen op één enkele chip, wordt communicatie tussen die kernen steeds belangrijker. Binnen een dergelijk gedistribueerd rekensysteem verloopt de communicatie op verschillende niveaus, elk met hun eigen karakteristieke prestatiekenmerken. Draden die op dezelfde kern worden uitgevoerd, kunnen communiceren via gedeeld geheugen in een *cache* of in een extern geheugen dat enkel via het netwerk bereikt kan worden. Draden die op verschillende kernen uitgevoerd worden, kunnen enkel informatie uitwisselen via een relatief trage netwerkverbinding.

De verhouding tussen interne en externe communicatie heeft een fundamentele impact op de uiteindelijke prestaties wanneer een programma wordt opgesplitst over de verschillende rekenkernen in het systeem. Een geschikte verdeling van zowel de berekeningen als de data over de verschillende rekenkernen, is dan ook uitermate belangrijk om tot een efficiënt resultaat te komen.

Communicatie optimaliseren begint met communicatie opmeten. Om een goed beeld te krijgen van hoeveel communicatie er nodig is, heb ik een profileerder ontwikkeld die communicatieprofielen opmeet van Java-programma's. Daarin wordt de informatie over datastromen in het programma gecombineerd met de dynamische oproepgraaf van het programma.

Hierbij wordt een onderscheid gemaakt tussen twee types communicatieprofielen: een eerste dat enkel de inherente communicatie bevat tussen methodes in het programma zonder rekening te houden met de datastructuren die daarvoor gebruikt worden en een tweede profiel dat expliciet de communicatie opmeet tussen methodes en da-

tastructuren (Bertels et al., 2008). Beide profielen hebben hun eigen specifieke toepassingen.

Het eerste communicatieprofiel kan door de ontwerper gebruikt worden als aanzet voor de partitionering van functionaliteit van programma's over meerdere rekenkernen of parallelle draden. De communicatie in een systeem is immers onafhankelijk van de specifieke implementatie en de inherente communicatie opgemeten in de geprofileerde, initiële implementatie, is dan ook een goede maat voor de communicatie in de uiteindelijke implementatie op een gedistribueerd rekensysteem.

Het tweede communicatieprofiel meet de communicatie van en naar datastructuren in Java. Het levert een goed beeld op van het gebruik van Java-objecten doorheen het programma en geeft dus meer informatie over concrete optimalisaties van de datalayout. Dit profiel vormt de basis voor mijn zelflerende algoritme om de plaatsing van objecten te optimaliseren.

Bemonstering voor communicatieprofielen. Profileren brengt onvermijdelijk een vertraging met zich mee. Zeker voor het opmeten van het eerste type communicatieprofiel is de toename van de uitvoeringstijd enorm omdat er een grote boekhouding moet worden bijgehouden van alle lees- en schrijfoperaties en van een schaduwobject voor elk object in het geheugen. De profilering kan fors versneld worden door gebruik te maken van bemonstering, maar dat kan dan weer nadelig zijn voor de kwaliteit van het opgemeten communicatieprofiel.

In dit proefschrift gebruik ik met succes een bemonsteringstechniek uit de oude doos, *reservoir sampling*, om de overlast van het profileren te reduceren met een factor 9, met behoud van een aanvaardbare, en vooraf statistisch vastgelegde, nauwkeurigheid. Dit resultaat werd gevalideerd aan de hand van een hele reeks benchmarkprogramma's uit de SPECjvm benchmark suite (Standard Performance Evaluation Corporation, 1998).

Communicatiebewuste partitionering. De communicatieprofielen vormen de basis voor een communicatiebewuste partitionering van de functionaliteit van programma's waarbij de verhouding tussen interne en externe communicatie als eerste criterium wordt gebruikt.

Hierbij focus ik op een specifieke vorm van partitionering, met name het identificeren van delen van de functionaliteit van een sys-

teem die geschikt zijn om ze af te zonderen van de kern van het systeem. Dit wordt uitgewerkt in het kader van de hardwareversnelde Java Virtuele Machine (JVM) waarbij een klassieke processor de initiële, hoofdpartitie, voor zijn rekening neemt en één of meerdere hardwareversnellers op een Field Programmable Gate Array (FPGA) als co-processor de afgezonderde partities voor hun rekening nemen. Faes et al. (2009) toonden immers aan dat een dergelijke aanpak voor de samenwerking tussen hardware en software interessante resultaten oplevert in de context van Java en de hardwareversnelde JVM.

Ik heb twee methodes uitgewerkt om programma's functioneel te partitioneren op basis van de opgemeten communicatieprofielen: statische partitionering en dynamische partitionering. Beide types hebben hun duidelijke voor- en nadelen en hun specifieke toepassingen.

Statische partitionering is bedoeld als ondersteuning voor een ontwerper die met de hand een programma partitioneert. In de oproepgraaf selecteert men een aantal methodes die geschikt zijn voor hardwareversnelling, bijvoorbeeld omdat ze veel intern parallellisme bevatten, en een aantal methodes die absoluut niet op de hardware uitgevoerd mogen worden, bijvoorbeeld omdat ze complexe controlestructuren bevatten.

De statische partitionering gaat dan incrementeel op zoek naar een communicatiebewuste partitionering die een geschikte grens tussen hardware en software vastlegt. Typisch worden dan niet enkel de aangeduide methodes overgeheveld naar de co-processor, maar worden ook enkele omliggende methodes mee verplaatst. Wanneer die omliggende methodes zeer veel data uitwisselen met de hardwareversnelde methodes, is het immers beter dat die communicatie volledig op de co-processor kan gebeuren.

Dynamische partitionering is een uitbreiding van het statische algoritme voor gebruik in de hardwareversnelde JVM. De Just-in-Time (JIT)-compilatie in deze JVM opent immers perspectieven om hardwareversnelling te beschouwen als een bijkomende stap in het compilatieproces. Dit heeft een aantal voordelen. Zo is in een aantal toepassingen en voor een aantal methodes de verhouding tussen berekeningen en communicatie invoerafhankelijk. Een dynamische aanpak kan hier adequaat op reageren en in de ene uitvoering een

concrete methode wel en in de andere uitvoering niet afleiden naar de co-processor op de FPGA.

Transparant hardware/software co-ontwerp. De hardwareversnelde JVM maakt het mogelijk om op een transparante manier aan hardware/software co-ontwerp te doen op een gemengd platform met een generieke processor en hardwareversnellers op FPGA als co-processor.

Om de prestaties te verbeteren beschikken zowel de hoofdprocessor als de hardwareversneller over hun eigen lokale geheugen. Beide fysieke geheugens zijn verenigd in het gedeelde-geheugenmodel van de JVM. Deze transparante hardwareversnelde JVM beheert de toegang tot alle objecten in het geheugen, ongeacht hun fysieke locatie. De hardwareversneller is meestal via een relatief traag communicatiemedium verbonden met de hoofdprocessor en het hoofdgeheugen. Daarom worden geheugentoegangen naar 'het andere geheugen' erg duur. Die moeten dus zo veel mogelijk vermeden worden.

Communicatiebewust plaatsen van objecten. Een belangrijke taak van de JVM is het zoeken van de meest geschikte geheugenlocatie van elk object in de gedistribueerde Java *heap*. De objecten zitten best in het geheugen dat het dichtst staat bij de processor (of versneller) die de data het vaakst gebruikt. Gegevens die enkel binnen een draad gebruikt worden, zullen zich dan steeds in het lokale geheugen bevinden met een minimalisatie van de externe communicatie tot gevolg. Voor data die gedeeld wordt tussen uitvoeringsdraden wordt gestreefd naar een communicatiebewuste geheugenallocatie.

Ik heb verschillende technieken voorgesteld om communicatiebewust geheugen toe te wijzen. Dynamisch bepaalt de JVM voor elk Java-object, de optimale geheugenlocatie. Mijn zelflerende aanpak probeert de gebruikspatronen voor elk object te schatten op basis van gemeten patronen voor objecten die eerder werden toegewezen (Bertels et al., 2009). Op basis hiervan kan het object geplaatst worden in het meest geschikte geheugen. Mijn strategieën voor het plaatsen van data in de geheugens leiden tot een vermindering van de hoeveelheid externe geheugentoegangen tot 86% (49% gemiddeld) voor de SPECjvm en DaCapo benchmarks (Blackburn et al., 2006; Standard Performance Evaluation Corporation, 1998, 2008).

Abstract

Communication is crucial. The focal point of this PhD thesis is the observation that communication has a critical effect on distributed computing systems. Due to the evolution towards a growing number of computational cores on a single chip, communication keeps gaining importance. Inside such a distributed system, communication occurs at different levels, each with their own performance characteristics. Threads running on the same core can communicate through a shared memory or cache, or through external memory which is reached through a network. Threads running on different cores only have the relatively slow network connection at their disposal as a means of communication.

The balance between internal and external communication has a fundamental impact on the resulting performance when a program is split among the computational cores of a parallel system. An optimal placement of both computation and data is thus of prime importance to obtain efficient results.

Optimising communication starts with measuring communication. To obtain a clear view on the amount of communication needed, I designed a profiler that measures communication streams inside Java programs. It combines the information of data flow graphs with the program's dynamic call graph.

We can distinguish between two types of communication profiles. The first type only considers communication between methods, which is inherent to the program and is not affected by the data structures that are used to communicate through. The second profile measures the communication between methods and data objects explicitly (Bertels et al., 2008). Both types of profile each have their own application domains.

The first type of communication profile can be used by a designer as a starting point for partitioning the functionality of the program into multiple parallel threads. This is possible because the inherent communication of the program, measured on the initial implementation, is independent of the specific implementation. The profile will therefore remain valid in a parallel implementation, and should provide a good initial approximation of the actual communication flows exhibited inside the parallel version of the application.

The second communication profile measures communication to and from data structures in Java. It gives insight into the usage patterns of Java objects throughout the duration of the program and therefore provides more information on concrete optimisations of the data layout. This profile forms the basis of my self-learning data placement algorithm.

Sampling of communication profiles. Profiling invariantly slows down the program under test significantly. Especially for the first type of communication profile, the increase in execution time is enormous due to the large bookkeeping effort that is required for each read and write operation. This profiling can be sped up significantly, however, by making use of sampling, which trades off a lower overhead for a slightly less accurate communication profile.

In this work, I successfully re-use a proven sampling technique, reservoir sampling, which can reduce the profiling overhead by a factor of 9, while maintaining a chosen, and statistically guaranteed level of accuracy. This result was validated on a series of benchmark programs from the SPECjvm benchmark suite (Standard Performance Evaluation Corporation, 1998).

Communication-aware partitioning. The communication profiles form the foundation of a communication-aware partitioning of a program's functionality. The ratio between the amount of internal versus external communication is the first criterion here.

In this work, I focus on a specific type of partitioning, being the identification of functional blocks in a system that can effectively be split off from the core of the system. This approach is applied to the context of a hardware-accelerated Java Virtual Machine (JVM), in which a conventional processor executes the initial, main partition, while one or more hardware accelerators, running on a Field Programmable Gate Array (FPGA), execute the partitions that were

split off. Faes et al. (2009) have shown, in the context of Java and the hardware-accelerated JVM, that this approach of cooperation between hardware and software gives interesting results.

I have worked on both static and dynamic methods of functionally partitioning programs based on communication profiles, each approach with its own advantages and separate application domains.

Static partitioning is meant as support for a designer who is manually partitioning a program. In the call graph, the designer selects a number of methods that are suitable for hardware acceleration, for instance because they contain a lot of internal parallelism, and other methods that are absolutely not suited for execution in hardware, due to for instance their complex control flow.

The method of static partitioning now incrementally searches for a communication-aware partitioning that defines a suitable boundary between hardware and software. Typically, not only those methods that were in the previous step marked for hardware execution are moved towards the hardware, but also several of their neighbouring methods. Indeed, when these neighbouring methods exchange a lot of information with the hardware-accelerated methods, it is much more beneficial to keep this intense communication stream entirely inside the co-processor.

Dynamic partitioning is an extension of the static method for use in the hardware-accelerated JVM. The Just-in-Time (JIT) compiler in this JVM opens a new perspective, in which hardware acceleration can be seen as an additional step in the compilation process. This approach has several advantages. In a number of applications, the balance between communication and computation is input-dependant. A dynamic approach can react adequately, by executing a method in hardware in some executions of the program, but not in others, according to the actual communication profile.

Transparent hardware/software co-design.
The hardware-accelerated JVM enables a transparent way of hardware/software co-design on a hybrid platform with a traditional instruction set processor and hardware accelerators on an FPGA acting as a co-processor.

To improve performance, both the main processor and the hardware accelerator each have their own main memory. Both physical memories are united in the shared-memory model of the JVM. This transparent hardware-accelerated JVM manages accesses to all objects in both these memories, independent of their physical location. Since the hardware accelerator is usually connected to the main processor using a relatively slow communication medium, data accesses to the 'other memory' are very expensive and must be avoided as much as possible.

Communication aware placement of objects. An important task of the JVM is deciding on the most appropriate memory location of each object inside the distributed Java heap. Objects should best reside in the memory that is closest to the processor (or accelerator) that is using the object the most. Data that is used only by one thread will always be in local memory, which results in no external communication. But for shared objects, one must consciously strive for a communication-aware memory allocation.

I have presented several techniques of communication-aware memory allocation. For each Java object, the JVM dynamically determines a suitable memory location. My self-learning approach tries to estimate the future usage patterns for each object, based on the measured usage patterns of objects that are already allocated (Bertels et al., 2009). On this basis, new objects can be placed at the most suitable location. My strategies for data placement can reduce the amount of external memory accesses by up to 86% (49% on average) for the SPECjvm and DaCapo benchmarks (Blackburn et al., 2006; Standard Performance Evaluation Corporation, 1998, 2008).

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

Lijst van afkortingen

AOP	Aspect-georiënteerd programmeren
ASIC	Application-Specific Integrated Circuit
CISC	Complex Instruction Set Computer
CLDC	Connected Limited Device Configuration
CLI	Common Language Infrastructure
CLR	Common Language Runtime
CPU	Central Processing Unit
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
GPU	Graphical Processing Unit
J2ME	Java Platform Micro Edition
JIT	Just-in-Time
JOP	Java Optimised Processor
JVM	Java Virtuele Machine
JVMTI	Java Virtual Machine Tool Interface
MPSOC	Multiprocessor System-on-Chip
OHA	Open Handset Alliance
RISC	Reduced Instruction Set Computer
RVM	Research Virtual Machine
SOC	System-on-Chip
VLIW	Very Long Instruction Word
VM	Virtuele Machine
WCET	Worst-Case Execution Time

Hoofdstuk 1

Inleiding

Software schrijven is niet evident. Parallelle, *soft real-time* systemen bouwen al helemaal niet omdat er steeds hogere eisen worden gesteld aan de functionaliteit van dergelijke systemen en omdat de hardware waarop ze moeten werken almaar complexer wordt.

In de software-wereld heeft men daar wat op gevonden. Vandaag worden steeds meer programma's ontwikkeld in *managed languages* als C# of Java, of zelfs in scripting-talen als Python, Perl of PHP. De programma's in deze talen worden uitgevoerd op hoog-niveau Virtuele Machines (VM's). Deze VM's vormen een abstractielaag tussen het platform waarop de software draait en het programma. De voordelen van deze hogere abstractielaag zijn duidelijk: programma's kunnen eenvoudig overgezet worden op andere platformen en de programmeur hoeft zijn hoofd niet meer te breken over de details van een specifiek platform.

Een extra abstractielaag tussen het programma en het platform biedt heel wat interessante mogelijkheden op het vlak van onder meer beveiliging, *type checking*, overdraagbaarheid, en geheugenbeheer. Al deze aspecten vergen traditioneel heel wat tijd en manuele inspanningen van de programmeur. Doordat de VM veel van dit werk overneemt, kan de ontwikkeltijd sterk teruggedrongen worden. De optimalisaties uitgevoerd door de VM vallen uiteen in twee groepen: statische optimalisaties die de programmeur ook zelf had kunnen uitvoeren en dynamische optimalisaties die enkel tijdens de uitvoering mogelijk zijn. Voor de dynamische optimalisaties is een VM uiteraard onontbeerlijk. Voor de statische groep is de manuele oplossing vaak nog net iets beter, maar dat verschil met optimalisaties door de VM weegt vaak niet op tegen de veel kortere ontwikkel-

tijd: de automatische optimalisaties kosten de programmeur immers geen enkele moeite. Om dezelfde reden heeft manueel geoptimaliseerde assemblercode ondertussen ook de duimen moeten leggen tegen C-code voor de meeste toepassingen.

Dankzij deze vele voordelen hebben *managed languages* de laatste jaren sterk aan belang gewonnen. De overgang van software draaiende op een platform naar software in een virtuele machine heeft een ware revolutie betekend in de software-wereld.

In dit werk zal ik aantonen dat dezelfde principes ook in de context van complexe, ingebedde, parallelle systemen met succes toegepast kunnen worden. De nood aan abstractie is alvast aanwezig: deze systemen zijn immers vaak complexer dan de traditionele softwareplatformen.

De hardware bestaat uit verscheidene parallelle rekenkernen die meestal niet allemaal gelijk zijn. Vele systemen bevatten een centrale processor die aangevuld wordt met gespecialiseerde co-processors of specifieke hardwareblokken met een welomlijnde functionaliteit. Sommige systemen bevatten Field Programmable Gate Arrays (FPGA's) waardoor ook de specifieke hardwareblokken herconfigureerbaar worden. Het platform bestaat dus uit verschillende subsystemen die met elkaar verbonden zijn door een communicatienetwerk op de chip. Het is niet zo eenvoudig om de verschillende rekenkernen efficiënt te gebruiken voor het uitvoeren van de toepassing. Er moet nu immers rekening gehouden worden met de kenmerken van de onderliggende communicatie-infrastructuur. Optimalisatie binnen een VM biedt hier een oplossing.

Bovendien is er een grote diversiteit aan platformen waarmee deze systemen worden gebouwd. De snelle evoluties in de markt en de steeds groter wordende commerciële druk om snel nieuwe toepassingen te ontwikkelen, vragen om kortere ontwikkeltijden en platformonafhankelijkheid. De roep om virutalisatie is dan ook groot.

Dit hoofdstuk vat de uitdagingen van het ontwerp van deze parallelle, ingebedde systemen samen. Daarna wordt dieper ingegaan op een concrete visie over wat VM's kunnen betekenen voor de ingebedde wereld. Vervolgens kom ik tot een samenvatting van de focus en bijdragen van deze scriptie.

1.1 Uitdagingen in hedendaags ontwerp

Verschillende trends beïnvloeden de manier waarop wij leven en daarmee ook de technologie die daarvoor gebruikt wordt. Het aantal ingebedde systemen blijft immens groeien. Vandaag zitten er, bijvoorbeeld, al bijna honderd processors in de gemiddelde auto.

Een eerste trend die ik hier wil aanhalen is die van de steeds veeleisender wordende toepassingen. De consumentenmarkt wordt al enkele jaren getekend door een steeds verder gaande integratie van verschillende toepassingen, in steeds kleiner en krachtiger wordende apparaten. Goede voorbeelden van deze integratie zijn onder meer terug te vinden in het immense gamma aan *smartphones* dat de wereld heeft veroverd, met als voorlopige apotheose de Apple iPhone.

Een tweede evolutie die de ontwerpers voor nieuwe uitdagingen plaatst, gaat over de almaar krachtiger wordende hardwareplatformen die nodig zijn om de veeleisende toepassingen van de toekomst uit te voeren. De complexiteit van deze platformen stijgt doordat er meer functionaliteit wordt geïntegreerd op een enkele chip zoals blijkt uit de evolutie naar System-on-Chip (SOC) en Multiprocessor System-on-Chip (MPSOC). De schalingseffecten die optreden bij deze steeds verder doorgedreven integratie op chip-niveau zijn niet de enige oorzaak van de toename van de complexiteit van het ontwerp. De toenemende complexiteit van de toepassingen die op deze systemen moeten draaien, leidt er immers ook vaak toe dat de afzonderlijke componenten van het systeem op zich ingewikkelder moeten worden.

De technologische verbeteringen die voortvloeien uit beide trends vinden snel hun weg naar de consument. Zo zorgen ze bijvoorbeeld voor een betere gebruikservaring: zowel op audiovisueel gebied als op het vlak van reactiesnelheid. Heel wat toepassingen vandaag vergen *soft real-time* gedrag. Deze snelle wisselwerking tussen wat technologisch haalbaar is en wat de consument verlangt en de toegenomen commerciële druk, heeft geleid tot een steeds korter wordende *time-to-market* waardoor de verschillende versies van deze systemen en platformen elkaar steeds sneller opvolgen.

In deze sectie ga ik dieper in op beide trends en schets ik de gevolgen ervan op systeemontwerp vandaag en in de toekomst. Het zijn deze trends waarop de gevirtualiseerde aanpak die ik in dit werk voorsta, een antwoord zal bieden.

1.1.1 Dynamische, adaptieve en veeleisende toepassingen

Veel van de ingebedde systemen vinden uiteindelijk hun weg naar de consumentenmarkt, bijvoorbeeld in digitale televisie, spelcomputers, set-top boxes of mobiele telefoons, camera's, PDA's, ... Tot voor enkele decennia bestonden de meeste van deze apparaten nog niet, zelfs hun belangrijkste voorlopers waren nog niet geboren. Vandaag komen er wekelijks nieuwe versies op de markt.

Het valt op dat de meeste van deze toepassingen erg veeleisend zijn. Ze bevatten stuk voor stuk signaalverwerkingsalgoritmes die grote hoeveelheden data moeten kunnen verwerken, onder meer videobeelden met een hoge resolutie.

Bovendien zijn niet enkel de prestaties belangrijk, ook het tijdsgedrag speelt een steeds grotere rol. Om de optimale gebruikservaring te leveren moet aan een waslijst van *real-time* vereisten worden voldaan. De gebruiker verwacht immers dat het systeem voldoende snel reageert. Daarbij moet een onderscheid gemaakt worden tussen *hard real-time* of *firm* en *soft real-time* gedrag. Videobeelden mogen niet haperen, maar af en toe mag er eens een deadline gemist worden en een beeld ontbreken, dat valt voor het menselijk oog immers niet op (*soft real-time*). Voor geluid ligt dat anders, het menselijk oor is erg gevoelig waardoor zelfs de kleinste hapering storend werkt (*firm real-time*). Andere toepassingen zijn kritischer en vereisen een onberispelijk tijdsgedrag, bijvoorbeeld omdat het missen van een deadline levensbedreigend kan zijn. In dat geval spreekt men over *hard real-time* vereisten.

De programma's zijn ook veelzijdig in die zin dat ze vaak een hele set van verschillende standaarden moeten ondersteunen bijvoorbeeld voor videodecodering (Amer et al., 2009) of *software-defined* radio (Bougard et al., 2008). Voor het gebruiksgemak moeten ze vlot tussen die standaarden kunnen wisselen, terwijl die toch erg verschillend zijn en soms zelfs een herconfiguratie van het systeem vereisen (Bruneel et al., 2007; Bruneel & Stroobandt, 2008a).

Bovendien is het systeem vaak ook dynamisch. Zo kan het gebeuren dat een nieuwe toepassing wordt ingeladen terwijl een andere toepassing bepaalde delen van het platform al bezet. Het is niet ondenkbaar dat dergelijke wijzigingen een cruciale invloed hebben op de optimale *mapping* van het programma op het platform. Ofwel moet de nieuwe toepassing zich dan tevreden stellen met de resterende *resources* en zich daaraan aanpassen, ofwel moet de volledige *mapping* herzien worden over alle toepassingen heen.

Een mogelijke oplossing hiervoor is een scenario-gebaseerde aanpak waarbij op voorhand wordt geoptimaliseerd voor verschillende gebruiksscenario's waartussen dynamisch gewisseld kan worden (Gheorghita et al., 2009). Maar het is even goed mogelijk dat de gebruiker na aankoop bijkomende software downloadt die *de facto* op voorhand niet gekend was. Scenario's alleen volstaan dan niet meer.

Het blijft dus een belangrijke uitdaging en een immense opgave om efficiënt om te gaan met de dynamiek die vandaag in bijna alle toepassingen vervat zit.

1.1.2 Krachtige, complexe hardware

De voorbije decennia werden gekenmerkt door een ongeziene technologische revolutie onder invloed van de gekende wet van Moore die stelt dat het aantal transistors op een chip verdubbelt om de achttien maanden. Deze wet was oorspronkelijk, in 1965, niet meer dan een empirische vaststelling door Gordon Moore, de latere medeoprichter van Intel (Moore, 1965). Maar al snel werd zijn wet de drijvende kracht achter de elektronica-industrie. De wet van Moore heeft het mogelijk gemaakt om steeds krachtiger systemen te bouwen binnen dezelfde chipoppervlakte. Ontwerpers konden daardoor steeds meer functionaliteit samenpakken op een enkele chip.

Aan deze evolutie kwam enkele jaren geleden een einde. Om verschillende redenen worden chips niet meer complexer gemaakt, maar zoeken ontwerpers tegenwoordig hun heil in het samenwerken van meerdere chips. In de eerste fase werden die chips samengebracht op bord-niveau, vandaag vindt de integratie van een volledige systeem plaats binnen een chip: SOC en MPSOC zijn geboren.

Deze thesis focust precies op deze heterogene multiprocessorsystemen waarin verscheidene componenten met elkaar samenwerken. Deze componenten zijn vaak onderling sterk verschillend: ingebedde processors, Digital Signal Processors (DSP's), FPGA's, Graphical Processing Units (GPU's), enzovoort.

Programma's worden opgesplitst in verschillende taken die worden verdeeld over alle processors in het systeem. Op die manier kan elke taak worden toegewezen aan de meest gespecialiseerde processor of de processor die het meest energiezuinig een bepaalde taak kan uitvoeren.

Communicatie is zeer belangrijk in deze systemen. Terwijl de berekeningen op de chip steeds sneller gebeuren, hinkt de snelheid van

de communicatie achterop. De relatieve *overhead* veroorzaakt door communicatie wordt almaar belangrijker zoals besproken wordt in paragraaf 1.2.1. Verschillende vragen duiken dan op: welk stuk code wordt best uitgevoerd op welk stuk hardware? Welk geheugen is best geschikt om welke data in op te slaan? Welke invloed heeft dit dan op het communicatieverloop en dus op de totale communicatiekost?

De antwoorden op deze vragen zijn erg afhankelijk van de precieze kenmerken van het platform én van de andere software die al op het platform aanwezig is. Belangrijke factoren zijn onder meer het aantal processors, het aantal en de plaatsing van de verschillende geheugens en de eigenschappen van de communicatie-infrastructuur.

Om deze dynamische gegevens op te vangen, leg ik in dit werk een abstractielaag tussen de toepassingssoftware en het platform. Die abstractielaag bestaat uit een Java Virtuele Machine (JVM) die wanneer dat nodig is, de juiste beslissingen probeert te nemen over welke component van het systeem bepaalde functionaliteit moet uitvoeren en waar in de geheugenhiërarchie specifieke Java-objecten best worden opgeslagen.

De basis waar dit werk op voortbouwt is de hardwareversnelde JVM die werd ontwikkeld in het doctoraat van Faes (2008). Hij heeft een bestaande JVM aangepast zodat die bruikbaar wordt als abstractielaag in een systeem met een klassieke processor en een FPGA als co-processor (Faes et al., 2009). De beslissing over de plaatsing van Java-objecten in het geheugen of over het al dan niet gebruiken van de co-processor voor het uitvoeren van specifieke Java-methodes, werd door Faes aan de programmeur zelf overgelaten.

In dit proefschift ligt de focus dan ook op het effectief, dynamisch nemen van deze beslissingen en wel op een communicatiebewuste manier die rekening houdt met de kosten die gepaard gaan met het gebruik van de co-processor(s). Hoewel het niet altijd mogelijk is om dynamisch de optimale beslissingen te nemen, zal ik aantonen dat de VM op basis van de juiste heuristieken erg dicht in de buurt kan komen. Door het feit dat de VM alle dynamische informatie in zich bevat, is de gevonden oplossing vaak wel beter en efficiënter dan een op voorhand statisch bepaalde oplossing.

1.2 Communicatie opmeten en gepast reageren

De titel van deze sectie is meteen de kern, en het algemene concept, van dit werk. In onderliggende tekst toon ik aan hoe communicatie in multiprocessorsystemen opgemeten kan worden en, nog belangrijker, op welke manier de kennis over de communicatiestromen kan worden gebruikt om die communicatie te optimaliseren. In eerste instantie wordt gepoogd de totale hoeveelheid communicatie te verminderen omdat elke vermindering uiteraard een verhoging van efficiëntie inhoudt. In tweede instantie wordt gezocht naar manieren om de *mapping* aan te passen om zo tot een efficiëntere communicatie te komen.

Ik zal hiervoor gebruik maken van virtualisatie. Zoals hoger aangehaald biedt de VM als centraal beslissingscentrum in het systeem alle mogelijkheden om het te optimaliseren. Bovendien blijkt het mogelijk te zijn om in die JVM de communicatie in het systeem op te meten. Op die manier komt alle informatie voor de communicatie-optimalisaties meteen daar terecht waar ze nodig is.

1.2.1 Communicatie is belangrijk

In de ingebedde systemen die ik in dit werk bestudeer, is er interne communicatie tussen de verschillende subsystemen van het grotere geheel. De communicatie verloopt op verschillende niveaus. Eerst en vooral is er communicatie binnen een processor. Maar ook twee draden die uitgevoerd worden op dezelfde processor kunnen met elkaar communiceren door de data die ze delen. Wanneer deze draden niet op dezelfde processor worden uitgevoerd dan moet de data uitgewisseld worden over een of ander communicatiekanaal. In een MPSOC blijft de data in dat geval nog steeds op dezelfde chip, maar soms moet de communicatie ook van de chip, bijvoorbeeld naar een extern geheugen.

Elk van deze niveaus van communicatie brengt een zekere kost met zich mee. Mede omdat de snelheid van de berekeningen in hedendaagse processors steeds blijft stijgen, is de relatieve kost van de communicatie de laatste decennia fors belangrijker geworden.

Om die communicatie in kaart te brengen, worden in hoofdstuk 3 twee manieren voorgesteld om het communicatieprofiel van een programma op te meten. De impact van communicatie tussen verschillende methodes in een programma kan op basis van dit model vrij

goed ingeschat worden. Deze schatting zal in een verdere fase de basis vormen voor de communicatie-optimalisatie.

1.2.2 Meten is weten

De communicatiepatronen kunnen in de tijd veranderen volgens de dynamiek en adaptiviteit van de toepassing, of door interactie tussen verschillende toepassingen. De eerste veranderingen zijn het gevolg van fasegedrag in het programma (Georges et al., 2004; Heirman, 2008; Nagpurkar, 2007). De interactie met andere toepassingen komt bijvoorbeeld voor wanneer nieuwe toepassingen worden ingeladen.

Wanneer plots datatrafiek ontstaat van bijvoorbeeld processor X naar geheugen M, terwijl X en M vrij ver van elkaar verwijderd zijn, dan zou de virtuele machine kunnen beslissen om de data van geheugen M naar geheugen N, veel dichter bij X, te verplaatsen. In hoofdstuk 5 ontwikkel ik technieken om deze beslissing *on-the-fly* te kunnen nemen en toon ik aan dat zelfs relatief eenvoudige technieken de hoeveelheid communicatie drastisch kunnen verminderen en op die manier dus ook de communicatiekost fors doen afnemen.

1.2.3 Dynamisch beslissen: snel en efficiënt

Communicatie opmeten kan op verschillende manieren gebeuren. Men kan één of enkele specifieke uitvoeringen van het programma profileren en de resultaten van deze communicatiemeting dan gebruiken voor optimalisaties van de latere uitvoeringen van het programma. Ofwel kan men de communicatie opmeten tijdens elke uitvoering van het programma. Beide opties hebben voor- en nadelen die ik verder in dit werk zal bespreken.

De profilering beperken tot één of enkele uitvoeringen van het programma, heeft als groot voordeel dat het arbeidsintensieve werk op voorhand gebeurt en er dus geen vertraging of intrusie optreedt tijdens de uitvoering. Nadeel is wel dat de beslissingen ook op voorhand vastgelegd worden. Alle keuzes die de VM moet maken tijdens de uitvoering van het programma zullen dus statische beslissingen zijn, gebaseerd op de beperkte informatie uit één of enkele communicatiemetingen. Merk op dat in dit geval niet alle potentiële optimalisaties worden benut: de VM kan niet inspelen op wat er tijdens de uitvoering gebeurt. Aan de andere kant is er ook een belangrijk voordeel van het volledig op voorhand profileren van het programma, met name dat het programma dan volledig gekend is en gea-

nalyseerd kan worden. De beslissingsalgoritmes hoeven tijdens de uitvoering geen 'leergeld' meer te betalen. Dat is wel het geval wanneer louter dynamisch wordt gewerkt, want dan kan enkel rekening gehouden worden met wat de VM tot dan toe heeft opgemeten.

Profileren tijdens de uitvoering van een programma en dynamisch beslissen, heeft als groot voordeel dat er zo maximaal ingespeeld kan worden op wat er op dat moment gebeurt in het programma. Het systeem kan dan adequaat reageren als er iets zou optreden dat op voorhand, tijdens een eenmalige profilering, niet kon voorzien worden. In die zin past dynamisch beslissen dus beter binnen de context van dynamische en adaptieve systemen.

Er zijn echter twee belangrijke nadelen van de volledig dynamische aanpak. Naast het eerder genoemde leergeld nodig om de optimalisaties tijdens de uitvoering te ontdekken, is er ook de intrusie en vertraging die de profilering onvermijdelijk met zich mee brengt tijdens de uitvoering.

Toch zijn er behoorlijk wat voorbeelden in de literatuur bekend waaruit blijkt dat het voordeel van dynamisch beslissen, namelijk adequaat kunnen inspelen op de concrete uitvoering van een programma, sterk opwegen tegen de eventuele nadelen. In hoofdstuk 2 licht ik enkele voorbeelden toe, onder meer het Dynamo-project bij Hewlett-Packard of de Just-in-Time (JIT) compilatie in vele hedendaagse JVM's.

In hoofdstuk 5 toon ik aan dat dynamisch beslissen een geschikte aanpak is voor het communicatiebewust plaatsen van Java-objecten in het geheugen van de hardwareversnelde JVM van Faes. Mijn dynamische algoritme plaatst de objecten zó in het geheugen dat een grote fractie van de duurste geheugentoegangen, van de hoofdprocessor naar de co-processor of omgekeerd, vervangen wordt door veel goedkopere, lokale geheugentoegangen. De tijdswinst die hierdoor geboekt wordt, overstijgt de vertraging door het profileren ruimschoots.

1.3 Focus en bijdragen van dit werk

Binnen de centrale thesis in dit werk, stip ik graag de drie belangrijkste bijdragen aan die hierna elk in een afzonderlijk hoofdstuk behandeld worden.

De eerste bijdrage, uitgewerkt in hoofdstuk 3, is het efficiënt opmeten van de communicatie in Java-programma's binnen een

communicatiemodel dat zowel de dynamische structuur van de methode-oproepen in het programma als de effectieve communicatie omvat.

Hoofdstuk 4 beschrijft mijn statisch algoritme voor het partitioneren van de functionaliteit van Java-programma's met het oog op een minimalisatie van de communicatiekost. Aan het einde van dit hoofdstuk geef ik een aanzet voor het dynamisch partitioneren van programma's als onderdeel van het JIT-compilatieproces.

De derde bijdrage is mijn zelflerende algoritme voor dynamische en efficiënte geheugenallocatie voor een hardwareversnelde JVM waaraan hoofdstuk 5 is gewijd.

1.3.1 Communicatie efficiënt opmeten

Met het steeds groeiende aantal rekenkernen op één enkele chip, wordt communicatie tussen die kernen steeds belangrijker. Binnen een dergelijk systeem verloopt de communicatie op verschillende niveaus, elk met hun eigen karakteristieke prestatiekenmerken. Draden die op dezelfde kern worden uitgevoerd, kunnen communiceren via gedeeld geheugen in een *cache* of in een extern geheugen dat enkel via het netwerk bereikt kan worden. Draden die op verschillende kernen uitgevoerd worden, kunnen enkel informatie uitwisselen via een relatief trage netwerkverbinding. Dit essentiële verschil tussen interne en externe communicatie en ook de hoeveelheid communicatie tussen verschillende draden, zullen een cruciale impact hebben op de uiteindelijke prestaties wanneer een enkelvoudig programma wordt opgesplitst over verschillende rekenkernen. In dit geval kan een significante prestatiewinst geboekt worden door de toepassing zodanig op te splitsen dat de communicatie tussen draden geminimaliseerd wordt en wanneer men ervoor zorgt dat de communicatiestromen in een niet-uniform netwerk optimaal gespreid worden rekening houdend met de groottes en vertragingen van die stromen. Om dit alles te realiseren is het uiteraard belangrijk dat de communicatiestromen in het programma gekend zijn.

In hoofdstuk 3 beschrijf ik een profileerder voor Java die datastromen tussen methodes kan opmeten. Dit resulteert in een communicatieprofiel van het programma, dat datastroominformatie combineert met de oproepgraaf. Ik meet twee verschillende profielen op: een eerste dat enkel de inherente communicatie bevat tussen methodes in het programma zonder rekening te houden met de datastruc-

turen die daarvoor gebruikt worden en een tweede profiel dat expliciet de communicatie opmeet tussen methodes en datastructuren (Bertels et al., 2008).

Het eerste communicatieprofiel zal in hoofdstuk 4 gebruikt worden als richting voor de handmatige partitionering van functionaliteit van programma's over meerdere rekenkernen of parallelle draden. Het idee hierachter is dat de hoeveelheid communicatie in een systeem onafhankelijk is van de specifieke implementatie en dat de inherente communicatie opgemeten in de geprofileerde, initiële implementatie, dus een goede maat is voor de communicatie in de uiteindelijke implementatie op een parallel platform.

Het tweede communicatieprofiel meet de communicatie van en naar datastructuren in Java. Het levert een goed beeld op van het gebruik van Java-objecten doorheen het programma. Deze informatie wordt in hoofdstuk 5 gebruikt om de plaatsing van deze objecten in een niet-uniforme geheugenruimte te optimaliseren.

In dit werk heb ik een profileerder gebouwd om communicatie te meten in Java-programma's. De keuze voor Java als ontwerpstaal steunt op twee aspecten: enerzijds is Java als taal uitermate geschikt voor programma-analyse en profilering, anderzijds opent Java meteen ook de deur naar een wijdverspreide en toegankelijke VM. Hoewel dit doctoraatsproefschrift een duidelijke keuze maakt voor Java, moet opgemerkt worden dat de hier uitgewerkte concepten overdraagbaar zijn op andere talen of omgevingen, waaronder C en C#. Het volstaat immers om de Java-profiler te vervangen door bijvoorbeeld een x86-profileringsraamwerk.

Profileren brengt altijd een zekere vertraging met zich mee. Zeker voor het opmeten van het eerste communicatieprofiel is de toename van de uitvoeringstijd enorm omdat alle lees- en schrijfoperaties naar het geheugen geïnstrumenteerd moeten worden en omdat van elk Java-object een schaduwobject moet worden bewaard. In sectie 3.4 toon ik aan dat het gebruik van *reservoir sampling* (Vitter, 1985) de overlast van het profileren drastisch reduceert. Deze bemonsteringsmethode maakt het mogelijk om de totale communicatie in het programma te schatten met een op voorhand vastgelegde, aanvaardbare nauwkeurigheid en *overhead*.

Ik heb aangetoond dat deze aanpak goed werkt voor een hele reeks van benchmarkprogramma's (Bertels et al., 2008) uit de SPECjvm benchmark suite (Standard Performance Evaluation Corporation, 1998). Met behulp van *reservoir sampling* kon ik de *overhead*

van de profilering verminderen met een factor 9.

1.3.2 Functionele partitionering van programma's

Er zijn verscheidene redenen om de functionaliteit van programma's op te delen in meerdere partities. Vooreerst is een dergelijke functionele partitionering absoluut noodzakelijk als men verschillende onderdelen van het programma naast elkaar, parallel wil uitvoeren. Ten tweede is het op heterogene platformen met meerdere rekenkernen met elk hun eigen specifieke eigenschappen, belangrijk dat een programma gepartitioneerd is om op die manier elke partitie te kunnen toewijzen aan de meest geschikte rekenknoop. Een derde pluspunt is dat partitionering ook inzicht kan verschaffen in de structuur van een programma, wat nuttig is voor de analyse van programma's.

Cruciaal bij het partitioneren van de functionaliteit van programma's, verder kortweg functionele partitionering genoemd, is de verhouding tussen de hoeveelheid berekeningen in een partitie, meer specifiek de rekentijd die hiervoor nodig is, en de hoeveelheid communicatie tussen deze partitie en alle andere partities. Het heeft immers geen zin om een deel van de functionaliteit van het programma af te zonderen in een aparte partitie als de kost die daarmee gepaard gaat groter is dan de verwachte winst.

Omdat communicatie zo belangrijk is, werk ik twee methodes uit om programma's functioneel te partitioneren op basis van de opgemeten communicatieprofielen: een statische partitionering en dynamische partitionering. Beide types hebben hun duidelijke voor- en nadelen en hun specifieke toepassingen.

Hierbij focus ik op een specifieke vorm van partitionering, met name het identificeren van delen van de functionaliteit van een systeem die geschikt zijn om ze af te zonderen van de kern van het systeem. Dit wordt uitgewerkt in het kader van de hardwareversnelde JVM waarbij een klassieke processor de initiële, hoofdpartitie, voor zijn rekening neemt en één of meerdere hardwareversnellers op een FPGA als co-processor de afgezonderde partities voor hun rekening nemen. Faes et al. (2009) toonden immers aan dat dit een interessante vorm van hardware/software co-ontwerp oplevert.

Mijn algoritme voor statische partitionering is bedoeld als ondersteuning voor een ontwerper die met de hand een programma partitioneert. Het uitgangspunt is het communicatieprofiel van het programma en de oproepgraaf die in dat profiel vervat zit. In de op-

roepgraaf worden een aantal methodes geselecteerd die geschikt zijn voor hardwareversnelling, bijvoorbeeld omdat ze inherent veel parallellisme bevatten dat met een co-processor op de FPGA uitgebuit kan worden. De ontwerper selecteert ook een aantal methodes die zeker niet op de hardware uitgevoerd mogen worden, bijvoorbeeld omdat ze complexe controlestructuren bevatten die typisch minder efficiënt geïmplementeerd kunnen worden op de FPGA. De selectie van deze methodes kan handmatig gebeuren op basis van de inschatting en ervaring van de ontwerper of semi-automatisch met behulp van een prestatieschatter die de haalbaarheid en potentiële winst van hardware-implementatie inschat. De statische partitioneerder zal dan incrementeel op zoek gaan naar een communicatiebewuste partitionering die de optimale grens tussen hardware en software vastlegt. Typisch worden dan niet enkel de aangeduide methodes overgeheveld naar de co-processor, maar worden ook enkele omliggende methodes mee verplaatst.

Het algoritme werd aangepast voor dynamische partitionering op dezelfde hardwareversnelde JVM. De JIT-compilatie in deze JVM opent immers perspectieven om hardwareversnelling te beschouwen als een bijkomende stap in het compilatieproces. Dit heeft een aantal voordelen. Zo is in een aantal toepassingen en voor een aantal methodes de verhouding tussen berekeningen en communicatie invoerafhankelijk. Een dynamische aanpak kan hier adequaat op reageren en in de ene uitvoering een concrete methode wel en in de andere uitvoering niet, afleiden naar de co-processor op de FPGA.

1.3.3 Communicatiebewuste geheugen-allocatie

Hardwareversnellers of andere toepassingsspecifieke co-processors worden gebruikt ter verbetering van de prestaties van rekenintensieve programma's. Belangrijke versnellingen werden bereikt in verscheidene toepassingsgebieden: multimedia (Eeckhaut et al., 2007; Lysecky et al., 2006; Vassiliadis et al., 2004), bio-informatica (Faes et al., 2006; Maddimsetty et al., 2006) en vele andere toepassingen waar kleine, maar rekenintensieve algoritmes met een voldoende graad van intern parallellisme worden gebruikt.

In deze aanpak wordt deze hardwareversnelling transparant ingebouwd in de JVM. Op die manier verloopt de versnelling transparant voor de programmeur en hoeft die zich niet bezig te houden met het beheer van de communicatie en de synchronisatie tussen de

hardware en de software (Faes et al., 2004).

Het kan zelfs nog een stap verder gaan als de hardwareversnellers tijdens de uitvoering worden geïnstantieerd. Op dat ogenblik wordt hardwareversnelling een onderdeel van de JIT-compilatie van het programma in de VM. De hardwareconfiguratie kan dan opgeladen worden uit een bibliotheek (Borg et al., 2006) of zelfs *on-the-fly* gegenereerd (Beck & Carro, 2005; Lysecky et al., 2006).

Om de prestaties te verbeteren beschikken zowel de hoofdprocessor als de hardwareversneller over hun eigen lokale geheugen. Beide fysieke geheugens zijn verenigd in het gedeelde-geheugenmodel van de JVM. Deze transparante hardwareversnelde JVM beheert de toegang tot alle objecten in het geheugen, ongeacht hun fysieke locatie. De plaatsing van de objecten in een van beide geheugens heeft nu uiteraard een belangrijke invloed op de totale systeemprestaties. omdat de hardwareversneller meestal via een relatief traag communicatiemedium verbonden is met de hoofdprocessor en het hoofdgeheugen, worden geheugentoegangen naar 'het andere geheugen' erg duur. Die moeten dus zo veel mogelijk vermeden worden.

De plaatsing van objecten in de gedistribueerde Java *heap* is nu een belangrijke taak van de JVM. Die moet de objecten toewijzen aan het geheugen dat het dichtst staat bij de processor (of versneller) die de data het vaakst gebruikt. Gegevens die enkel binnen een draad gebruikt worden, zullen zich dan steeds in het lokale geheugen bevinden. Op die manier wordt de externe communicatie geminimaliseerd. Voor data die gedeeld wordt tussen uitvoeringsdraden wordt gestreefd naar een communicatiebewuste geheugenallocatie.

Het probleem van de optimale geheugentoewijzing van objecten kan niet opgelost worden door statische analyse alleen omdat met statische technieken enkel conservatieve uitspraken kunnen worden gedaan over het feit of objecten al dan niet enkel door een draad gebruikt worden. Sommige objecten zullen ook door meerdere draden gebruikt worden en dan is het belangrijk de verhouding van het gebruik door de ene of de andere draad te kennen. Omdat die verhouding vaak invoerafhankelijk is, kan die dus ook niet op voorhand statisch bepaald worden. Een *runtime*-aanpak wordt helemaal onvermijdelijk wanneer men een dynamisch systeem beschouwt waarin functionaliteit gemigreerd kan worden van de *general-purpose* hoofdprocessor naar een hardwareversneller tijdens de uitvoering van het programma.

Ik stel verschillende technieken voor om communicatiebewust

geheugen toe te wijzen in hoofdstuk 5. Voor elk Java-object, wordt de optimale geheugenlocatie bepaald op basis van het gebruikspatroon van dit object en eerder aangemaakte, gelijkaardige objecten. Ik vergelijk verschillende strategieën om dynamisch objecten aan geheugens toe te wijzen, waaronder het zelflerende algoritme dat ik in het kader van dit doctoraat heb uitgewerkt. Deze zelflerende aanpak probeert de gebruikspatronen voor elk object te schatten op basis van gemeten patronen voor objecten die eerder werden toegewezen (Bertels et al., 2009). Mijn strategieën voor het plaatsen van data in de geheugens leiden tot een vermindering van de hoeveelheid externe geheugentoegangen tot 86% (49% gemiddeld) voor de SPECjvm (Standard Performance Evaluation Corporation, 1998, 2008) en DaCapo benchmarks (Blackburn et al., 2006).

Hoofdstuk 2

Een virtuele machine als abstractielaag

Cruciaal voor de concepten die ik in dit proefschrift voorstel en uitwerk is het idee van de Virtuele Machine (VM) als intelligente abstractielaag tussen een programma en het onderliggende platform. De bedoeling van deze virtuele abstractielaag is dat ze de mogelijkheden van het platform zo goed als mogelijk benut. In de verdere hoofdstukken focus ik voornamelijk op optimalisaties die de communicatie in het systeem verbeteren. De beslissingen die de VM moet nemen voor deze optimalisaties, zullen gebaseerd zijn op metingen van de communicatiestromen in het systeem.

Dit hoofdstuk geeft een overzicht van VM's in het algemeen en van de werking van de Java Virtuele Machine (JVM) in het bijzonder. Er wordt ingegaan op de specifieke kenmerken die nodig zijn om communicatie-optimalisatie mogelijk te maken.

Na enkele voorbeelden van VM's in de wereld van complexe ingebedde systemen, komt ook de hardwareversnelde JVM van Faes, waarop ik in dit werk voortbouw, uitgebreid aan bod. Deze JVM regelt de vlotte samenwerking tussen generieke processor en een coprocessor in herconfigureerbare hardware. Het is precies in een dergelijke JVM dat communicatie-optimalisatie onontbeerlijk is.

2.1 Verschillende gezichten van virtualisatie

De geschiedenis van de computerarchitectuur heeft heel wat ideeën voortgebracht voor virtualisatie. Verscheidene implementaties van

VM kwamen op die manier tot stand, vaak vanuit diverse invalshoeken. Dit heeft geleid tot verschillende types VM's die elk sterk staan in een specifiek toepassingsdomein waarvoor ze de juiste specifieke kenmerken hebben. Hier geef ik een kort overzicht van virtualisatie en de classificatie van de verschillende VM's om tot slot te komen tot een overzicht van de sterke en minder sterke punten van VM's. Voor een uitgebreide beschrijving verwijs ik naar het boek van Smith en Nair (2005).

2.1.1 Virtualisatie van een volledig systeem

Een van de eerste VM's zag het levenslicht bij IBM halfweg de jaren 1960. IBM had immers vastgesteld dat klanten nauwelijks nog nieuwe hardware kochten, omdat dat telkens een gigantische rompslomp met zich meebracht. Bijna alle software moest herschreven worden bij de overgang van de ene *mainframe* naar zijn opvolger. Terwijl achterwaartse compatibiliteit vandaag vaak als evident wordt beschouwd, was het in die tijd immers nog de absolute uitzondering, zowel bij IBM als bij zijn concurrenten. Maar IBM zag als eerste het licht en ging samen met zijn nieuwe *mainframes* ook VM's ontwerpen die in staat waren om alle oudere software probleemloos uit te voeren.

IBM ontwikkelde in die jaren de IBM System/360 Model 40 (Adair et al., 1966). De doelstelling was dubbel: enerzijds overdraagbaarheid van de code en compatibiliteit met vroegere platformen, anderzijds ook het mogelijk maken van *time sharing* om zo verschillende gebruikers tegelijk toegang te geven tot het systeem. De VM die hiervoor ontwikkeld werd was een zogenaamde systeem-VM die een abstractielaag vormt tussen de hardware enerzijds en het besturingssysteem en de daarop draaiende software anderzijds. Het systeem was zo opgebouwd dat bovenop de abstractielaag tegelijkertijd verscheidene besturingssystemen konden werken, volledig gescheiden van elkaar. Op die manier werd *time sharing* mogelijk op een veilige manier, gebruikers konden elkaars werk niet beïnvloeden.

De overdraagbaarheid van de software werkte in twee richtingen: zowel naar het verleden toe als naar de toekomst. De System/360-modellen waren allemaal voorzien van een specifieke emulatiemodus waarin oudere software van voor het System/360-tijdperk uitgevoerd konden worden. De doelstelling voor de toekomst was dat alle latere systemen die IBM zou bouwen via een aangepaste

systeem-VM de instructieset van de System/360 zouden kunnen blijven gebruiken. De z/VM die IBM vandaag levert bij zijn IBM zSeries *mainframes* steunt nog steeds op dezelfde basisconcepten.

2.1.2 Dynamische optimalisatie

Een heel ander voorbeeld is dat van het Dynamo-project bij Hewlett-Packard (Bala et al., 2000). Daarin werd een VM gebruikt om UNIX-programma's voor de HP PA8000-processor tijdens de uitvoering te optimaliseren.

Alle instructies worden door het systeem geïnterpreteerd en alle sprongpaden in het programma worden geprofileerd. Van elk basisblok wordt bijgehouden hoe vaak het werd uitgevoerd. Wanneer blijkt dat een basisblok erg vaak wordt uitgevoerd —vaker dan een zekere drempelwaarde— dan wordt het blok, of worden de opeenvolgende blokken, geoptimaliseerd, volgens het principe van een conventionele Just-in-Time (JIT) compiler.

De optimalisaties die Dynamo uitvoert zijn zeer uiteenlopend. Het gaat onder meer om het inlijnen van code voor een beter gebruik van de *cache*, of het wegoptimaliseren van sprongen die altijd genomen worden, of optimalisaties binnen de oproep van dynamische bibliotheken. Ze verschillen bovendien van optimalisaties die de compiler zelf zou kunnen uitvoeren. Ten eerste omdat Dynamo alle uitgevoerde instructies kan bekijken (en dus optimaliseren), zowel die van het programma zelf als die van externe bibliotheken die door het programma gebruikt worden. Ten tweede omdat Dynamo beschikt over dynamische informatie van de concrete uitvoering van het programma, zoals statistieken over het spronggedrag, die voor veel programma's heel erg invoerafhankelijk zijn waardoor een statische compiler er geen rekening mee kan houden.

Speciaal aan dit voorbeeld is dat de VM een abstractielaag vormt tussen twee identieke instructiesets, tussen het programma en het besturingssysteem in. Eigenlijk is de VM in deze toepassing in principe overbodig, ze is niet essentieel voor de uitvoering. Het programma kan perfect rechtstreeks werken op het platform met direct contact met het besturingssysteem.

Hewlett-Packard heeft in deze studie aangetoond dat de snelheidswinst door dynamische optimalisatie ruimschoots het efficiëntieverlies compenseert dat veroorzaakt wordt door de bijkomende, en principieel overbodige, abstractielaag van de VM.

2.1.3 Besturingssystemen virtualiseren

Een andere type systeem-VM ontstaat wanneer de virtuele abstractielaag ligt tussen twee besturingssystemen: een gastheer en een gast. Dit type VM maakt het mogelijk om bijvoorbeeld programma's die voor Windows geschreven zijn uit te voeren op een computer die is uitgerust met het Linux-besturingssysteem of op een computer met Mac OS X.

Deze vorm van virtualisatie heeft commercieel enorm aan belang gewonnen sinds de recente opkomst van *multicore* computers. Via virtualisatie is het immers mogelijk om verscheidene besturingssystemen naast elkaar op dezelfde computer te draaien. Dat kan op een computer met één enkele processor, maar het gaat uiteraard efficiënter op een *multicore* waarbij elke VM zijn eigen processor kan aanspreken.

Door deze vorm van virtualisatie kunnen bedrijven en organisaties flink besparen op hun IT-budget, zowel in de aankoop van nieuwe machines als in het onderhoud en op de elektriciteitsrekening. De gemiddelde belasting van veel servers is immers relatief laag zodat hun taken overgenomen kunnen worden door virtuele servers waarvan er meerdere op één enkele fysieke server kunnen draaien (Microsoft, 2008; VMWare, 2008).

2.1.4 Transmeta Crusoë

Ook vanuit een heel andere invalshoek bleek een VM bijzonder nuttig. Dat was alvast de mening van de kleine processorfabrikant Transmeta toen die in 2000 de concurrentie wou aangaan met het veel grotere Intel. Transmeta wou een processor maken die met een veel lager energieverbruik dezelfde programma's kon uitvoeren als de populaire x86-processors van Intel (Halfhill, 2000).

Om dat doel te bereiken werd gekozen voor een Very Long Instruction Word (VLIW) architectuur (Fisher, 1983) waarin verscheidene instructies parallel uitgevoerd worden, wat typisch leidt tot een efficiënter gebruik van de *resources* op de chip.

Samen met de processor werd een VM ontworpen die de vertaalslag zou maken tussen de complexe x86-instructieset en de eenvoudiger, maar parallelle VLIW-instructies. Deze taak van de VM was dubbel: enerzijds een vertaling maken tussen de CISC-instructies van de x86 en de eenvoudigere RISC-instructies van de VLIW en anderzijds in de originele instructiestroom op zoek gaan naar voldoen-

de onafhankelijke instructies die dan als atomen in een molecule samengepakt kunnen worden in de bredere, parallelle instructies van de VLIW.

De Transmeta Crusoë werd op die manier de eerste, populaire laag-vermogen-processor voor mobiele toepassingen die tegelijk in grote mate compatibel was met alle bestaande *legacy code*.

2.1.5 Virtualisatie voor platformonafhankelijkheid

In de loop der jaren werden ook VM's ontwikkeld met als specifieke doelstelling om de overdraagbaarheid van programma's te garanderen. De VM vormt dan een abstractielaag tussen een concrete programmeertaal en het besturingssysteem. Hierbij biedt de VM de architectuur van een virtuele processor of zelfs een virtueel platform aan aan de programmeur.

Op die manier hoeft software maar een keer geschreven en gecompileerd te worden, voor die virtuele architectuur, en moet bij overgang naar een nieuw platform enkel de VM herschreven worden. Merk op dat de instructieset van de VM in dit geval meestal geen bestaande instructieset is, maar een instructieset die specifiek is ontworpen als platformonafhankelijke tussenlaag tussen het programma en een zo breed mogelijke set van doelplatformen.

Een van de eerste voorbeelden van deze klasse van VM's is de P-code-machine die werd ontwikkeld voor het snel en efficiënt, platformonafhankelijk uitvoeren van Pascalprogramma's (Nori et al., 1975; Overgaard, 1980). Pascal-compilers vertaalden de programma's naar P-code die dan geïnterpreteerd werd door een emulator van een hypothetische processor met de P-code als instructieset.

Om de efficiëntie van de interpreteer-stap te verhogen, werd bij het ontwerp van de VM en de bijbehorende P-code-instructieset gekozen voor een stapelarchitectuur. Een belangrijk voordeel hiervan is dat het compilatieproces om P-code om te zetten naar rechtstreeks uitvoerbare machinecode hierdoor vrij eenvoudig kon blijven. Een ander voordeel is dat code voor een stapelarchitectuur typisch relatief compact is en dat was in de tweede helft van de jaren 1970 zeker ook belangrijk.

2.1.6 Een intelligente uitvoeringsomgeving

Een VM met een specifiek voor virtualisatie ontworpen instructieset, is niet enkel interessant als vehikel om platformonafhankelijkheid

te creeëren. Het samen ontwerpen van de VM met de instructieset, maakt het ook mogelijk om in de architectuur van de VM extra functionaliteit te voorzien waar de programmeur kan op steunen tijdens de uitvoering van zijn programma.

De kernconcepten van mijn doctoraat zijn uitgewerkt in de context van Java en de JVM. Naast de gebruikelijke ondersteuning van de JVM voor automatisch geheugenbeheer en allerlei controles die ervoor zorgen dat de uitvoering correct en betrouwbaar gebeurt, zoals bijvoorbeeld *type checking* en *array bound checking*, regelt de hardwareversnelde JVM ook de soepele samenwerking van de hoofdprocessor met één of meerdere hardwareversnellers op een Field Programmable Gate Array (FPGA) die als co-processor fungeren. Op die manier wordt de JVM dus een intelligente uitvoeringsomgeving.

Omdat de JVM uitvoerig aan bod komt in paragraaf 2.4, houd ik het hier bij de VM's die door James Gosling als belangrijkste inspiratiebron wordt genoemd voor de ontwikkeling van zijn JVM (Allman, 2004): de Smalltalk-VM met ingebouwde JIT-compiler.

Smalltalk zag het levenslicht in de jaren 1970 in het Palo Alto Research Center van Xerox (Kay, 1996). Smalltalk wordt door velen beschouwd als de eerste object-geöriënteerde programmeertaal en heeft vanuit die optiek heel wat invloed gehad op de ontwikkeling van onder meer C++, Java en Ruby.

Het concept object-oriëntatie werd in Smalltalk dan ook consequent doorgetrokken. Werkelijk alles[1] in Smalltalk is een object, dat wil zeggen, een concrete instantie van een klasse. Die klassen zijn zelf objecten van het type *metaclass*. Smalltalk kent, in tegenstelling tot bijvoorbeeld C++, geen primitieve datatypes, ook *integer* en *boolean* zijn objecten. Code-fragmenten worden ook als objecten beschouwd die via *message passing* kunnen worden doorgegeven tussen objecten.

De eerste versies van Smalltalk bleven een tijd lang een interne ontwikkeling van Xerox. Het is pas met Smalltalk-80 dat Smalltalk een ruimere wereldwijde verspreiding en ook erkenning kreeg. Het is die versie waarvoor enkele jaren later een VM met een efficiënte, meerstraps-JIT-compiler werd ontwikkeld (Deutsch & Schiffman, 1984).

Hoewel Smalltalk tegenwoordig veel minder gebruikt wordt, leven de concepten vandaag nog voort. Talen als Java en C# zijn net als Smalltalk object-geöriënteerd en de meest verpreide VM's voor Java

[1] Alles, behalve de variabelen, die echter wel steeds verwijzen naar een object.

en C#, respectievelijk de JVM (Lindholm & Yellin, 1999) en Microsofts Common Language Runtime (CLR) (Box et al., 2002), bouwen voort op ideeën die in de Smalltalk-VM al aanwezig waren.

2.2 Just-in-Time-compilatie

Na een kort overzicht van enkele concrete voorbeelden van VM's, beschrijf ik in deze paragraaf de belangrijkste kenmerken van JIT-compilatie, soms ook dynamische compilatie genoemd. Voor een meer gedetailleerd overzicht verwijs ik naar het werk van Aycock (2003).

Voor de eigenlijke uitvoering van de programmacode bestaan er verschillende technieken: compileren en interpreteren. Bij compileren wordt de code vertaald (gecompileerd) naar een vorm die later rechtstreeks uitgevoerd kan worden op een specifieke machine. Een *interpreter* slaat die voorbereidende stap helemaal over en doet hetzelfde werk, de vertaling naar machinecode, pas tijdens de uitvoering van het programma. Een JIT-compiler bevindt zich in het midden tussen deze twee uitersten en combineert op die manier het beste van twee werelden.

De efficiëntie van een VM met een JIT-compiler benadert de efficiëntie van statisch gecompileerde programma's. Minder vaak uitgevoerde delen van het programma worden geïnterpreteerd en daardoor dus relatief traag uitgevoerd, maar voor hete code zal de JIT-compiler tijdens de uitvoering efficiënte machinecode genereren. De JIT-compiler bevat bovendien een *cache* voor veelvuldig uitgevoerde code, zodat niet alles telkens opnieuw geïnterpreteerd of gecompileerd moet worden. Het deel van het programma dat het meest impact heeft op de totale uitvoeringstijd, de meest frequent uitgevoerde code, ondervindt daardoor nauwelijks nog hinder van de trage *interpreter*.

Toch behoudt de JIT-compiler de flexibiliteit van een *interpreter*. De code blijft platformonafhankelijk en daardoor makkelijk overdraagbaar. Bovendien heeft de JIT-compiler kennis van wat er concreet, tijdens de uitvoering, gebeurt: hij kan voor het compileren gebruik maken van dynamische informatie die een *offline* compiler niet heeft.

Het begon allemaal met de eerste generatie *interpreters* die elke instructie afzonderlijk naar machinecode vertaalden (Calingaert, 1979). In een geoptimaliseerde versie werden veelvoorkomende vertalin-

gen bijgehouden in een *cache* (Lang et al., 1986). Dit bracht al een zekere versnelling met zich mee, maar de snelheid van rechtstreeks uitgevoerde machinecode lag nog veraf. Van extra optimalisatie was geen sprake omdat er nooit verder werd gekeken dan een enkele instructie.

De tweede generatie *interpreters* werkte op basis van codeblokken in plaats van individuele instructies. Op die manier werden heel wat bijkomende optimalisaties mogelijk (May, 1987). Zo werden sprong-instructies uit de originele broncode vertaald naar machinecode-spronginstructies die op de juiste manier springen tussen vertaalde code. Dit zorgt voor een belangrijke snelheidswinst in de *interpreter*.

Pas vanaf de derde generatie kan men echt van een JIT-compiler spreken. Op dat ogenblik worden de codeblokken immers niet meer snel-snel vertaald en gecachet, maar worden ze echt gecompileerd en geoptimaliseerd. Dit vergt echter meer tijd dan een eenvoudige vertaling. Hierbij moet dan een afweging gemaakt worden tussen de tijd die nodig is voor het efficiënt compileren van een codeblok en de tijd die gewonnen wordt door een meer efficiënte uitvoering.

De meeste JIT-compilers werken met een meertrapscompilatie: op het ogenblik dat een codeblok de JIT-compiler binnenkomt, wordt die code geïnterpreteerd of snel-snel vertaald. Bij een volgende uitvoering van hetzelfde blok wordt dan de vertaalde versie uit de *cache* opgehaald. Pas als een blok voldoende vaak uitgevoerd werd, en het dus de moeite loont om dat blok te optimaliseren, wordt de tweede compilatietrap gebruikt om efficiëntere code te genereren.

De Jikes Research Virtual Machine (RVM) (Alpern et al., 2000) werkt bijvoorbeeld met een *baseline compiler* en een *optimising compiler* met drie optimalisatietrappen (Burke et al., 1999). In elke stap wordt betere en efficiëntere code gegenereerd. Maar dat betekent ook dat er in elke compilatiestap meer tijd wordt gespendeerd om meer optimalisaties te vinden en door te voeren om effectief efficiëntere code te genereren.

2.3 Sterke punten van virtualisatie

De eerder genoemde voorbeelden illustreren meteen ook de sterke punten van VM's die ik hier nog even samenvat.

2.3.1 Platformonafhankelijkheid

Met een VM is het mogelijk om een systeem platformonafhankelijk te maken. Dezelfde software kan draaien op andere hardware zolang er voor dat platform maar een VM bestaat.

Soms gaat het over VM's die een specifieke, 'echte' architectuur emuleren omwille van compatibiliteit met andere reeds bestaande systemen, zoals bij IBM's System/360 en de Transmeta Crusoë.

In andere gevallen gaat het over een nieuwe architectuur van een 'hypothetische' machine. Deze VM's situeren zich aan de andere kant van het spectrum bij de hoog-niveauprogrammeertalen. Samen met de programmeertaal Java werd ook de JVM en zijn architectuur ontwikkeld. Het grote voordeel hiervan was dat de taal ontwikkeld kon worden, los van de specifieke eigenaardigheden van een concrete, bestaande architectuur. De nieuwe taal Java en de nieuwe architectuur, de JVM, sluiten perfect bij elkaar aan.

Zowel VM's die samen met een specifiek platform werden ontwikkeld als VM's die samen met een concrete taal werden ontwikkeld, verlossen de programmeur van de last om rekening te houden met de onderliggende hardware.

2.3.2 Dynamische optimalisatie

Het uitvoeren van software binnen een VM maakt het mogelijk om in te spelen op situaties die zich voordoen tijdens een specifieke uitvoering. Een JIT-compiler kan daardoor delen van het programma efficiënter compileren dan een statische compiler op voorhand zou kunnen.

Het meest extreme voorbeeld hiervan was het Dynamo-project van Hewlett-Packard waarbij die bijkomende optimalisaties de enige reden vormden voor het gebruik van een VM. In het geval van de JVM en CLR zijn de extra optimalisaties voornamelijk bedoeld om het efficiëntieverlies door virtualisatie te compenseren.

2.3.3 Afscherming en verhoogde veiligheid

De VM levert een afscherming tussen het programma en het besturingssysteem die de mogelijkheden van software om 'buiten de lijntjes te kleuren' sterk kan inperken.

Typisch zorgt de VM ervoor dat het programma enkel zijn eigen data kan lezen of wijzigen waardoor programma's elkaar niet meer

(bedoeld of onbedoeld) kunnen beïnvloeden behalve via specifiek daarvoor voorziene kanalen. Net zoals een besturingssysteem ervoor zorgt dat programma's niet in elkaars geheugen kunnen schrijven, bewaakt de VM het geheugengebruik binnen een programma. Veel VM's hebben immers zicht op informatie waar het besturingssysteem geen rekening mee kan houden. Zo weten ze bijvoorbeeld welke data in het programma *public* of *private* zijn.

Een besturingssysteem heeft zicht op geheugenregio's die door een programma in gebruik worden genomen. Ook op dat vlak kan een VM veel preciezer te werk gaan. De VM kent de structuur van de objecten in het geheugen en kan dus afdwingen dat het programma die objecten correct gebruikt en bijvoorbeeld niet voorbij de grenzen van een *array* schrijft of data wegschrijft die niet het correcte datatype heeft. Deze controles gebeuren ofwel tijdens de uitvoering van het programma ofwel op het ogenblik dat de code binnenkomt in de VM in een afzonderlijke verificatiestap.

Die afscherming is ergens wel beperkend, maar in veel gevallen werkt ze vooral bevrijdend omdat ze het werk van de programmeur eenvoudiger maakt. Hij kan zelf niet meer belissen op welke locatie in het geheugen bepaalde gegevens worden opgeslagen, maar aan de andere kant hoeft de programmeur zich ook geen zorgen meer te maken over die geheugenlocatie. De programmeur kan zelf niet meer bepalen welk deel van het geheugen vrijgemaakt mag worden, maar aan de andere kant hoeft de programmeur zich hier ook niets meer van aan te trekken. In de praktijk zijn de beperkingen nauwelijks beperkend en ligt de nadruk vooral op het bevrijdende en vereenvoudigdende aspect.

2.3.4 Eenvoudig en efficiënt geheugenbeheer

Programma's uitvoeren in een virtuele omgeving maakt het mogelijk om het geheugenbeheer sterk te automatiseren. Door automatisch geheugenbeheer in de VM hoeft de programmeur zich zelf geen zorgen meer te maken over het alloceren en vrijgeven van geheugen.

Het grote voordeel van automatisch geheugenbeheer is dat er minder fouten optreden omdat het bij definitie correct werkt, *correct by construction*. De *segmentation fault* die kan ontstaan als er iets foutloopt bij geheugenallocatie of het vrijgeven van geheugen, behoort definitief tot het verleden. Het systeem zal objecten in het geheugen die niet meer bereikbaar zijn, i.e. waar geen referenties meer naar be-

staan, automatisch wissen, waardoor geheugenlekken in programma's niet meer kunnen voorkomen.

Hierbij moet echter een afweging gemaakt worden tussen eenvoud en efficiëntie. Het spreekt voor zich dat een goede programmeur ook zonder automatisch geheugenbeheer correcte programma's kan schrijven. Misschien kan hij het geheugen nog efficiënter beheren dan de VM dat zou doen. Als men ook de tijd in rekening brengt die deze programmeur nodig heeft om zelf het geheugenbeheer in zijn programma te optimaliseren, dan blijkt automatisch geheugenbeheer in veel gevallen toch de beste keuze. De efficiëntie van automatisch geheugenbeheer benadert immers vaak die van handmatig geheugenbeheer.

2.4 Java Virtuele Machine

De programmeertaal Java en de JVM werden begin jaren 1990 ontwikkeld bij Sun (Lindholm & Yellin, 1999). In 1991 werd gestart met de ontwikkeling van een nieuwe taal die toen nog Oak heette. Vlak voor de lancering in 1994 werd ze omwille van een probleem met het recht op de naam hernoemd tot Java.

In die eerste jaren waren programma's geschreven in Java tot 10 keer trager dan vergelijkbare programma's in C of C++. Door het gebruik van een JIT-compiler is dat ondertussen lang verleden tijd: voor sommige toepassingen is Java vandaag zelfs sneller (Lewis & Neumann, 2003; Reinholtz, 2000) net omdat de JIT-compiler dynamisch een aantal optimalisaties kan doorvoeren die een statische compiler niet gebruikt omdat hij op voorhand niet kan inschatten of ze de moeite lonen.

Java-programma's worden in twee stappen gecompileerd. Eerst vertaalt een statische Java-compiler het programma naar platformonafhankelijke Java-bytecodes. Tijdens de uitvoering zullen die bytecodes door een dynamische compiler, de JIT-compiler van de JVM, worden vertaald naar de rechtstreeks uitvoerbare machinecode van het platform. Om de JVM te begrijpen, is het dus belangrijk de opbouw en kracht van de Java-bytecodes en de mogelijkheden van Java te leren kennen.

2.4.1 Java-bytecodes

De instructieset van de JVM, de Java-bytecodes, is erg krachtig. Achter deze instructies, de elementaire bouwblokjes van Java-programma's, schuilt immers vaak heel wat functionaliteit. Veel meer dan het geval is bij typische machinecode.

Als voorbeeld, ontleend aan Faes (2008), haal ik graag de `new`-bytecode aan, die een nieuw object aanmaakt van een bepaalde klasse. Bij het uitvoeren van deze instructie zal de JVM geheugen moeten reserveren, alle velden van het object initialiseren en de constructor van de klasse oproepen.

De JVM moet uiteraard weten hoe een object van dit type eruit ziet, welke velden het heeft en ook welke code er bij het object hoort. Al die informatie zit opgeslagen in de Java-klasse. Als de klasse nog niet ingeladen was, dan gebeurt dat ook op het moment dat de `new`-bytecode uitgevoerd wordt. Dan wordt binnen de JVM ook de *bytecode verifier* opgeroepen om de pas ingeladen klasse te verifiëren. Mocht er iets fout lopen, zal de JVM een *exception* opwerpen.

De complexiteit van de `new`-bytecode is dus niet te vatten in een enkele instructie in conventionele hardware-machinecode, maar als bytecode beschrijft ze zeer elegant de rijke functionaliteit die de JVM aanbiedt.

De *bytecode verifier* vormt een essentieel onderdeel van de JVM. Hij zorgt ervoor dat alle bytecodes die door de JVM worden uitgevoerd veilig en geldig zijn. Een deel van zijn taak bestaat bijvoorbeeld uit het bewaken van de integriteit van de stapel: de bytecodes in een Java-methode moeten zo opgebouwd zijn dat het geheel van *pop*- en *push*-instructies perfect klopt en dat op elk ogenblik de datatypes van wat er op de stapel ligt overeenkomen met wat de code verwacht. Dit wordt statisch geanalyseerd bij het inladen van een klasse in de JVM.

Daarnaast doet de JVM ook heel wat controles tijdens de uitvoering. Zo wordt bij elke toegang naar een *array* gecontroleerd of de index binnen de grenzen van die *array* blijft. Gelijkaardige controles zorgen ervoor dat er nooit geschreven kan worden naar een variabele die een *null*-object voorstelt.

De bytecodes zijn zo opgebouwd dat alle type-informatie en informatie over de opbouw van de stapel en lokale variabelen in elke methode, vervat zit in de bytecode. De VM heeft dus toegang tot al deze hoog-niveau-informatie tijdens de uitvoering van het programma. Dit is uniek ten opzichte van C en andere zwak getypeer-

de talen zoals Python. Deze talen zijn minder expliciet waardoor statische compilers bijvoorbeeld minder veronderstellingen kunnen maken over de code, wat het moeilijker maakt programma's in C of Python te analyseren en te optimaliseren.

2.4.2 De taal Java

De syntax van Java is geïnspireerd op die van C en C++ (Lindholm & Yellin, 1999), maar is wel opvallend eenvoudiger: Java gebruikt slechts 48 *keywords*, daar waar C++ er 62 nodig heeft.

Het belangrijkste doel van C en bij uitbreiding C++ was om de programmeur toegang te geven tot de beste prestaties en als dat nodig is ook tot manuele laag-niveau-optimalisaties, zonder dat hij daarbij moest afdalen tot het niveau van behoorlijk omslachtige assemblercode.

Het idee van Java daarentegen was om een efficiënt platform aan te bieden waarmee de programmeur snel en eenvoudig alles kan doen wat hij nodig heeft, zonder dat hij zich zorgen moet maken om de details van het lagere assemblerniveau. Java werd immers specifiek ontwikkeld om uitgevoerd te worden in een VM, op een hoger abstractieniveau.

Een goed voorbeeld daarvan is opnieuw het Java-geheugenmodel. Java is zo opgebouwd dat er enkel een instructie bestaat voor het aanmaken van nieuwe objecten in het geheugen, `new`, maar geen enkele instructie voor het verwijderen van die objecten en het opnieuw vrijgeven van het geheugen. Java steunt immers volledig op automatisch geheugenbeheer (*garbage collection*) in de JVM.

2.4.3 Java Virtuele Machines in de software-wereld

Hoewel Sun de Java-taal en de bijbehorende JVM heeft ontworpen en gespecifieerd (Lindholm & Yellin, 1999), zijn er in de loop der jaren ook andere JVM's op de markt gekomen. Enkele jaren na de uitgave van de eerste JVM, die relatief traag werkte, heeft Sun de Java Hotspot VM op de markt gebracht. Deze nieuwe JVM bevatte voor het eerst een geavanceerde JIT-compiler, die oorspronkelijk ontwikkeld werd bij Longview Technologies (Griesemer, 1999).

Ook andere fabrikanten brachten hun eigen JVM's op de markt. Zo is er de Jikes RVM (Alpern et al., 2000), die oorspronkelijk onder de naam Jalapeño binnen IBM ontwikkeld werd, en tot op vandaag in de onderzoeksgemeenschap geldt als de meest vooruitstrevende,

open-bron JVM. IBM heeft ook nog steeds een commerciële JVM, de IBM J9, die in tegenstelling tot Jikes wel volledig compatibel is met de Java-standaard. IBM J9 is geoptimaliseerd voor gebruik binnen, onder meer, IBM's Websphere-omgeving.

2.5 Virtuele Machines in de ingebedde wereld

Hoewel de focus oorspronkelijk lag op webtoepassingen en *applets*, wordt Java meer en meer gebruikt voor ingebedde toepassingen.

2.5.1 Java Platform Micro Edition

Specifiek voor ingebedde toepassingen heeft Sun een lichtgewichtversie van Java ontwikkeld, de Java Platform Micro Edition (J2ME) (Sun, 2000). Binnen deze standaard bestaan er verschillende J2ME-profielen die elk een subset van de Java-standaard definiëren en die ook vastleggen welke de minimale systeemvereisten zijn waaraan een systeem moet voldoen om een zeker profiel te ondersteunen. Op die manier zal software geschreven voor een specifiek profiel, gegarandeerd werken op systemen die aan dat profiel voldoen.

Het belangrijkste profiel is wellicht de Connected Limited Device Configuration (CLDC) (Sun, 2005a). Hiervoor volstaat een eenvoudige 16-bit processor met een klokfrequentie van 50 MHz en een RAM-geheugen van slechts 300 KiB, wat overeenkomt met het lagere uiteinde van de markt voor onder meer PDA's in 2005.

Maar het kan nog kleiner en nog compacter. De kleinste JVM's vandaag zitten verstopt in zogenaamde *smart cards*, zoals bijvoorbeeld de Belgische elektronische identiteitskaart (Sun, 2005b). Deze kaarten werken met de *Java Card*-technologie (Chen, 2000). De beperkingen ten opzichte van de volledige Java-standaard zijn voornamelijk: *floating point* bewerkingen worden niet ondersteund, het is niet mogelijk om zelf *class loaders* te definiëren en de mogelijkheden voor het afhandelen van *exceptions* zijn beperkt.

Binnen Sun werd ook de Squawk VM ontwikkeld die gebruikt zal worden in de volgende generatie *smart cards* (Shaylor et al., 2003).

2.5.2 PicoJava: een processor die bytecodes uitvoert

Terwijl Sun zich hoofdzakelijk heeft gespecialiseerd in het ontwikkelen van JVM's voor conventionele ingebedde processors, is het ook

mogelijk om specifieke processors te ontwikkelen die rechtstreeks Java-bytecode kunnen uitvoeren.

De eerste processor die bytcodes direct kan uitvoeren is PicoJava (McGhan & O'Connor, 1998). PicoJava, een technologie die bij Sun zelf werd ontwikkeld, kon de uitvoering van Java-programma's versnellen met een factor 20 in vergelijking met een klassieke JVM op een toendertijd recente Intel-processor. Ondanks de meer dan behoorlijke prestaties werd PicoJava door Sun nooit gelanceerd als een echt product. Daarvoor was de processor te groot en te complex. Niettemin blijft het een belangrijk vergelijkingspunt voor nieuwere bytecodeprocessors.

2.5.3 Java Optimised Processor

In de academische wereld maakt de Java Optimised Processor (JOP) al enige tijd furore (Schoeberl, 2008). Dit is een sterk gespecialiseerde processor die rechtstreeks Java bytecode uitvoert. Het belang van deze processor ligt niet zo zeer in het feit dat hij rechtstreeks Java ondersteunt, maar wel in de betrouwbare voorspelbaarheid van de uitvoeringssnelheid op deze processor. Specifiek voor *real-time* systemen, werd gestreefd naar een accurate schatting van de Worst-Case Execution Time (WCET).

In tegenstelling tot de JVM die een Complex Instruction Set Computer (CISC) is, kiest Schoeberl in de JOP resoluut voor een Reduced Instruction Set Computer (RISC)-architectuur. De Java-bytecodes worden dus niet rechtstreeks uitgevoerd, maar ze worden eerst omgezet in een reeks van RISC-microcode-instructies. Dat is dus vergelijkbaar bij wat er gebeurt in een traditionele JVM op een conventionele computer, al is het nu de hardware die voor de vertaling zorgt en geen JVM in software. Omdat elke microcode-instructie exact een klokcyclus in beslag neemt en omdat de vertaling van bytecodes naar microcode eveneens vastligt —bytecodes zijn onafhankelijk van elkaar— kan de WCET van een Java-programma op de JOP accuraat berekend worden. Merk op dat de JOP geen *caches* bevat zodat ook de tijd die nodig is voor geheugenoperaties exact vastligt.

Het moet gezegd dat JOP slechts een deel van de Java-standaard implementeert. Van de 201 verschillende bytecodes worden er 43 uitgevoerd in een enkele klokcyclus, voor 93 bytecodes wordt een sequentie van microcode-instructies gebruikt en 40 bytecodes worden in Java zelf geëmuleerd.

2.5.4 ARM Jazelle DBX

Het is ook mogelijk om klassieke processors uit te rusten met Java-technologie. Recenter lanceerde ARM een serie processors met de Jazelle DBX-technologie (Porthouse, 2005) waardoor ook die processors Java-bytecode rechtstreeks kunnen uitvoeren. DBX staat voor *Direct Bytecode eXecution*.

Deze processors kunnen echter niet alle bytecodes zelf uitvoeren. De bedoeling is dat een speciale JVM in software draait op het klassieke deel van de processor. Deze JVM heeft weet van de Jazelle-ondersteuning in de processor. Op die manier kan de VM tot 95% van alle uitgevoerde bytecodes rechtstreeks laten uitvoeren door de Jazelle-processor, maar de overige functionaliteit moet op de klassieke manier door de VM geregeld worden.

2.5.5 Dalvik en het Android-platform

In 2007 lanceerde Google het Android-platform voor mobiele telefoons. Tegelijkertijd werd ook de Open Handset Alliance (OHA) opgericht, een overkoepelende organisatie van verschillende bedrijven die mee het Android-platform willen ontwikkelen of het platform willen ondersteunen met hun systemen.

De kern van het platform wordt gevormd door het Linux-besturingssyteem waarop de Dalvik VM (Bornstein, 2008) draait.

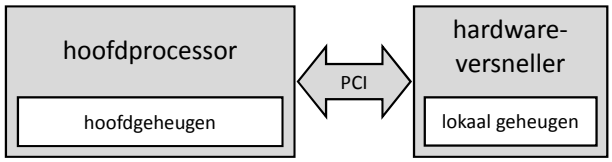
De volledige ontwikkelomgeving steunt op het Java-platform. Alle programma's worden ontwikkeld in Java, de standaard-IDE is Eclipse en Google biedt heel wat Java-bibliotheken aan om specifieke onderdelen van de Android-telefoons aan te sturen.

Nadat de toepassing getest en ontwikkeld is in Java, wordt de Java-bytecode vertaald naar specifieke Dalvik-code die dan door de Dalvik VM kan worden uitgevoerd. Er wordt op het Android-platform dus geen gebruik gemaakt van de JVM.

Bijzonder is dat de Dalvik VM registergebaseerd werkt en dus niet stapelgebaseerd zoals de oorspronkelijke JVM.

2.6 Hardwareversnelde Java virtuele machine

In zijn doctoraat stelt Faes (2008) de hardwareversnelde JVM voor. Dit is een specifiek voor hardware/software co-ontwerp aangepaste versie van de IBM's Jikes VM.



Figuur 2.1: Dit hybride hardwareplatform bevat een generieke hoofdprocessor en een toepassingsspecifieke hardwareversneller op de FPGA.

Faes' aanpak voor hardware/software co-ontwerp is het klassieke concept van een versneller als co-processor: het hardwareplatform is een hybride systeem dat een generieke hoofdprocessor bevat en een toepassingsspecifieke hardwareversneller (figuur 2.1). Die versneller voert een klein maar rekenintensief deel van het Java-programma uit, terwijl de hoofdprocessor de rest van het programma voor zijn rekening neemt. Meestal wordt de rest van het programma immers gedomineerd door controlecode die inherent sequentieel is zodat een parallelle FPGA hiervoor geen of nauwelijks meerwaarde biedt.

Faes (2004) toonde aan dat de JVM het middel bij uitstek is om de complexiteit te beheersen van de communicatie en synchronisatie tussen de hardwareversneller en de generieke processor. Door dit alles te verbergen binnen een abstractielaag gevormd door de JVM hoeft de programmeur zich hierover geen zorgen meer te maken.

De JVM kan zijn rol als abstractielaag tussen de hardwareversneller en het Java-programma uiteraard pas correct spelen als er een duidelijk verband bestaat tussen de hardware enerzijds en concepten in Java en de JVM anderzijds. Faes definieert een rigoureuze equivalentierelatie tussen de functionaliteit van de hardwareversneller in de FPGA en één of meerdere Java-methodes in de software. De programmeur ziet de hardwareversneller, door de abstractielaag van de JVM, dus als een 'gewone' Java-methode die hij kan oproepen zoals alle andere Java-methodes in het programma. Deze volledig transparante aanpak geeft de JVM de mogelijkheid om tijdens de uitvoering van het programma te beslissen om voor een concrete methode al dan niet de co-processor op te roepen, afhankelijk van de aanwezigheid of beschikbaarheid van de bijbehorende hardwareversneller.

De JVM kan zelfs meer taken op zich nemen dan louter het tijdig opstarten van de juiste, reeds bestaande co-processor. Als hercon-

figureerbare hardwareversnellers gebruikt worden, is het zelfs mogelijk om dynamisch functionaliteit te verplaatsen van de generieke processor naar de FPGA. De nieuwe configuratie voor de FPGA kan dan uit een bibliotheek worden gehaald, maar kan eventueel zelfs *on-the-fly* gegenereerd worden (Beck & Carro, 2005; Bruneel et al., 2007; Bruneel & Stroobandt, 2008a; Lysecky et al., 2006). Deze manier van werken kan gezien worden als een uitbreiding van het concept JIT-compilatie van klassieke softwarecompilatie naar hardwaregeneratie.

Op figuur 2.1 is te zien dat zowel de hoofdprocessor als de hardwareversneller elk hun eigen lokale geheugen hebben. In deze hardwareversnelde JVM strekt de Java *heap* zich uit over beide fysieke geheugens. Java-objecten kunnen dus probleemloos in het ene of in het andere geheugen geplaatst worden. Faes et al. (2005) hebben hiervoor de *garbage collector* van de JVM aangepast.

De processor en de co-processor hebben in dit systeem dan wel toegang tot de Java-objecten in beide geheugens, maar uiteraard zijn de toegangstijden voor geheugentoegangen over de relatief trage PCI-bus heel wat groter dan de toegang van elke component naar zijn eigen lokale geheugen. In hoofdstuk 5 breid ik het werk van Faes uit met een zelflerend algoritme dat probeert voor elk Java-object het meest geschikte fysieke geheugen te vinden.

2.7 Besluit

In dit hoofdstuk heb ik een overzicht gegeven van VM's in het algemeen en van de werking van de JVM in het bijzonder.

Hoofdstuk 3

Communicatiestromen in systemen en programma's

Communicatie in systemen is erg belangrijk omdat het een cruciale impact heeft op de systeemprestaties. Daarom is het nuttig om communicatie of datastromen in programma's te modelleren en op te meten. Eens er een model is voor de datastromen, kan dat gecombineerd worden met een model voor de architectuur van een praktische implementatie van een systeem. Die combinatie maakt het mogelijk om prestatieschattingen te maken en op basis daarvan bepaalde parameters in het systeem te wijzigen: bijvoorbeeld de systeempartitionering, de communicatie-infrastructuur in het systeem enzovoort.

In dit hoofdstuk beschrijf ik twee types communicatieprofielen om de datastromen in programma's te modelleren en hoe ik deze profielen in de praktijk kan opmeten.

Het eerste type communicatieprofiel vormt de basis voor het statisch partitioneren van programma's in hoofdstuk 4. In hoofdstuk 5 maak ik gebruik van het tweede type communicatieprofiel om in de hardwareversnelde Java Virtuele Machine (JVM) een communicatiebewuste plaatsing van de objecten in het geheugen te realiseren.

Omdat het opmeten van deze profielen, in het bijzonder het eerste type, veel tijd in beslag neemt, besteed ik ook uitgebreid aandacht aan de mogelijkheid om bemonsterend te profileren met *reservoir sampling*. Deze techniek laat toe om de profilering drastisch te versnellen en tegelijkertijd toch een aanvaardbare, en vooraf statistisch vastgelegde, nauwkeurigheid te behouden.

3.1 Model voor datastromen in programma's

Communicatie in een systeem wordt steeds veroorzaakt door data die tussen verschillende programma's of tussen verschillende onderdelen van hetzelfde programma wordt uitgewisseld. Om de impact van de communicatie in een systeem correct te kunnen inschatten, wordt eerst een model opgesteld van de datastromen in een programma. Er zijn verschillende manieren om datastromen te modelleren. Welk model het meest geschikt is, hangt af van wat men met de gegevens wil doen. Daarom start deze paragraaf met de belangrijkste kenmerken van de modellen die in verwant werk gebruikt worden.

3.1.1 Actors en/of datastructuren

Data stroomt in een programma van producenten naar consumenten. Wanneer in een programma een berekening wordt uitgevoerd, dan produceert die berekening een resultaat. De methode die deze berekening uitvoert, wordt dan *producent* genoemd. Op het moment dat het resultaat van deze berekening later in het programma wordt gebruikt, spreekt men van consumptie van data. De betreffende methode is dan een *consument*. Omdat de meeste producenten tegelijk ook als consument van data optreden en vice versa, worden ze vaak abstract *actor* genoemd. Een actor kan dan zowel data produceren als consumeren.

Wanneer twee actors met elkaar communiceren dan gebeurt dat via een kanaal. Binnen computerprogramma's is dat kanaal in veel gevallen een datastructuur in het geheugen. De ene actor schrijft naar die datastructuur en de ander leest de resultaten van de ene actor om ze verder te verwerken. Data kan ook doorgegeven worden als argument bij een methode-oproep.

Er zijn verschillende manieren om naar de datastromen in programma's te kijken. Een eerste optie is om enkel de datastromen tussen de actors te beschouwen zonder rekening te houden met de specifieke datastructuren die het kanaal vormden voor de datastroom. Deze aanpak volg ik in hoofdstuk 4 voor statische partitionering van programma's. Dit communicatieprofiel geeft een abstract beeld van de communicatie die effectief nodig is tussen de verschillende actors voor het correct uitvoeren van het programma. Het is erg nuttig voor

systeemontwerp waarbij de *mapping* van communicatie op concrete datastructuren later gebeurt.

Een tweede optie is het bekijken van de datastromen tussen actors en datastructuren. Dit geeft een overzicht van welke actors hoe vaak communiceren met welke datastructuren. Deze aanpak wordt onder meer gevolgd in het onderzoek naar optimalisatie van geheugenstructuren (Catthoor et al., 1998). De datastructuren zelf liggen daar immers al vast en er komen vragen aan bod als: welke datastructuren worden in welke geheugens geplaatst? Kan een *cache* worden toegevoegd voor het minimaliseren van de datatoegangen naar een specifieke datastructuur?

In hoofstuk 5 toon ik aan dat dit tweede communicatieprofiel tijdens de uitvoering van een programma incrementeel kan worden opgebouwd. De resultaten hiervan blijken bijzonder nuttig te zijn om de plaatsing van Java-objecten in het geheugen van de hardwareversnelde JVM te optimaliseren zodat er uiteindelijk minder externe communicatie is.

De concrete interactie tussen actor en datastructuur kan ook mee gemodelleerd worden. Het model bevat dan onder meer precieze informatie over welke elementen van een rij worden aangesproken door een specifieke methode of lusnest en over de volgorde waarin dat gebeurt. Onder meer Bastoul (2004), Beyls (2004) en Devos (2008) leverden op dit vlak baanbrekend werk. De modellen die zij voor hun werk hanteerden bevatten een polyedrische voorstelling van de datatransfers in programma's.

3.1.2 Statische of dynamische analyse

Datastromen in programma's kunnen zowel statisch als dynamisch bestudeerd worden. Beide vormen van analyse vullen elkaar aan omdat ze verschillende aspecten van de communicatie in programma's kunnen blootleggen.

Een statische analyse poogt op basis van een analyse van de broncode informatie te verzamelen over wat het gedrag van een programma tijdens de uitvoering zal zijn. Vaak is de uitvoering van een stukje broncode, en zeker de uitvoering van een volledig programma, invoerafhankelijk. Een statische analyse tast in dat geval de grenzen af van mogelijke uitvoeringen. De resultaten van een statische analyse zijn bij defintie degelijk en betrouwbaar: ze gelden voor elke mogelijke uitvoering van het programma, onafhankelijk van de

invoer. Maar net doordat de eigenschappen die de statische analyse naar boven brengt gegarandeerd moeten gelden voor *alle* mogelijke uitvoeringen van het programma, zijn de eigenschappen conservatief en soms weinig specifiek.

Ernst (2003) geeft hiervan een mooi voorbeeld. Statische analyse van een functie f zou kunnen vertellen dat f altijd een niet-negatief getal teruggeeft. Die uitspraak is duidelijk en correct voor alle uitvoeringen, maar ze is veel minder specifiek dan stellen dat f steeds de absolute waarde van zijn argument berekent. Die laatste stelling is echter wel veel nuttiger en bruikbaarder als analyse. Voor kleinere stukjes programmacode die nauwelijks of zelfs helemaal niet invoerafhankelijk zijn, zoals het voorbeeld van de absolutewaardeberekening, is statische analyse bijzonder nuttig. Maar voor de analyse van volledige programma's die bijna bij definitie invoerafhankelijk zijn, levert een dynamische aanpak vaak concretere informatie op.

Bij een dynamische analyse wordt uitgegaan van één concrete uitvoering van het programma of een relatief beperkt aantal uitvoeringen. Op die manier wordt veel meer concrete informatie opgehaald. De prijs hiervoor is dat de analyse niet zonder meer veralgemeend mag worden naar andere uitvoeringen van hetzelfde programma. De eigenschappen van een programma die vastgesteld worden in een dynamische anlyse zijn daardoor optimistisch. Vaak kan een dynamisch analyse dan ook sterkere uitspraken doen over het gedrag van een programma.

In dit proefschrift werk ik uitsluitend met dynamische communicatieprofielen. Deze keuze is onder meer gebaseerd op de hoger aangehaalde voordelen, maar ze is ook ingegeven door de context van dit werk: de hardwareversnelde JVM. In een dynamische omgeving als een virtuele machine, kunnen tijdens de uitvoering van het programma optimalisaties worden doorgevoerd. Deze optimalisaties zullen dan *de facto* gebaseerd zijn op informatie die dynamisch beschikbaar is en opgemeten wordt.

3.2 Communicatie opmeten in programma's

3.2.1 Keuze van de specificatietaal

Als uitgangspunt voor het opmeten van de communicatie wordt in dit proefschrift gebruik gemaakt van een uitvoerbare beschrijving van het systeem. In dat computerprogramma worden met een ge-

paste profileerder alle datastromen opgemeten. De taal waarin het programma beschreven is, moet aan een aantal eisen voldoen om profilering mogelijk en zinvol te maken.

Allereerst moet het een hoog-niveaubeschrijving van het programma zijn. Het is immers de bedoeling om de ontwerper reeds in een vroeg stadium van het ontwerp bij te staan, wanneer er nog geen fundamentele ontwerpsbeslissingen genomen zijn. Talen als structurele VHDL of assembler vallen dus al af.

Bovendien is het erg handig als de beschrijvingstaal objectgeoriënteerd is en sterk getypeerd. Dat geeft de broncode een stevige structuur met een duidelijk, vast objectmodel. Matlab of Python zouden om die redenen dus geen goede keuze zijn.

Java voldoet aan al deze eisen en is bovendien makkelijk te analyseren. Die taal leek dan ook ideaal om dit onderzoek uit te voeren. Later kan de ontwikkelde methodologie eventueel uitgebreid worden naar andere specificatietalen. Uitbreiding naar C# voor de .NET-omgeving van Microsoft (Box et al., 2002) is conceptueel vanzelfsprekend.

3.2.2 Keuze van een profileringsraamwerk

Java Virtual Machine Tool Interface (JVMTI). De JVMTI maakt het mogelijk om een zogenaamde *profiler agent*, geschreven in *native code* voor het gegeven platform, toe te voegen aan lopende Java-programma's. Deze aanpak kan gebruikt worden om een profileerder te schrijven in C of C++ die op de hoogte wordt gehouden van alle relevante gebeurtenissen in de JVM, onder meer lees- en schrijfoperaties naar het geheugen, het inladen van nieuwe klassen in de JVM, het aanmaken van nieuwe objecten en de *garbage collection*. Op die manier krijgt de profileerder inzicht in de werking van het Java-programma op een laag niveau. De profileerder kan ook actief informatie opvragen aan de JVM. Hij kan bijvoorbeeld vragen aan welke Java-objecten een fysiek adres gekoppeld is.

Een belangrijk nadeel van de JVMTI is dat de koppeling tussen de profileerder en het Java-programma of de JVM zeer zwak is. De profileerder kan geen rechtstreeks inzicht krijgen in de interne toestand van de JVM of een overzicht krijgen van alle objecten die zich in de JVM bevinden. Om bij te houden welke objecten er zijn en welke methoden toegang krijgen tot die objecten, zou de profileerder zelf een volledige representatie moeten maken, en voortdurend aanpassen,

van de interne toestand van de JVM. Dit is uiteraard eerder inefficiënt voor een profileerder die deze informatie allemaal nodig heeft om een datastroomgraaf te kunnen opbouwen.

Virtuele machine aanpassen. Een andere mogelijkheid voor het profileren van Java-programma's is om rechtstreeks in te grijpen in de JVM. Dat is immers de plaats waar alles gebeurt: het Java-programma wordt er, bytecode voor bytecode, uitgevoerd en ook alles wat met het lezen of schrijven van het geheugen te maken heeft, zoals het aanmaken van nieuwe objecten, de lees- en schrijfoperaties zelf en de *garbage collection*, wordt gestuurd vanuit de JVM.

De Jikes Research Virtual Machine (RVM) van IBM, oorspronkelijk Jalapeño genoemd, is een open-bron virtuele machine die volledig in Java geschreven is (Alpern et al., 2000). Deze eigenschappen maken de RVM tot een uniek onderzoeksplatform waarbinnen het zeker mogelijk is om een profileerder te implementeren. Toch zijn er ook een aantal belangrijke nadelen of beperkingen.

Allereerst is het niet triviaal om een JVM aan te passen. Als het louter gaat over het maken van een profileerder, dan is het aanpassen van de JVM waarschijnlijk niet de snelste of handigste methode. Maar als voor een specifieke toepassing al aanpassingen zijn gemaakt in de JVM, zoals bijvoorbeeld voor het werk van Faes (2008) waar ik in hoofdstuk 5 op voortbouw, dan kan het wel zinvol zijn om de profileerder ook in te bouwen in de JVM. De belangrijkste inspanningen om in de JVM te duiken, zijn dan immers al gebeurd.

Een tweede reden om er niet meteen voor te kiezen om de JVM aan te passen, is dat een van de belangrijkste voordelen van het gebruik van Java en de JVM, met name de platformonafhankelijkheid, dan verloren gaat. Er moet dan immers een keuze gemaakt worden voor één specifieke JVM op één specifiek platform. In het geval van de Jikes RVM is dat dan Linux.

Aspect-georiënteerd programmeren (AOP). Het profileren van programma's komt er essentieel op neer dat bijkomende functionaliteit, instrumentatiecode, aan het programma moet worden toegevoegd. Een interessante techniek om dat op een betrouwbare en controleerbare manier te doen is AOP (Kiczales et al., 2001, 1997). Een klassiek voorbeeld van AOP is het toevoegen van *logging* aan een programma door bij het begin en aan het einde van elke methode een oproep toe te voegen van de profileringsmethodes `beginMethod()` of

`endMethod()`. Een AOP-raamwerk laat in dit geval toe om zeer precies te definiëren welke extra methodes moeten worden toegevoegd en waar die in het originele programma moeten komen.

AOP is een wijdverbreide techniek voor hoog-niveau *code injection*. Toch zijn standaard AOP-bibliotheken die ik ter beschikking had bij de aanvang van dit onderzoek, niet geschikt voor de gedetailleerde datastroomprofilering die ik nodig had. Voor het opmeten van datastromen moeten immers alle geheugentoegangen geïnstrumenteerd worden en dat vereist een aanpak tot op een lager niveau, met name dat van individuele instructies.

Vandaag zijn er heel wat meer mogelijkheden, waaronder het specifiek voor Java ontworpen Javana (Maebe et al., 2006a). Javana is gebouwd bovenop Diota (Maebe et al., 2002), een dynamisch en transparant instrumentatieraamwerk voor de x86-architectuur dat qua functionaliteit vergelijkbaar is met Valgrind (Nethercote & Seward, 2007). Javana combineert de flexibele en gedetailleerde aanpak van Diota op het lage niveau van individuele assemblerinstructies met de kracht van hoog-niveauconcepten in Java en de JVM. Zowel de JVM als Diota werden aangepast om wederzijdse communicatie mogelijk te maken. Daardoor kan Javana perfect elke lees- of schrijfoperatie in het geheugen monitoren, op assemblerniveau, en dit linken aan concrete velden van Java-objecten op het hogere niveau. Als ik opnieuw zou beginnen aan dit onderzoek, dan zou ik wellicht met Javana van start gaan.

Bytecode-instrumentatie. Recent heeft Sun het concept van bytecode-instrumentatie toegevoegd aan de Java-standaard in versie 1.5. Vooraleer een klasse wordt geladen in de JVM, krijgt een specifieke instrumentatieklasse die door de profileerder wordt aangeleverd, de kans om de inhoud van de nieuwe klasse te wijzigen. Verscheidene wijzigingen zijn mogelijk waaronder het toevoegen van extra velden en extra methodes, het aanpassen van bestaande velden of wijzigen van de functionaliteit in elke methode. De *instrumentation package* laat ook toe om klassen die al in de JVM zijn ingeladen alsnog te wijzigen. In dat geval gelden de wijzigingen echter alleen maar voor nieuwe objecten van die klasse, niet voor eerder aangemaakt objecten.

Door rechtstreeks de bytecode, en dus functionaliteit, van alle methodes in elke klasse te instrumenteren, kan de profileerder het programma zo aanpassen dat hij op de hoogte blijft van elke in-

dividuele operatie die door de JVM wordt uitgevoerd. Bytecode-instrumentatie biedt op die manier de flexibiliteit om een erg veelzijdige profileerder te schrijven.

Het feit dat de instrumentatie van het oorspronkelijke Java-programma in deze oplossing ook volledig vanuit Java zelf gebeurt, maakt het hele instrumentatieproces volledig transparant voor de gebruiker. Bovendien blijft het Java-programma zo ook volkomen platformonafhankelijk.

Het eigenlijke manipuleren van Java-bytecode is echter niet triviaal. Gelukkig bestaan er enkele goede standaardbibliotheken om snel en efficiënt de inhoud van de methodes van een Java-klasse te wijzigen. In het begin van mijn doctoraatsonderzoek heb ik BCEL (Dahm, 2001) gebruikt. Later ben ik overgestapt op ASM (Bruneton et al., 2002) omdat dat iets eenvoudiger werkt en ook tot 11 keer minder vertraging veroorzaakt.

3.3 Opmeten van communicatie tussen actors

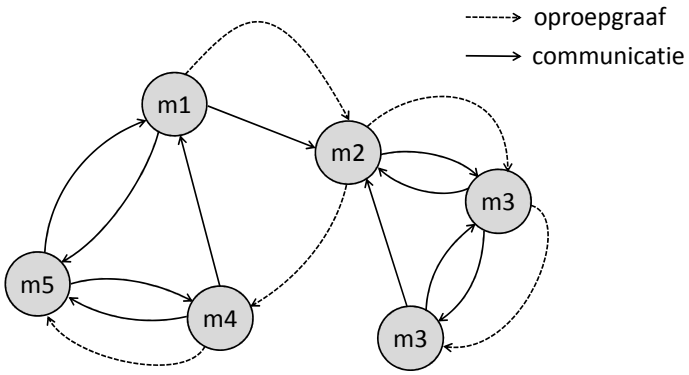
3.3.1 Globaal overzicht

Zoals hoger beschreven werd, is het de bedoeling van de profileerder om de datastromen tussen verschillende actors in het programma te detecteren. Een datastroom ontstaat wanneer een actor (de consument) data leest die door een andere actor (de producent) geschreven werd. Voor het profileren komt het er dus op aan om alle lees- en schrijfoperaties naar het geheugen in het programma te onderscheppen.

Dit gebeurt als volgt: voor elke geheugenlocatie wordt bijgehouden welke actor (A) er het laatst een waarde in geschreven heeft. Op het ogenblik dat die waarde dan gelezen wordt door een consument (B), moet een datastroom van A naar B geregistreerd worden.

Tijdens de uitvoering van het programma wordt dynamisch de datastroomgraaf aangemaakt die de communicatie weergeeft tussen de verschillende actors. Telkens als er een nieuwe actor opduikt, wordt de graaf uitgebreid met een nieuwe knoop. De registratie van een datastroom leidt dan tot het toevoegen van een nieuwe tak in de graaf.

Bovendien wordt elke nieuwe actor ook dynamisch toegevoegd aan de oproepgraaf van het programma. Figuur 3.1 toont hoe dit werkt: de oproepgraaf, in feite een oproepboom, wordt weergege-



Figuur 3.1: Een concreet voorbeeld van communicatieprofiel met communicatie tussen actors. Dit profiel bestaat uit de oproepgraaf van het programma (stippellijn) en de datastroomgraaf (volle lijnen) die weergeeft hoe de actors met elkaar communiceren.

ven met een stippellijn, de volle lijnen geven de takken van de datastroomgraaf. Merk op dat de actors allemaal methodes van het programma zijn en dat dezelfde methode, als ze meerdere keren wordt opgeroepen vanuit verschillende andere methodes, ook meerdere keren in het communicatieprofiel voorkomt. Dit is het geval voor actor *m3*.

3.3.2 Concrete werking van de profileerder

Bytecodes instrumenteren

De werking van de profileerder steunt op de *instrumentation package* die in Java 1.5 werd toegevoegd: `java.lang.instrument`. Deze package laat toe om een *Java agent* te schrijven die binnen de JVM wordt uitgevoerd en het proces van het laden van de klassen onderschept. Op die manier kunnen alle klassen die de JVM binnenkomen omgeleid worden via een `ClassFileTransformer` die op zijn beurt de bytecode van de binnenkomende klassen kan wijzigen.

Het feit dat deze *Java agent* volledig in Java geschreven wordt, betekent dat er al een aantal klassen, hoofdzakelijk standaardbibliotheken, in de JVM aanwezig moeten zijn voor de *Java agent* geladen kan worden. Die klassen kunnen dus niet meer omgeleid worden bij het inladen. Maar ook daarvoor is een oplossing voorzien: het is

namelijk mogelijk om alle reeds ingeladen klassen op te vragen en ze alsnog te instrumenteren. Op die manier kan er voor gezorgd worden dat alle[1] klassen in de Java-omgeving geïnstrumenteerd worden.

Bijhouden van de 'boekhouding'

Voor elke geheugenlocatie moet de profileerder kunnen bijhouden welke actor de producent is van de inhoud van deze geheugenlocatie. Wanneer een andere actor later dezelfde waarde consumeert, moet de profileerder de producent immers kunnen opzoeken om de datastroom van producent naar consument correct te registreren. Deze boekhouding vergt een aantal datastructuren die in deze paragraaf beschreven staan.

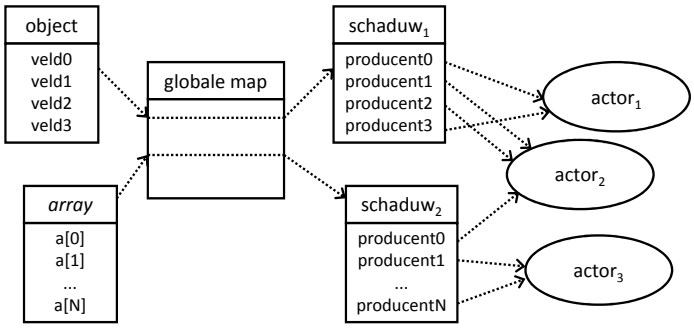
De geheugenlocaties waar de profileerder een overzicht van moet bijhouden nemen verschillende vormen aan. Het kan gaan over klassenvelden of instantievelden of over rijen. Voor elk van deze drie opties heeft de profileerder een specifieke aanpak.

Voor statische velden, ook wel klassenvelden genoemd, houdt de profileerder één globale tabel bij met per veld een referentie naar de actor die producent is van de waarde in dat veld. Deze eenvoudige aanpak is werkbaar doordat het aantal statische velden in een Java-programma beperkt is. Het aantal statische velden is immers van dezelfde grootte-orde als het aantal klassen in een programma.

Voor niet-statische velden of instantievelden ligt dat anders. Elk object O dat een concrete instantiëring is van de klasse K, heeft een eigen waarde voor het niet-statische veld V. Dit aantal loopt snel op waardoor het niet meer haalbaar is om de producenten van al deze velden in één globale tabel op te nemen. Tabel 3.7 toont dat een typisch Java-programma enkele miljoenen objecten kan aanmaken. Daarom wordt per object een *schaduwobject* aangemaakt dat een lijst bevat van referenties van actors. Het schaduwobject dat hoort bij object O van type K zal voor elk instantieveld in de klasse K een referentie naar een actor bevatten.

Voor alle elementen van rijen (*arrays*) moet de profileerder ook bijhouden welke actor dit element produceerde. Het aantal rijen en de grootte ervan is, net als het aantal objecten, vaak vrij hoog. Daar-

[1] Er is een beperkt aantal uitzonderingen van klassen waarvan de implementatie nauw gekoppeld is met de interne werking van de JVM. Voor de Sun Hotspot JVM is er gelukkig maar één uitzondering: `java.lang.Thread`.



Figuur 3.2: De profileerder maakt voor elk Java-object een schaduwobject aan, waarin voor elk veld van het originele object wordt bijgehouden welke actor de producent is van de data in dat veld.

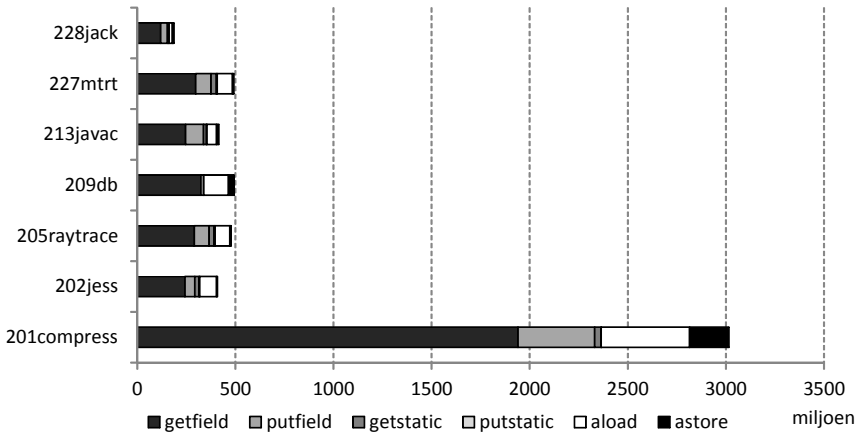
om wordt ook voor rijen een afzonderlijk schaduwobject opgemaakt dat dan per index in de rij een referentie naar een actor bijhoudt.

De profileerder moet uiteraard de koppeling kunnen maken tussen de schaduwobjecten en de bijbehorende objecten of rijen. Dit gebeurt aan de hand van een globaal bijgehouden hakseltabel (*hash map*) die objecten of rijen afbeeldt op hun schaduwobject (figuur 3.2).

Bytecodes onderscheppen

Om de communicatiegraaf op te bouwen, onderschept de profileerder de relevante gebeurtenissen die optreden tijdens de uitvoering van het programma: het oproepen van en terugkeren uit methodes en alle lees- en schrijfoperaties. Tijdens het instrumenteren, worden op de juiste plaatsen in de bytecode van elke methode, extra oproepen ingevoegd naar profileringsroutines in de klasse `Profiler`. Deze paragraaf somt alle gebeurtenissen op beschrijft de bijbehorende profileringsroutines, waarvan de pseudocode in figuur 3.4 staat.

De profileringsroutines `enterMethod()` en `exitMethod()` zorgen voor het opbouwen van de dynamische oproepgraaf. In de instrumentatiefase worden alle Java-methodes zo aangepast dat zij als eerste instructie `enterMethod()` aanroepen en dat vlak voor elke *return*-bytecode de routine `exitMethod()` wordt opgeroepen. Als argument krijgt `enterMethod()` de volledige naam van de methode mee, inclusief de naam van de klasse, de *package* en de methodesignatuur. De klasse `Profiler` bouwt daarmee intern de volledige oproepgraaf op.



Figuur 3.3: Voorkomen van diverse Java-bytecodes tijdens de uitvoering van de programma's van de SPECjvm98 benchmark suite.

Daardoor weet de profileerder altijd perfect welke methode op dit ogenblik door de JVM wordt uitgevoerd en met welke actor in de oproepgraaf en communicatiegraaf die methode overeenkomt.

Naast methode-oproepen wordt ook alle bytecodes die staan voor een lees- of schrijfoperatie onderschept. Vlak voor deze bytecodes wordt eveneens een sprong gemaakt naar profileringscode in `Profiler`. Hieronder volgt een kort overzicht van de bytecodes die mijn profileerder onderschept. Figuur 3.3 toont hoe vaak elk van die bytecodes voorkomt in de programma's van de SPECjvm98 benchmark suite (Standard Performance Evaluation Corporation, 1998).

`GETFIELD` en `PUTFIELD`: deze bytecodes staan voor het lezen en schrijven van een instantieveld V in een object O. Bij `PUTFIELD` wordt de referentie naar de actor waartoe de methode behoort, weggeschreven in het schaduwobject van O. Bij `GETFIELD` wordt een datastroom geregistreerd van de actor die het laatst naar het veld V in object O schreef, naar de actor die de `GETFIELD`-instructie uitvoert.

De bytecodes `GETSTATIC` en `PUTSTATIC` lezen en schrijven een statisch veld V uit klasse K. De profilering gaat hierbij analoog als bij `GETFIELD` en `PUTFIELD`. Er is echter een belangrijk verschil: bij de niet-statische instructies moet telkens het bijhorende schaduwobject opgezocht worden, terwijl dat bij `GETSTATIC` en `PUTSTATIC` niet het geval is. De klasse K waaruit het veld V gelezen of geschreven wordt,

```
globaal Graaf oproepgraaf, datastroom;
globaal HakselMap schaduwObjecten, producentenStatischeVelden;
globaal Actor huidigeActor;

proc enterMethod(String methodNaam):
    voor alle Actor kind in huidigeActor.kinderen:
        if kind.methodNaam == methodNaam:
            huidigeActor := kind
            return
    Actor actor := new Actor(methodNaam)
    huidigeActor.kindToevoegen(actor)
    huidigeActor := actor
end

proc exitMethod():
    huidigeActor := huidigeActor.ouder
end

function getSchaduw(Object object):
    if schaduwObjecten.bevat(object):
        return schaduwObjecten.get(object)
    else:
        Schaduw schaduw := new Schaduw()
        schaduwObjecten.put(object, schaduw)
        return schaduw
end

proc registreerPutField(Object object, String veldNaam):
    Schaduw schaduw := getSchaduw(object)
    schaduw.setProducent(veldNaam, huidigeActor)
end

proc registreerGetField(Object object, String veldNaam):
    Schaduw schaduw := getSchaduw(object)
    Actor producent := schaduw.getProducent(veldNaam)
    datastroom.takToevoegen(producent, huidigeActor)
end

proc registreerPutStatic(String klasseEnVeldNaam):
    producentenStatischeVelden.put(klasseEnVeldNaam, huidigeActor)
end

proc registreerGetStatic(String klasseEnVeldNaam):
    Actor producent := producentenStatischeVelden.get(klasseEnVeldNaam)
    datastroom.takToevoegen(producent, huidigeActor)
end
```

Figuur 3.4: Routines die door de profileerder worden opgeroepen

is immers al bekend op het moment van de instrumentatie. Daardoor verloopt de profilering voor statische velden heel wat efficiënter dan voor niet-statische velden.

Tot slot zijn er nog de bytecodes `xALOAD` en `xASTORE` die een element van een rij lezen of schrijven. Dit verloopt volkomen analoog als het lezen en schrijven van instantievelden uit objecten: eerst wordt het bijhorende schaduwobject opgezocht in de globale hakseltabel en vervolgens wordt de datastroom geregistreerd (bij lezen) of wordt de referentie van de actor gekopieerd in het schaduwobject (bij schrijven).

Zelfinstrumentatie

Wanneer alle methodes in alle klassen geïnstrumenteerd worden, dan kan dat tot problemen leiden. De klasse `Profiler` maakt voor zijn werking gebruik van een aantal methodes uit de standaardbibliotheken die ook gebruikt worden door het programma dat geprofileerd wordt.

Het is echter niet mogelijk om de door `Profiler` gebruikte standaardmethodes volledig uit te sluiten van instrumtentatie. Dat zou immers verkeerde resultaten opleveren als het geprofileerde programma diezelfde methodes zou gebruiken. Het programma zou in dat geval dus slechts gedeeltelijk geprofileerd worden.

In het andere geval waarbij ik alle methodes zou instrumenteren, geeft dat ook verkeerde resultaten wanneer de klasse `Profiler` ze gebruikt. Dan gaat de profileerder zichzelf profileren en dat leidt tot een oneindige recursie en dus onvermijdelijk ook tot het vastlopen van de JVM.

Dit probleem heb ik opgelost door het invoeren van een globale profileringsvlag die de profilering aan- en uitschakelt. Telkens wanneer naar een methode in de klasse `Profiler` wordt gesprongen, wordt de profilering uitgeschakeld. Na het terugkeren uit `Profiler` wordt de profilering hervat. Op die manier zorgt de profilering niet voor perturbatie en worden programma's volledig geprofileerd.

3.4 Bemonsterend opmeten van datastromen

Elk producent-consument-paar in het programma moet worden onderzocht door onze profileerder. Vanwege het grote aantal lees- en schrijfoperaties veroorzaakt dit een enorme *overhead*, zowel in de

uitvoeringstijd als in het geheugengebruik. Door de hoeveelheid producent-consument-paren die geanalyseerd worden te verminderen, kan zowel het geheugengebruik als de vertraging aanzienlijk verminderd worden.

Elke leesoperatie in het programma leidt tot een producent-consument-paar. Het doel van de bemonsteringsmethode is dus het selecteren van een beperkt aantal leesoperaties die in aanmerking worden genomen voor het opbouwen van de datastroomgraaf en dat met een aanvaardbare nauwkeurigheid. In deze paragraaf gebruik ik daarvoor *reservoir sampling* (Vitter, 1985) om dit te bereiken. Paragraaf 3.4.1 beschrijft het algemene principe van *reservoir sampling* en geeft een aantal belangrijke eigenschappen van deze techniek.

Het effect van bemonstering is echter niet uitsluitend positief. Bemonstering zorgt ervoor dat de datastroomgraaf van het programma sneller en efficiënter kan worden opgemeten, maar bemonstering introduceert daarbij ook fouten in de datastroomgraaf. Door het feit dat *reservoir sampling* een uniforme en volstrekt willekeurige steekproef neemt van de producent-consument-paren in het programma, kan gegarandeerd worden dat de fout binnen een op voorhand bepaald betrouwbaarheidsinterval blijft. Bewijs hiervoor wordt geleverd in paragraaf 3.4.2.

3.4.1 Het principe van *reservoir sampling*

Het doel van *reservoir sampling* is het aantal meetgegevens te verminderen door willekeurig een beperkt aantal elementen uit de volledige dataset te selecteren. *Reservoir sampling* slaagt erin dit te doen zonder de volledig omvang van de originele dataset op voorhand te kennen. Deze techniek werd ontwikkeld voor de bemonstering van records in een groot databestand dat op tape is opgeslagen (Vitter, 1985). Voor een bemonstering met *reservoir sampling* moet de tape slechts één keer volledig en lineair doorlopen worden. Bij het profileren van communicatiestromen in Java-programma's doet zich dezelfde situatie voor: tijdens de uitvoering van het programma komt elke lees- of schrijfoperatie één keer voorbij mijn profileerder.

Reservoir sampling werkt als volgt: in de eerste stap worden de eerste n elementen van de oorspronkelijke dataset in het reservoir geplaatst. Vanaf element $n + 1$ worden alle nieuwe elementen één voor één geëvalueerd. Er wordt dan voor elk element m willekeurig beslist om dit element al dan niet toe te voegen aan het reservoir. Bij

een positieve beslissing wordt een reeds aanwezig element a uit het reservoir verwijderd en vervangen door element m. Hierbij wordt $0 \leqslant a < n$ opnieuw willekeurig gekozen. Dit algoritme leidt tot een reeks van exact n elementen, onafhankelijk van de grootte van de dataset.

Vitter (1985) toonde aan hoe de kansverdeling van de beslissingen gekozen moet worden om ervoor te zorgen dat de n elementen in het reservoir op elk ogenblik een uniforme, willekeurige steekproef vormen van de volledige set. Dit wil zeggen dat de waarschijnlijkheid dat een specifiek element m op het einde in het reservoir zit, gelijk is voor alle elementen in de originele dataset.

In het begin worden de meeste elementen van de dataset geselecteerd en opgenomen in het reservoir. Maar naar mate het algoritme vordert worden steeds meer en meer elementen overgeslagen. De waarschijnlijkheid dat een element m in het reservoir wordt opgenomen daalt dus tijdens de loop van het algoritme.

Li (1994) stelt een optimalisatie voor van de basisimplementatie van Vitter. In Li's L-algoritme wordt niet voor elk element afzonderlijk beslist of het al dan niet in het reservoir moet worden opgenomen. In de plaats daarvan beslist het algoritme volgens de juiste kansverdeling hoeveel elementen uit de originele dataset moeten worden overgeslagen (S) vooraleer een volgend element in het reservoir wordt opgenomen.

Initieel worden nog steeds de eerste n elementen van de oorspronkelijke dataset in het reservoir geplaatst. Daarna volgt de initialisatie van de hulpvariabele W:

$$W \leftarrow 1. \tag{3.1}$$

Alle volgende stappen vergen twee berekeningen. Telkens wordt eerst W incrementeel aangepast om vervolgens op basis van de nieuwe waarde van W, te berekenen hoeveel elementen uit de dataset overgeslagen moeten worden: S. In onderstaande formules voor de berekening van W en S staat $\text{random}()$ voor een functie die een willekeurig reëel getal geeft in het bereik $]0, 1[$.

$$W \leftarrow W \exp\left(\frac{\log(\text{random}())}{n}\right) \tag{3.2}$$

$$S \leftarrow \left\lfloor \frac{\log(\text{random}())}{\log(1-W)} \right\rfloor \tag{3.3}$$

Na het overslaan van S elementen, wordt het volgende element m uit de oorspronkelijke dataset aan het reservoir toegevoegd. Net zoals in het algoritme van Vitter, wordt een willekeurig element (uniforme distributie) uit het reservoir geselecteerd, verwijderd en vervangen door element m. Daarna herhaalt men de twee berekeningen uit bovenstaande vergelijkingen om een nieuwe waarde voor S te berekenen.

Het L-algoritme werkt dankzij deze optimalisatie veel sneller dan het oorspronkelijke algoritme van Vitter zonder aan de statistische eigenschappen van *reservoir sampling* te raken. Dit wordt uitvoerig bewezen in (Li, 1994).

3.4.2 Statistisch bewezen nauwkeurigheid

Het belangrijkste voordeel van *reservoir sampling* ten opzicht van andere technieken is het feit dat het een uniforme, willekeurige steekproef garandeert. Daardoor kan ik in deze paragraaf aantonen dat de communicatiegraaf die via *reservoir sampling* wordt opgemeten een voldoende nauwkeurige schatting is van de werkelijke communicatiegraaf. Omdat de grootste datastromen in het programma, de meeste impact hebben op de communicatiekost, is het hier voornamelijk van belang dat die grootste datastromen accuraat worden ingeschat. De nauwkeurigheid wordt hieronder dan ook bepaald op basis van de relatieve grootte van de kleinste datastroompjes die men accuraat wil kunnen opmeten.

Het doel van onze profilering is het inschatten van de hoeveelheid data die stroomt tussen elke twee actors in de communicatiegraaf. De niet-bemonsterde versie van de communicatiegraaf geeft in absolute cijfers weer hoeveel communicatie er is tussen twee actors. De bemonsterde versie geeft enkel een schatting van de relatieve fractie van de totale communicatie in het programma die verloopt via elke tak.

Noem F_e de fractie van de communicatie die over tak e loopt. Deze fractie kan berekend worden als f_e/N, met f_e het totaal aantal producent-consument-paren op deze tak e en N het aantal producent-consument-paren in de volledige communicatiegraaf G. De schatting $\hat{F}_e$ van F_e zal dan berekend worden als c_e/n, waarbij c_e het aantal producent-consument-paren is op de tak e in de steekproef met reservoirgrootte n.

Walpole & Myers (1993) bewijzen dat het betrouwbaarheidsinterval voor $\hat{F}_e$, in de veronderstelling dat de steekproef voldoende groot is en normaal verdeeld, gelijk is aan

$$\hat{F}_e \pm z_{\alpha/2}\frac{s_e}{\sqrt{n}}, \tag{3.4}$$

met s_e de steekproef-standaardafwijking en $z_{\alpha/2}$ een parameter die de gewenste betrouwbaarheid bepaalt. De steekproefvariantie s_e^2 wordt dan gegeven door

$$s_e^2 = \frac{n\sum_{i=1}^{n} x_{e,i}^2 - \left(\sum_{i=1}^{n} x_{e,i}\right)^2}{n(n-1)}. \tag{3.5}$$

In dit geval is $x_{e,i}$ ofwel 1 (producent-consument-relatie ligt op tak e) ofwel 0 (producent-consument-relatie ligt op een andere tak). Daarom kan de steekproefvariantie s_e^2 herschreven worden als

$$s_e^2 = \frac{nc_e - c_e^2}{n(n-1)}, \tag{3.6}$$

waardoor

$$\frac{s_e}{\sqrt{n}} = \sqrt{\frac{(c_e/n)(1-c_e/n)}{n-1}} = \sqrt{\frac{\hat{F}_e(1-\hat{F}_e)}{n-1}}. \tag{3.7}$$

Uit vergelijking 3.4 blijkt dat de verwachte relatieve fout r voor een betrouwbaarheidsinterval α gegeven is door

$$r = z_{\alpha/2}\frac{s_e}{\sqrt{n}}/\hat{F}_e, \tag{3.8}$$

of na combinatie met de vergelijking 3.7:

$$r = z_{\alpha/2}\sqrt{\frac{\hat{F}_e(1-\hat{F}_e)}{n-1}}/\hat{F}_e. \tag{3.9}$$

Deze uitdrukking voor de relatieve fout kan herschreven worden om de gepaste reservoirgrootte n te berekenen die nodig is om een gegeven relatieve fout r te halen voor een verwachte nauwkeurigheid voor F_e. Dit resulteert in:

$$n \geqslant 1 + z_{\alpha/2}^2\frac{1-F_e}{r^2F_e}. \tag{3.10}$$

Het volgende concrete voorbeeld illustreert deze aanpak. Om een schatting te bepalen van de relatieve communicatie op elk tak, die, met 95% betrouwbaarheid ($z_{\alpha/2} = 1,96$), nauwkeurig is tot op 5%, en dat voor alle takken die staan voor minstens 0,1% van de totale communicatie in het programma ($F_e \geqslant 0{,}1\%$), is een reservoir met 1.535.104 elementen nodig.

Hierbij moet worden opgemerkt dat deze statistische discussie alleen betrekking heeft op de belangrijkste takken in de communicatiegraaf. Communicatie over takken die een kleiner aandeel hebben in de communicatiegraaf ($F_e < 0,1\%$) kan alleen maar nauwkeurig worden geschat als alle producent-consument-paren in het programma in rekening worden gebracht. Een bemonsterende aanpak, met *reservoir sampling* of eender welke andere techniek, kan op geen enkele manier garanderen dat dergelijke kleine datastroompjes correct geschat worden.

Voor een volledige analyse van alle data-afhankelijkheden in een programma, is het uiteraard wél belangrijk om ook die kleinere datastromen correct te meten. Maar dat is in dit werk niet de bedoeling. Meer nog: als men effectief alle mogelijke data-afhankelijkheden tussen methodes in een programma wil kennen, dat kan dat alleen door middel van een statische analyse en nooit via een dynamische analyse uitgevoerd op één of slechts enkele specifieke uitvoeringen van een programma.

Mijn bedoeling is, zoals aangehaald, helemaal anders. Ik wil met dit communicatieprofiel een inschatting kunnen maken van de prestaties van het programma wanneer het gepartitioneerd wordt (hoofstuk 4). Kleinere datastromen hebben slechts een beperkte impact op de communicatiekost in een programma, zodat het feit dat ze minder nauwkeurig worden ingeschat bij deze profilering, geen beperking vormt voor de bruikbaarheid van het bemonsterend opgemeten communicatieprofiel.

3.5 Datastroommeting in de praktijk

Om de aanpak geschetst in dit hoofdstuk aan de praktijk te toetsen, werden de communicatiegrafen van zes programma's uit de SPECjvm98 benchmark suite (Standard Performance Evaluation Corporation, 1998) bemeten. Alle metingen werden uitgevoerd op een AMD Opteron 242 met klokfrequentie 1,6 GHz en 4 GiB RAM-geheugen.

Tabel 3.1: Het aantal actors in de communicatiegraaf hangt af van de afbeelding van methodes in het programma op actors in de graaf.

programma	methodes		
	dynamisch	statisch	in context
201compress	225.989.456	804	841
202jess	134.689.951	1242	1287
205raytrace	307.946.068	952	991
209db	75.641.171	818	852
213javac	120.091.066	1591	1637
228jack	65.609.073	1048	1093

3.5.1 Complexiteit van de communicatiegrafen

In mijn eerste experiment heb ik de communicatiegraaf opgemeten voor de SPECjvm98-programma's. De communicatiegraaf die daarbij werd opgesteld was telkens gebaseerd op de fijnst mogelijke granulariteit, met name elke methode-oproep werd als een afzonderlijke actor beschouwd. Deze fijnste granulariteit is een geschikte test om na te gaan in hoeverre de profileerder in staat is de grote hoeveelheid data beschikbaar in een datastroommeting correct te behandelen. Dit is eigenlijk het *worst-case* scenario.

Tabel 3.1 toont hoeveel actors dit opleverde voor elk van de zes programma's. Ter vergelijking geeft de tabel ook twee andere opties voor het afbeelden van methode-oproepen op actors. Een eerste alternatief is de statische optie die elke methode in de code als actor beschouwt. Op die manier vormt elke methode in de code dus één knoop in de communicatiegraaf, onafhankelijk van hoe vaak die methode wordt opgeroepen. Alternatief is het met mijn profileerder ook mogelijk om methodes in hun context te beschouwen. Voor die profilering wordt ook de volledige dynamische oproepgraaf van het programma opgemeten. Dat laat toe om een onderscheid te maken tussen methode A die wordt opgeroepen door methode B of een methode A opgeroepen door methode C. Mehode A komt dan als twee verschillende actors voor in de communicatiegraaf. Maar meerdere oproepen van methode A door methode B worden wel beschouwd als dezelfde actor.

Tabel 3.2: Uitvoeringstijd bij het profileren

programma	origineel	profilering	vertraging	voorspelling
201compress	7,22 s	178.276 s	24692 ×	177.748 s
202jess	3,79 s	24.377 s	6432 ×	25.108 s
205raytrace	3,62 s	31.461 s	8691 ×	29.647 s
209db	13,51 s	29.776 s	2204 ×	30.624 s
213javac	22,94 s	21.081 s	919 ×	24.274 s
228jack	14,15 s	12.126 s	857 ×	11.072 s

3.5.2 Uitvoeringstijd tijdens profilering

Tabel 3.2 toont de uitvoeringstijden van de programma's met en zonder profilering. De kolom *origineel* geeft de oorspronkelijke uitvoeringstijd van het programma, daarnaast staat de tijd die nodig was om het volledige communicatieprofiel op te bouwen: eerst in seconden met daarnaast de relatieve vertraging.

Er zit een behoorlijk grote spreiding op de vertraging van uitvoeringstijd voor de volledige niet-bemonsterende profilering. Het profileren van *201compress* duurt iets meer dan twee dagen, terwijl die klus voor *228jack* in minder dan 4 uur geklaard was. Deze grote verschillen hangen samen met de hoeveelheid werk die de profileerder moet verzetten. Het belangrijkste werk ligt in het registreren van alle lees- en schrijfoperaties voor het opbouwen van de datastroomgraaf. Daarnaast moet ook elke methode-oproep opgevangen worden om de dynamische oproepgraaf op te bouwen. Tabel 3.3 toont exact over hoeveel operaties het gaat. Meer dan 3 miljard lees- en schrijfoperaties verklaren de vertraging voor benchmark *201compress*.

Over al deze programma's heen, heb ik berekend dat de vertraging voor elke gebeurtenis die door de profileerder wordt geregistreerd gemiddeld 59,5μs bedraagt. Op basis van het opgemeten aantal gebeurtenissen (tabel 3.3) en van de opgemeten uitvoeringstijden (tabel 3.2), heb ik met de oplosser in Excel geschat hoeveel tijd er nodig is voor elk van de verschillende profileringsroutines. Het uitgangspunt voor de schatting is onderstaande formule voor de uitvoeringstijd bij profilering ($t_{\text{profilering}}$), waarin $t_{\text{origineel}}$ de originele uitvoeringstijd is en A_{lees}, A_{schrijf} en A_{methodes} respectievelijk het

Tabel 3.3: Aantal gebeurtenissen geregistreerd door de profileerder

programma	leesoperaties	schrijfoperaties	methodes
201compress	2.423.643.636	591.650.957	225.989.456
202jess	350.239.067	57.781.973	134.689.951
205raytrace	391.657.375	86.612.692	307.946.068
209db	449.942.463	43.051.177	75.641.171
213javac	309.335.038	105.490.511	120.091.066
228jack	143.157.090	43.257.030	65.609.073

aantal leesoperaties, schrijfoperaties en methode-oproepen.

$$t_{\text{profilering}} = t_{\text{origineel}} + v_l A_{\text{lees}} + v_s A_{\text{schrijf}} + v_m A_{\text{methodes}} \tag{3.11}$$

Startend van de eerder genoemde globale vertraging van 59,5μs per operatie, berekende Excel de kost voor een methode-oproep (v_m) en voor de registratie van lees- en schrijfoperaties (respectievelijk v_l en v_s). Zoals verwacht, gaat het registreren van een methode-oproep zeer snel: 4,7μs. Het registreren van een schrijfoperatie kost 38,1μs en voor een leesoperatie is 63,6μs nodig. Het profileren van geheugenoperaties is zeer duur omdat bij het lezen of schrijven van een object telkens het bijbehorende schaduwobject opgezocht moet worden. Eens het schaduwobject gevonden is, kan voor een schrijfoperatie vrij snel de producent van de data opgeslagen worden. Leesoperaties vragen nog wat meer werk: eens de producent van de gelezen data bekend is (uit het schaduwobject), moet het gevonden producent-consument-paar aan de datastroomgraaf toegevoegd worden.

Op basis van vergelijking 3.11 en de berekende factoren v_s, v_l en v_m, heb ik de verwachte uitvoeringstijd na profileren voorspeld. Het resultaat van die voorspelling staat ook in tabel 3.2 en blijkt vrij nauwkeurig te zijn. Merk op dat enkel de *overhead* voor de profilering zelf in rekening werd gebracht. De vertraging die optreedt bij instrumenteren van de klassen op het ogenblik dat ze in de JVM worden ingeladen, is niet meegenomen in de voorspelling.

In een tweede experiment heb ik *reservoir sampling* gebruikt voor de profilering van dezelfde communicatiegrafen. Hieruit bleek dat *reservoir sampling* de uitvoeringstijd van de profilering effectief drastisch deed dalen: een gemiddelde daling met een factor 9,45 (geo-

Tabel 3.4: Profileringsvertraging met en zonder *reservoir sampling*

programma	origineel	volledig	$n = 10^3$	$n = 10^6$
201compress	7,22 s	24692 ×	902 ×	1391 ×
202jess	3,79 s	6432 ×	876 ×	971 ×
205raytrace	3,62 s	8691 ×	1662 ×	1743 ×
209db	13,51 s	2204 ×	165 ×	187 ×
213javac	22,94 s	919 ×	141 ×	150 ×
228jack	14,15 s	857 ×	110 ×	178 ×

metrisch) voor een reservoir met 10^3 elementen en 7,67 voor een reservoir met 10^6 elementen. Tabel 3.4 toont de relatieve vertraging wanneer *reservoir sampling* gebruikt werd in vergelijking met profilering zonder bemonstering. Kolom *origineel* geeft de oorspronkelijke uitvoeringstijd voor het uitvoeren van de programma's. De relatieve uitvoeringstijd voor een volledige niet-bemonsterde profilering wordt weergegeven in de derde kolom (*volledig*). De volgende twee kolommen geven de relatieve uitvoeringstijd voor de bemonsterende aanpak met *reservoir sampling* met twee verschillende reservoirs: de eerste keer met 1000 elementen ($n = 10^3$) en de tweede keer 1 miljoen elementen ($n = 10^6$).

De bemonsterde aanpak vermindert de *overhead* in uitvoeringstijd drastisch voor alle programma's. Uit de definitie van het L-algoritme voor *reservoir sampling* blijkt ook dat de relatieve *overhead* door de *reservoir sampling* afneemt naar mate programma's langer lopen. Li (1994) stelde voor de uitvoeringstijd van zijn L-algoritme volgende formule op:

$$D + K_L n \ln\left(\frac{N}{n}\right) + O(n), \tag{3.12}$$

met D de tijd nodig om alle elementen in de set met grootte N te scannen, K_L een factor die de uitvoeringstijd van het berekenen van S en W bevat en n de reservoirgrootte. De term $O(n)$ staat voor de tijd die nodig is om het reservoir te vullen met de eerste n elementen.

Via *reservoir sampling* wordt de reeks van miljoenen leesoperaties teruggebracht tot een reservoir van n elementen. Tijdens de profilering worden echter meer dan n elementen opgenomen in het reservoir: $n \ln(N/n)$. Tabel 3.5 toont om hoeveel elementen het gaat. De eerste kolom in de tabel geeft het aantal leesoperaties (van de

Tabel 3.5: Aantal monsters bij *reservoir sampling*

programma	leesoperaties	$n = 10^3$	$n = 10^6$
201compress	2.423.643.636	14.794	7.796.391
202jess	350.239.067	12.939	5.859.918
205raytrace	391.657.375	12.788	5.972.184
209db	449.942.463	13.138	6.113.087
213javac	309.335.038	12.742	5.736.895
228jack	143.157.090	11.843	4.964.508

verschillende benchmarks). De volgende kolommen geven voor een reservoirgrootte van 1000 en van 1 miljoen *samples* het aantal monsters dat tijdens mijn experimenten effectief opgemeten werd. Men kan berekenen dat die aantallen overeenkomen met de door Li (1994) vooropgestelde $n \ln(N/n)$.

De tabel toont dat de 2,4 miljard leesoperaties in *201compress* uiteindelijk teruggebracht wordt tot 14.794 monsters bij een reservoir van 1000 en tot 7.796.391 bij een reservoir van 1 miljoen monsters. De *overhead* neemt dus, zoals vooropgesteld, fors af bij toenemende reservoirgrootte: voor het grote reservoir worden slechts 7 keer meer monsters opgemeten dan er overblijven in het eindresultaat, waar dat bij een reservoir van 1000 nog 14 keer meer was.

Daar waar bij een volledige profilering de uitvoeringstijd hoofdzakelijk ging naar het verwerken van alle leesoperaties, is dat in het geval van de bemonsterende aanpak niet meer het geval. Zelfs met een reservoir van 1 miljoen elementen wordt nauwelijks 1% van alle leesoperaties in het programma opgemeten. Dit verklaart waarom de vertraging voor profilering met een reservoir van 1 miljoen elementen niet veel groter is dan met een reservoir van 1000 elementen. Voor *201compress* worden in het ene geval 527 keer meer elementen opgemeten, en toch duurt die profilering slechts 1,54 keer langer.

Die factor 1,54 varieert tussen de benchmarks: voor *205raytrace* liggen de uitvoeringstijden voor beide reservoirs het dichtst bij elkaar (1,05×), bij *228jack* is het verschil 1,62×. Dat verschil is wellicht deels te wijten aan andere, niet-onderzochte, intrinsieke eigenschappen van de benchmarkprogramma's of aan variaties tussen experimenten die optreden in de JVM, zoals de automatische geheugensanering (*garbage collection*).

Tabel 3.6: Relatief geheugengebruik tijdens het profileren

programma	origineel	volledig	$n = 10^3$	$n = 10^6$
201compress	8,8 MiB	8,16 ×	11,55 ×	21,42 ×
202jess	10,1 MiB	5,33 ×	33,72 ×	35,28 ×
205raytrace	6,7 MiB	13,77 ×	79,29 ×	69,63 ×
209db	8,4 MiB	14,68 ×	19,09 ×	25,51 ×
213javac	8,1 MiB	13,79 ×	50,72 ×	45,41 ×
228jack	4,7 MiB	10,59 ×	27,78 ×	36,93 ×

3.5.3 Geheugengebruik tijdens profilering

Tijdens de experimenten heb ik ook het gemiddelde geheugengebruik gemeten op het niveau van het besturingssysteem (Linux). De cijfers in deze paragraaf omvatten het volledige, gemiddelde geheugengebruik van de JVM tijdens de uitvoering van het Java-programma. Door het feit dat de geheugenmeting 'onder de JVM gebeurt, worden de cijfers enigszins beïnvloed door de werking van de *garbage collector*. Geheugen dat in principe niet meer gebruikt wordt door het Java-programma, maar dat nog niet werd opgekuist en vrijgegeven door de *garbage collector*, draagt nog altijd bij tot effectieve geheugengebruik op het niveau van het besturingssysteem.

Tabel 3.6 toont absolute cijfers voor het gemiddelde geheugengebruik tijdens de uitvoering van de oorspronkelijke programma's. Daarnaast geeft de tabel ook de relatieve toename weer van het geheugengebruik voor de volledige profilering zonder bemonstering en voor de bemonsterende aanpak met *reservoir sampling*. Het geheugengebruik neemt fors toe door het profileren omdat dan voor elk Java-object ook een complex schaduwobject moet worden bijgehouden met daarin voor elk veld in het oorspronkelijke object een referentie naar een actor-object.

Bij de bemonsterende aanpak met *reservoir sampling* is het geheugengebruik nog groter. Het communicatieprofiel kan nu immers niet meer eenvoudigweg incrementeel worden opgebouwd door bij elke lees- en schrijfoperatie de communicatieteller van één van de takken van de graaf te verhogen.

Voor de schrijfoperaties naar het geheugen verandert er niets. Zij worden, net als in de situatie zonder bemonstering, rechtstreeks geregistreerd in de schaduwobjecten. Maar de leesoperaties geven nu

niet meer rechtstreeks aanleiding tot een verhoging van een tellertje. Zij komen terecht in het reservoir als een afzonderlijk object met referenties naar de twee actors van de tak in de graaf waartoe ze behoren. Pas op het einde van het programma, kan de graaf opgebouwd worden door elke leesoperatie die dan nog overblijft in het reservoir, in rekening te brengen.

Het gaat maximaal over duizend of 1 miljoen objecten (de grootte van het reservoir) maar om tot die objecten te komen, wordt er voortdurend, bij elke leesoperatie, een nieuw objectje aangemaakt en een ander objectje verwijderd. De *garbage collector* slaagt er dan niet altijd in om die verwijderde objecten snel weg te werken.

Voor de meeste benchmarkprogramma's neemt het geheugengebruik, zoals verwacht, toe bij stijgende reservoirgrootte. Bij de benchmarks *205raytrace* en *213javac* is er een kleine daling merkbaar. Vermoedelijk ligt de oorzaak hiervan bij de (onvoorspelbare) *garbage collector*: bij een groter reservoir worden meer objectjes aangemaakt, waardoor het kan gebeuren dat de *garbage collector* vaker ingeschakeld wordt. Geheugen wordt dan sneller vrijgegeven waardoor het opgemeten geheugengebruik in Linux daalt. De cijfers voor het geheugengebruik sluiten in dat geval dichter aan bij het daadwerkelijk gebruikte geheugen in Java.

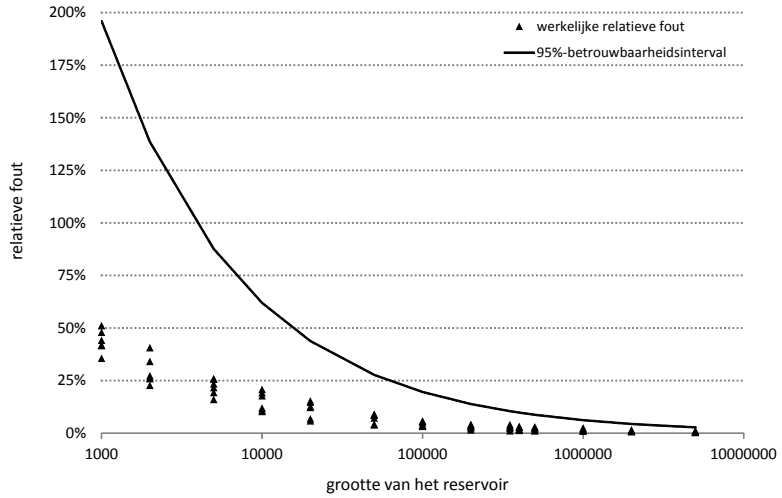
3.5.4 Nauwkeurigheid na bemonstering

Tot slot evalueerde ik de nauwkeurigheid van *reservoir sampling*. Figuur 3.5 vergelijkt de werkelijke relatieve fout (WRF) op de communicatiegraaf met de voorspelde relatieve fout zoals blijkt uit de statistische discussie in paragraaf 3.4.2. Deze werkelijke relatieve fout wordt gedefinieerd als

$$\text{WRF} = \sqrt{\frac{1}{M}\sum_{i=1}^{M}\left(\frac{\hat{F}_{e,i} - F_{e,i}}{F_{e,i}}\right)^2} \tag{3.13}$$

waarin $F_{e,i}$ het werkelijke relatieve aandeel van tak i in communicatiegraaf G voorstelt en $\hat{F}_{e,i}$ het geschatte relatieve aandeel van tak i in de bemonsterde communicatiegraaf. Enkel de M meest belangrijke takken waarvoor $F_{e,i} > 0,1\%$ worden in rekening gebracht voor de berekening van de relatieve fout.

Deze vergelijking toont aan dat de voorspelde nauwkeurigheid in de praktijk effectief bereikt werd. De volle lijn in deze figuur geeft



Figuur 3.5: Relatieve fout als functie van de grootte van het reservoir. De volle lijn geeft het 95%-betrouwbaarheidsinterval aan voor de relatieve fout op zeer kleine takken die staan voor 0,1% van de totale communicatie. De meetpunten geven de werkelijke relatieve fout voor alle benchmarks en verschillende reservoirgroottes.

de voorspelde relatieve fout met een 95% betrouwbaarheidsinterval voor takken die exact 0,1% van de totale communicatie in het programma voorstellen. Als alle takken effectief zo klein zouden zijn, dan zou de werkelijke fout dus met een probabiliteit van 95% onder deze curve liggen, en met een probabiliteit van 5% erboven. In de praktijk blijkt dat de meeste takken in de communicatiegraaf staan voor meer tot veel meer dan 0,1% van de totale communicatie. Deze belangrijke takken kunnen veel nauwkeuriger geschat worden, dan de kleinere takken van 0,1%. De werkelijke relatieve fout ligt dus voor alle benchmarks en alle reservoirgroottes veel lager dan de berekende 95%-kansgrens voor de fout.

3.6 Communicatieprofiel mét datastructuren

Tot nu toe kwam in dit hoofdstuk enkel het communicatieprofiel aan bod dat abstractie maakt van de concrete datastructuren waarlangs de communicatie verloopt en louter communicatie tussen de actors

toont. In deze paragraaf bespreek ik het alternatieve model dat wél rekening houdt met de datastructuren. Dit communicatieprofiel met datastructuren kan eenvoudig incrementeel worden opgebouwd tijdens de uitvoering van het programma en zal als basis dienen voor de communicatiebewuste dataplaatsing in hoofdstuk 5.

3.6.1 Het algemene concept

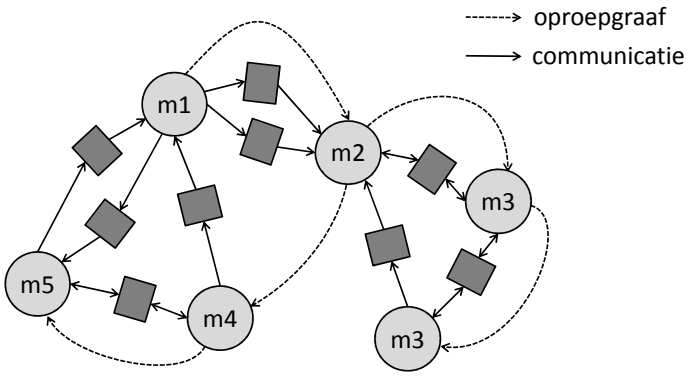
Figuur 3.6 toont hoe dit communicatieprofiel eruit ziet. Het bestaat opnieuw uit twee grafen: de oproepgraaf van het programma en een communicatiegraaf. De oproepgraaf, in feite een boom, bevat alle methodes in het programma en laat zien door welke andere methodes ze worden opgeroepen. De methodes (of actors) worden in figuur 3.6 weergegeven als cirkels. Methodes *m2*, *m4* en *m5* in figuur 3.6 worden opgeroepen door *m1*. Bijzonder is methode *m3* die zowel door *m2* wordt opgeroepen als, recursief, door zichzelf. De twee verschillende oproepen van deze methode komen als afzonderlijke knopen voor in de oproepgraaf.

De communicatiegraaf bevat twee types knopen: de actors en de datastructuren. De cirkelvormige knopen in het communicatieprofiel zijn de actors, de vierkante knopen zijn de datastructuren. De communicatiegraaf is een gewogen, gerichte graaf waarin het gewicht van de takken van actors naar datastructuren aangeeft hoeveel bytes door een actor werden geschreven naar een specifiek object. Het gewicht van de takken van datastructuren naar actors geeft aan hoeveel bytes door een actor gelezen werden uit een specifiek object.

Dit communicatieprofiel bevat zowel alle informatie over de interne communicatie in het programma als een beschrijving van het programmaverloop. De actors in de communicatiegraaf, die de communicatie tussen actors en datastructuren bevat, komen immers allemaal terug in de dynamische oproepgraaf.

3.6.2 Objecten clusteren per creatieplaats

Dit communicatieprofiel met datastructuren, heb ik opgemeten voor een aantal programma's uit de SPECjvm en DaCapo benchmark suites (Blackburn et al., 2006; Standard Performance Evaluation Corporation, 1998, 2008). Tabel 3.7 geeft voor elk van die programma's een aantal kenmerkende cijfers. De tabel toont onder meer in de eerste kolom het aantal methodes in de oproepgraaf van deze programma's, oftewel het aantal actors in de communicatiegraaf.



Figuur 3.6: Een concreet voorbeeld van communicatieprofiel met datastructuren. Dit profiel bestaat uit de oproepgraaf van het programma (stippellijn) en de datastroomgraaf (volle lijnen) die weergeeft hoe de actors (cirkels) communiceren met de verschillende datastructuren (vierkanten).

Uit de tweede kolom van de tabel blijkt dat deze benchmarkprogramma's zeer veel objecten aanmaken. Zodanig veel, dat het bijzonder onpraktisch zou zijn om die allemaal een afzonderlijke knoop te geven in het communicatieprofiel. Bovendien geeft het bijhouden van statistieken over één enkel object, weinig inzicht.

Vandaar mijn idee om de objecten te clusteren naar gelang de plaats in de code waar ze aangemaakt worden. Alle objecten die afkomstig zijn van dezelfde `new`-instructie in het programma, verder creatieplaats genoemd, worden in de communicatiegraaf voorgesteld door dezelfde knoop. Daarbij vertrek ik van de veronderstelling dat objecten met dezelfde creatieplaats, gelijkaardige eigenschappen hebben en een gelijkaardig communicatiepatroon delen. In hoofdstuk 5 kom ik hierop terug, en daar zal blijken dat deze veronderstelling realistisch is en bruikbare resultaten oplevert. Tabel 3.7 laat zien dat het aantal creatieplaatsen inderdaad heel wat kleiner is dan het aantal objecten. Gemiddeld over alle benchmarkprogramma's worden 214.832 objecten aangemaakt per creatieplaats.

3.6.3 Communicatie tussen actors en data opmeten

De profileerder die nodig is voor het opmeten van de communicatie tussen actors en datastructuren is heel wat eenvoudiger dan die

Tabel 3.7: Communicatie tussen actors en datastructuren

benchmark	actors	objecten	creatieplaatsen
202jess	4.276	57.110.881	906
209db	216	235.218.045	112
213javac	608.449	37.412.366	1.452
227mtrt	849	44.957.556	136
228jack	15.744	14.659.392	600
antlr	204.504	12.174.051	956
bloat	153.872	254.629.288	1.652
chart	3.985	24.823.327	952
crypto.aes	567	187.220.070	236
crypto.rsa	698	2.114.975	234
crypto.signverify	472	70.139	226
eclipse	7.065	550.339	1.564
fop	25.627	12.427.516	872
hsqldb	78.229	211.553.758	1.320
jython	762.799	69.932.033	1.512
pmd	390.837	75.350.682	306
serial	766	29.106.713	396
xalan	460	79.299	122
xml.validation	566	180.597	252

voor het eerste communicatieprofiel dat enkel communicatie meet tussen actors, omdat nu de datastructuren waarlangs de communicatie gebeurt rechtstreeks in het profiel worden opgenomen en dus niet weggeäbstraheerd moeten worden.

De basisprincipes voor het profileren zijn gelijk aan die voor het opmeten van communicatie tussen actors onderling, zoals besproken in paragraaf 3.3.2. De profileerder zal elke Java-klasse instrumenteren door middel van bytecode-transformatie op het ogenblik dat de klasse de JVM binnenkomt.

Voor het opbouwen van de oproepgraaf is er geen enkel verschil met mijn eerste profileerder die abstractie maakt van de datastructuren. Ook deze nieuwe profileerder voegt bij de start van elke methode code toe die controleert of er voor deze methode al een bijbehorende knoop bestaat in de oproepgraaf. Als dat niet het geval is, gaat het om een nieuwe actor en wordt die als nieuwe knoop aan de oproepgraaf toegevoegd. De nieuwe actor wordt gelinkt aan de actor van de oproepende methode.

Het opbouwen van de communicatiegraaf is fundamenteel verschillend en veel eenvoudiger voor dit communicatieprofiel met zowel actors als datastructuren. Bij elke lees- of schrijfoperatie naar het geheugen wordt de `Profiler` opgeroepen met twee parameters: een referentie naar de actor die de geheugentoegang uitvoert en een referentie naar het object waarom het gaat. Elk object wijst nu via een globale map naar zijn eigen creatieplaats. Voor elke creatieplaats is er tijdens de fase van het instrumenteren waarbij de profileerder elke klasse onder handen neemt, immers al een objectje aangemaakt dat deze creatieplaats voorstelt en dient als knoop in de communicatiegraaf. Het enige wat de klasse `Profiler` moet doen, is die creatieplaats opzoeken en een tak toevoegen in de communicatiegraaf tussen de gegeven actor en de creatieplaats ofwel de reeds bestaande tak opzoeken en het gewicht van die tak verhogen.

3.6.4 Gedeeltelijk en incrementeel opbouwen

Afhankelijk van de toepassing hoeft dit communicatieprofiel niet altijd volledig opgemeten te worden. Soms is slechts een deel van de informatie nodig. Bovendien kan dit communicatieprofiel perfect incrementeel opgebouwd worden. Op elk ogenblik is een communicatieprofiel 'tot nu toe' beschikbaar en de profileerder kan dus op elk ogenblik stopgezet worden. Ook het omgekeerde is mogelijk: de

profileerder kan eventueel pas ingeschakeld worden nadat het programma al een tijdje bezig is.

Bij het eerste communicatieprofiel dat abstractie maakt van de datastructuren, was het niet mogelijk om het profiel slechts gedeeltelijk op te bouwen, omdat dat pas correct kan gebeuren als *alle* schrijfoperaties van bij de start van de uitvoering geregistreerd werden. Dit om de volledige boekhouding van de schaduwobjecten niet in de war te sturen. Elke leesoperatie moet immers expliciet teruggekoppeld worden naar de laatste schrijfoperatie naar dezelfde geheugenlocatie.

Voor dit tweede communicatieprofiel tussen actors, worden lees- en schrijfoperaties volledig los van elkaar behandeld, wat dus een grotere flexibiliteit mogelijk maakt. Dezelfde flexibiliteit komt ook van pas als men het profiel bemonsterend wil opmeten. Een klassieke bemonsteringsmethode waarbij om de n lees- en schrijfoperaties, 1 operatie wordt geregistreerd en de andere $n - 1$ compleet genegeerd worden, is dus perfect geschikt om dit communicatieprofiel op te meten.

In hoofdstuk 5 maak ik intensief gebruik van de mogelijkheid om het communicatieprofiel met datastructuren incrementeel en gedeeltelijk te meten, om uiteindelijk te komen tot een communicatiebewuste plaatsing van Java-objecten in het gedistribueerde geheugen van de hardwareversnelde JVM. Deze JVM werkt op twee rekenkernen die optreden als processor en co-processor, elk met hun eigen lokale geheugen. Om een efficiënte geheugenallocatie te verkrijgen in een dergelijk systeem, is het van belang om een goed beeld te hebben van welke rekenkern, hoe vaak, communiceert met welke objecten in het geheugen.

Details over de specifieke actors die de geheugentoegangen uitvoeren zijn minder belangrijk. Daarom zal ik voor de profilering in hoofdstuk 5 alle methodes die door het programma worden opgeroepen, groeperen in twee actors, op basis van de rekenkern waarop ze uitgevoerd worden. Wanneer deze vereenvoudiging volledig doorgevoerd wordt, blijven in het uiteindelijke systeem enkel twee tellertjes over per creatieplaats. In deze tellers kunnen dan alle datatoegangen naar de objecten van die creatieplaats geteld worden: de ene teller telt toegangen van de processor en de andere telt de toegangen van de co-processor.

3.7 Besluit

In dit hoofdstuk heb ik twee communicatieprofielen gedefinieerd die elk op hun eigen manier de communicatie in Java-programma's kunnen beschrijven. Het eerste profiel zal in hoofdstuk 4 nuttig blijken voor het functioneel partitioneren van programma's, bijvoorbeeld in de context van hardware/software co-ontwerp, het tweede profiel vormt de basis voor het communicatiebewuste geheugenbeheer dat in hoofdstuk 5 wordt uitgewerkt.

Voor het opmeten van de communicatie tussen actors onderling, zonder rekening te houden met datastructuren, heb ik een bemonsteringsmethode uit de oude doos gehaald, *reservoir sampling*, waarmee ik erin geslaagd ben om de vertraging ten gevolge van het profileren drastisch terug te dringen voor een reeks van benchmarkprogramma's uit de SPECjvm98 suite (Standard Performance Evaluation Corporation, 1998). Dit was mogelijk zonder al te veel in te boeten op de nauwkeurigheid van de analyse, in functie van de gewenste nauwkeurigheid kan immers, statisch, de minimale grootte van het reservoir vastgelegd worden, zodat een 95%-betrouwbaarheidsinterval gegarandeerd wordt.

Het tweede communicatieprofiel dat zowel de actors als de datastructuren in rekening brengt, bewijst in hoofdstuk 5 zijn nut voor een zelflerend algoritme voor communicatiegerichte optimalisatie van de plaatsing van Java-objecten in het geheugen van de hardwareversnelde JVM. Het kan snel en efficiënt opgemeten worden, maar concretere resultaten zijn terug te vinden in hoofdstuk 5.

Hoofdstuk 4

Functionele partitionering

Partitionering is belangrijk om verschillende redenen. Vooreerst maakt het functioneel opdelen van programma's het mogelijk om verschillende onderdelen naast elkaar, parallel uit te voeren. Een tweede reden voor partitionering geldt voor heterogene platformen waarop het noodzakelijk is dat een programma gepartitioneerd is om elke partitie te kunnen toewijzen aan de meest geschikte rekenknoop die de beste prestaties levert of de meest energiezuinige implementatie. Tenslotte kan partitionering ook inzicht verschaffen in de structuur van een programma, wat nuttig is voor de analyse van programma's.

Cruciaal bij het partitioneren van de functionaliteit van programma's, in dit werk kortweg functionele partitionering genoemd, is de verhouding tussen de hoeveelheid berekeningen in een partitie en de hoeveelheid communicatie tussen deze partitie en alle andere partities. Het heeft immers geen zin om een deel van de functionaliteit van het programma af te zonderen in een afzonderlijke partitie als de kost die daarmee gepaard gaat groter is dan de verwachte winst. Dit hoofdstuk bouwt dan ook voort op de communicatiemetingen van de profileerder besproken in hoofdstuk 3, om deze partitioneringskost in rekening te kunnen brengen.

Er bestaan twee verschillende soorten partitionering: statische partitionering en dynamische partitionering. Beide types hebben hun duidelijke voor- en nadelen en hun specifieke toepassingen. In dit hoofdstuk werk ik een methode uit voor statische hardware/software-partitionering in het kader van de hardwareversnelde Java Virtuele Machine (JVM). Ik geef ook een aanzet voor de uitbreiding van deze statische methode naar een volledig dynami-

sche aanpak die binnen het proces van de Just-in-Time (JIT) compiler geïntegreerd zou kunnen worden.

4.1 Communicatiebewuste partitionering

Partitionering is die stap in het ontwerpproces waarbij de functionaliteit (code) van een programma wordt opgesplitst in verschillende stukken die op afzonderlijke rekenknopen kunnen worden uitgevoerd. In dit werk ga ik op zoek naar een communicatiebewuste partitionering (Bertels et al., 2008).

Het partitioneren van programma's leidt onvermijdelijk tot een toename van de totale communicatiekost omdat communicatie die oorspronkelijk binnen de grenzen van een partitie en dus ook binnen één rekenknoop bleef, nu tussen verschillende rekenknopen moet lopen. Communicatiebewuste partitionering streeft er naar die communicatiekost zo klein mogelijk te houden.

Partitionering kan gezien worden als het zoeken naar evenwicht in een complex samenspel tussen twee krachten. Een eerste kracht probeert de componenten van een systeem bij elkaar te houden, terwijl een tweede kracht ze juist uit elkaar drijft. De samenhoudende kracht in deze evenwichtsoefening is het streven naar minimale communicatie tussen de rekenknopen. De uit elkaar drijvende kracht is in dit geval de potentiële snelheidswinst door parallellisatie of door eenvoudigweg sneller of efficiënter uitvoeren van een deel van de functionaliteit van het programma op een andere rekenknoop van de heterogene hardware.

4.2 Functionaliteit afleiden naar een co-processor

Er bestaan verschillende vormen van functionele partitionering. In dit werk gaat het over het identificeren van delen van de functionaliteit van een systeem die geschikt zijn om ze af te zonderen van de kern van het systeem, bijvoorbeeld om ze uit te voeren op een co-processor.

Na een korte introductie van hardware/software-partitionering gebruik makend van co-processors, beschrijf ik hoe dit gebeurt in een hardwareversnelde JVM en welke eisen dit concrete platform stelt aan een goede partitionering.

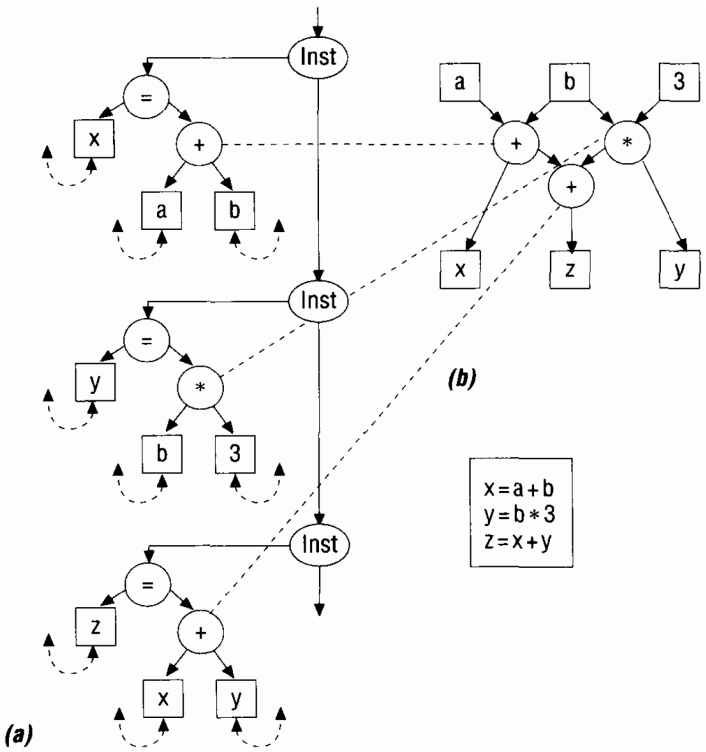
4.2.1 Het ontstaan van hardware/software co-ontwerp

De term hardware/software co-ontwerp is ontstaan aan het begin de jaren 1990. Een historische mijlpaal in de geschiedenis van co-ontwerp was de CODES-workshop in Colorado in 1992. Daar presenteerden zowel Gupta & De Micheli (1993) als Ernst et al. (1993) hun aanpak voor hardware/software co-ontwerp, respectievelijk het Vulcan-project en het Cosyma-systeem.

Beide systemen verdelen de functionaliteit van het programma incrementeel in twee delen, een hardware- en een softwaredeel, met als doel het programma uit te voeren op een generieke Central Processing Unit (CPU) met een Application-Specific Integrated Circuit (ASIC) als co-processor.

Hoewel het algemene uitgangspunt en de doelstelling van Vulcan en Cosyma gelijklopend zijn, is de manier waarop het probleem wordt aangepakt fundamenteel verschillend. De aanpak van Gupta & De Micheli (1993) vertrekt van het idee dat specifiek ontworpen hardwareblokken de snelste uitvoering van de code garanderen. Maar omdat ASIC's duur zijn, wordt ervoor gekozen een deel van de functionaliteit toch in software uit te voeren. In een volledige hardware-implementatie zal Vulcan stap voor stap hardwareblokken vervangen door equivalente, maar tragere, software. Vulcan gaat hiermee door zolang de prestatie-eisen van het systeem voldaan zijn.

Cosyma volgt de omgekeerde weg en vertrekt van een volledige software-implementatie die eigenlijk te traag is. Stap voor stap worden dan delen van het programma vervangen door hardwareversnellers tot het systeem uiteindelijk aan de opgelegde snelheidseis voldoet. Cosyma vertrekt van een programma dat volledig is geschreven in C^x, een subset van C. De C^x-code wordt eerst omgezet in een *extended syntax graph* zoals weergegeven op figuur 4.1. Deze grafische voorstelling bevat zowel de syntactische structuur van de originele code (a) als een datastroomgraaf met de elementaire operaties (b). Het Cosyma-systeem zal de *extended syntax graph* partitioneren en iteratief verschillende partities omzetten in C-code voor de CPU of in HardwareC voor een specifiek hardwareblok op de ASIC, waarna het volledige systeem gesimuleerd wordt. Dit iteratieve proces gaat door tot een oplossing werd gevonden die aan de prestatiecriteria voldoet.



Figuur 4.1: Grafische voorstelling van het systeem in Cosyma. Deze figuur toont een deel van de *extended syntax graph* met de structuur van de originele code (a) en het bijbehorende deel van de datastroomgraaf (b). Bron: Ernst et al. (1993).

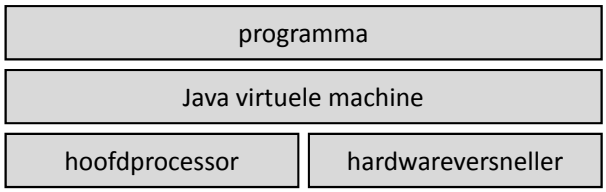
4.2.2 Hardware/software co-ontwerp in Java

In dit werk steun ik op de hardwareversnelde JVM van Faes et al. (2004) die zó is opgebouwd dat ze de complexiteit van de communicatie en synchronisatie tussen een generieke processor en een co-processor op een Field Programmable Gate Array (FPGA), perfect weet te beheersen. Doordat de complexiteit van het hardware/software-platform verborgen zit in de abstractie van de JVM hoeft de programmeur zich hierover geen zorgen meer te maken (figuur 4.2). Hij kan zijn Java-programma schrijven zonder rekening te houden met hardwareversnelling. Later kan een hardwareontwerper dan een hardwareversneller ontwerpen voor één of meerdere methodes van het Java-programma. Het volstaat dan om de FPGA te configureren met deze hardwareversneller, en de JVM hiervan op de hoogte te brengen door middel van een klein configuratiebestand dat beschrijft op welk adres de FPGA aangesproken kan worden en welke Java-methodes door deze versneller uitgevoerd kunnen worden. Daarna handelt de JVM alles zelf verder af.

De hardwareversnelde JVM voorgesteld door Faes et al. (2004) onderschept alle methode-oproepen in het Java-programma waarvoor een specifieke hardwareversneller aanwezig is. In dat geval wordt de oproep doorgestuurd en wordt de uitvoering van het programma overgeheveld naar de hardwareversneller. Deze JVM houdt ook rekening met het geheugen: hij zorgt ervoor dat zowel de hardwareversneller als de hoofdprocessor alle objecten op de Java *heap* kunnen gebruiken. Dat wil zeggen dat de hardwareversneller gebruik kan maken van alle data die in het Java-programma aangemaakt werd. Het gaat zelfs omgekeerd: de co-processor op de FPGA kan zelf ook objecten aanmaken, zowel in het hoofdgeheugen van de computer als in zijn eigen lokaal geheugen op het FPGA-bord. De JVM wordt hiervan op de hoogte gebracht en zal, volledig transparant voor de programmeur en voor de hardware-ontwerper, het geheugenbeheer hiervan op zich nemen. De Java-methodes die in software worden uitgevoerd, zullen dus eveneens gebruik kunnen maken van de objecten die in het geheugen op het FPGA-bord zijn opgeslagen.

4.2.3 Equivalentie tussen hardware en software

Specifiek aan het concept van een hardwareversnelde JVM zoals uitgewerkt door Faes et al. (2004), is dat de hardwareversneller integraal deel uitmaakt van het platform, maar dat die hardware vol-



Figuur 4.2: De Java Virtuele Machine vormt een abstractielaag die de complexiteit van het onderliggende, hybride platform verbergt.

ledig transparant blijft voor de Java-programmeur. Hiervoor moet uiteraard de equivalentie tussen een hardwarecomponent en softwareconcepten uit Java exact gedefinieerd worden.

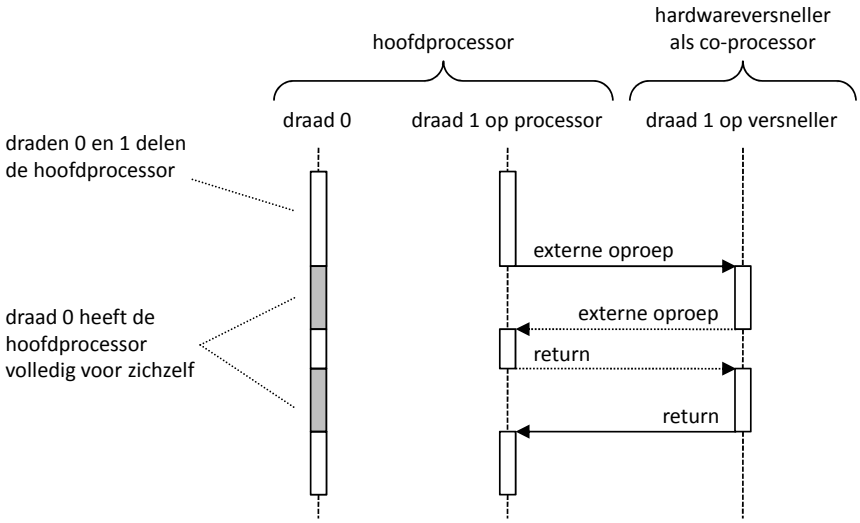
In de hier gebruikte aanpak omvatten de hardwareversnellers het functionele gedrag van de bytecode in een overeenkomstige Java-methode. Deze hardwareversneller heeft dus geen interne toestand. Bij elke oproep moeten zowel de parameters van de methode als de bijbehorende toestand —de overeenkomstige Java-klasse voor statische methodes of een objectreferentie voor virtuele methodes— naar de hardwarecomponent doorgestuurd worden.

De JVM gebruikt dezelfde aanpak: software-objecten omvatten in principe zowel de toestand als de functionaliteit, maar tijdens de uitvoering worden beide aspecten gescheiden. Statische informatie over een specifieke klasse en de Java-bytecode voor de bijbehorende functionaliteit worden slechts één keer opgeslagen, terwijl de toestand bijgehouden wordt per geïnstantieerd object. Deze manier om equivalentie te bereiken tussen de hardwareversneller en Java-methodes werd in detail beschreven door Faes (2008) en is vergelijkbaar met de aanpak voorgesteld door Borg et al. (2006).

4.2.4 Communicatie tussen hardware en software

Methode-oproepen waarvoor een hardware-equivalent beschikbaar is, worden onderschept door de JVM. De uitvoering van de huidige draad wordt overgeheveld naar de hardwareversneller, tenzij die niet beschikbaar is —bijvoorbeeld wanneer de hardwareversneller gebruikt wordt door een andere draad. In dat laatste geval wordt de Java-versie van deze methode gewoon op de hoofdprocessor uitgevoerd.

De communicatie tussen de hoofdprocessor en de hardwareversneller is gebaseerd op *remote calls*. Figuur 4.3 toont twee van de-



Figuur 4.3: De communicatie tussen de hoofdprocessor en de hardwareversneller verloopt via methode-oproepen. Draden uitgevoerd op de processor kunnen de versneller opstarten, die op zijn beurt de uitvoering opnieuw kan overhevelen naar de processor, bijvoorbeeld voor het opwerpen van *exceptions*.

ze oproepen. De eerste externe methode-oproep, weergegeven als een volle lijn, gebeurt door de draad die uitgevoerd wordt op de hoofdprocessor. Deze externe oproep start de hardwareversneller. De parameters van de methode worden doorgegeven door middel van referenties. De hoofdprocessor stopt de uitvoering van de huidige draad zolang de hardwareversneller bezig is. Ondertussen kan de hoofdprocessor andere draden blijven uitvoeren.

Wanneer de hardwareversneller een andere Java-methode oproept die geen hardware-equivalent heeft of eenvoudigweg onmogelijk te implementeren is in hardware, dan kan de hardwareversneller op zijn beurt de hulp inroepen van de hoofdprocessor via een *callback*-mechanisme. Dit kan bijvoorbeeld gebruikt worden voor het lezen of schrijven van bestanden of voor het opwerpen van *exceptions*. Figuur 4.3 geeft een voorbeeld van een dergelijke *callback*, opnieuw als een externe oproep, in stippellijn.

4.3 Statische partitionering

Met statische partitionering wordt in dit onderzoek bedoeld het partitioneren van de functionaliteit van een programma vóór het uitgevoerd wordt. Men zou dus ook kunnen spreken over *offline* partitioneren.

Om de partitionering op een communicatiebewuste manier te doen, wordt uitgegaan van profileringsresultaten waarbij de communicatie tussen verschillende methodes in het programma wordt opgemeten, zonder rekening te houden met de datastructuren waarlangs de communicatie loopt. Ik ga er immers vanuit dat het resultaat van de partitionering wordt gebruikt om het programma, minstens gedeeltelijk, te herimplementeren op een nieuw platform met meerdere rekenkernen. Dat zou een multicore-platform kunnen zijn of een heterogene Multiprocessor System-on-Chip (MPSOC) met daarin één of meer algemene processors gecombineerd met bijvoorbeeld specifieke hardwareblokken of Digital Signal Processors (DSP's). Op dat moment moet de programmeur eigenhandig beslissen over hoe de data in het geheugen wordt opgeslagen: soms vergt dit een aanpassing van de datastructuren ten opzichte van de software-implementatie. Hiervoor moet de programmeur een idee hebben van de instrinsieke datacommunicatie in het programma zodat hij die in de datastructuren kan optimaliseren.

Zoals beschreven in hoofdstuk 3, geeft deze manier van profileren, zonder rekening te houden met de datastructuren, een goed beeld van de effectieve communicatie in het programma. Deze werkelijke communicatie komt niet altijd rechtstreeks overeen met de manier waarop datastructuren in de initiële implementatie werden opgebouwd en gebruikt.

Beschouw, bij wijze van voorbeeld, een beeldverwerkingsalgoritme dat werkt in verschillende stappen. Een beeld, in het programma voorgesteld als een matrix van beeldpunten, doorloopt vier filterstappen die steeds het volledige beeld filteren en de resultaten hiervan telkens opnieuw wegschrijven in dezelfde matrix. Het communicatieprofiel dat de originele datastructuren mee in rekening brengt, zal enkel weergeven dat vier verschillende Java-methodes elk lezen en schrijven van hetzelfde beeld-object. Door die datastructuren niet mee te nemen in de analyse, maakt het communicatieprofiel onmiddellijk duidelijk dat het om sequentiële stappen gaat waarbij de eerste filterstap beeldpunten genereert die door de tweede worden ge-

bruikt enzovoort.

Elke filterstap zal dus resultaten doorgeven aan de volgende en enkel data lezen die in de vorige stap geschreven werd. In de uiteindelijke implementatie kan de ontwerper rekening houden met het *streaming* gedrag van deze filteroperatie.

4.3.1 Verschillende manieren om te partitioneren

Het partitioneren van een systeem kan op verschillende manieren gebeuren. In dit doctoraatsonderzoek heb ik me toegelegd op het partitioneren van programma's op basis van effectief uitvoerbare programma's in Java. Een dergelijke uitvoerbare specificatie van het systeem heeft onder meer als voordeel dat ze door middel van profilering geanalyseerd kan worden (Bertels & Stroobandt, 2006; Helaihel & Olukotun, 1997). Voor communicatiebewuste partitionering bedoeld in dit werk, bestaat die analyse allereerst uit het opmeten van de communicatie in het Java-programma.

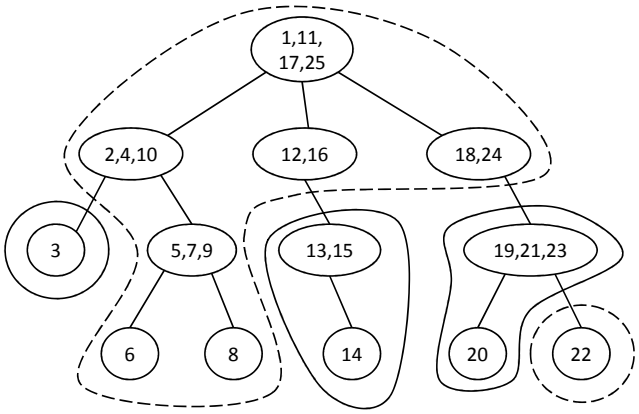
Andere manieren om systemen te partitioneren vertrekken nogal vaak van abstracte taakgrafen die de verschillende taken in het systeem en de communicatie daartussen modelleren. Het grote voordeel van dergelijke modellen is dat ze wiskundig correct en in detail bestudeerd kunnen worden. Nadeel is echter dat deze taakgrafen moeilijk op te stellen zijn voor echt complexe programma's met bijvoorbeeld een belangrijk invoerafhankelijk gedrag. En als het toch lukt om daarvoor een model op te stellen, dan zijn de resultaten die uit de analyse daarvan voortvloeien vaak weinig precies. In programma's met data-afhankelijk gedrag kan er immers een grote spreiding zitten op de hoeveelheid communicatie.

Hier duikt dus opnieuw het verschil op tussen statische en dynamische analyse dat ik in paragraaf 3.1.2 al aanhaalde. Een statische analyse kan in het beste geval de aanwezige spreiding op de hoeveelheid communicatie exact inschatten.

4.3.2 Splitsen volgens de oproepgraaf

In de hardwareversnelde JVM van Faes et al. (2004) waar dit werk op voortbouwt, zal elke hardwareversneller functioneel equivalent zijn met een Java-methode of een groep van Java-methodes, zoals aangehaald in paragraaf 4.2.3.

Gezien de opbouw van de hardwareversnelde JVM is het dus interessant om bij partitionering te splitsen op methodegrenzen. Daar-



Figuur 4.4: In de oproepgraaf van een draad in het programma, wordt elke oproep van een methode voorgesteld door een knoop. De volgorde waarin de methodes worden opgeroepen is hier weergegeven met nummers. De oproepgraaf wordt dus diepte-eerst doorlopen. De delen die uitgevoerd worden in hardware of in software zijn telkens subgrafen van deze oproepgraaf.

om heb ik bij het profileren van de communicatieprofielen steeds de dynamische oproepgraaf van het programma opgemeten. Deze oproepgraaf is een boom met als knopen de verschillende methodes in het programma, de takken geven aan welke methode welke andere methode oproept. Methodes die door meerdere methodes worden opgeroepen, komen ook meerdere keren voor in de graaf.

De hardwareversneller kan, als dat nodig is, ook terug de software oproepen, maar dat is een dure operatie die best vermeden wordt. Het is dus verstandig om niet enkel op te splitsen bij een methodegrens, maar om meteen ook alle methodes die door die afgesplitste methode worden opgeroepen, mee te nemen in hardware. Met andere woorden: het is altijd best om subbomen van de oproepgraaf —die in feite een oproepboom is— af te splitsen. Figuur 4.4 toont hoe dit in zijn werk gaat. De grootste partitie, die de eerst uitgevoerde *main*-functie (nummer 1 in figuur 4.4) bevat, wordt uitgevoerd op de hoofdprocessor. Deze partitie is omcirkeld met een streepjeslijn. Dan zijn er drie partities die afgesplitst worden naar de co-processor, in figuur 4.4 aangeduid met een volle lijn. Tot slot is er methode 22 die niet in de hardware uitgevoerd kan worden en waarvoor dus een *callback* naar de software nodig is.

4.3.3 Criteria voor functionele partitionering

Verderop in dit werk beschrijf ik het partitioneringsalgoritme om programma's volgens hun oproepgraaf op te delen in verschillende stukken. Programma's worden zo opgesplitst dat het uiteindelijke systeem sneller en efficiënter gebruik kan maken van de beschikbare hardware.

Er zijn verschillende mogelijkheden om de efficiëntie te verhogen door partitionering.

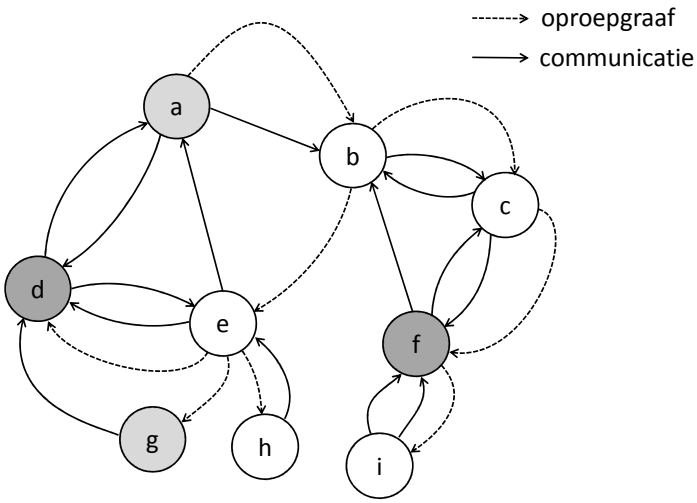
Allereerst maakt partitionering het mogelijk om verschillende delen van het programma te spreiden over de beschikbare rekeneenheden. Deze manier van parallellisatie kan een aanzienlijke versnelling van de uitvoering van het programma bewerkstelligen.

Een tweede interessante mogelijkheid voor efficiëntieverhoging is mogelijk op heterogene platformen waarvan de verschillende rekeneenheden elk gespecialiseerd zijn in een bepaald deel van de berekeningen in het programma. Na partitionering kan voor elk deel van het programma de meest geschikte rekenknoop uitgekozen worden. Dit is bijvoorbeeld het geval wanneer een deel van het programma geschikt is voor hardwareversnelling met behulp van een specifiek daarvoor ontworpen hardwareblok of een specifieke coprocessor. Het is dit type van efficiëntiewinst dat ik beoog in dit werk en dat ik verder in detail uitwerk in hoofdstuk 5.

Beide criteria, parallellisatie en (hardware-)versnelling, moeten bij partitionering in evenwicht worden gebracht met het criterium dat de bijkomende communicatiekost door het opsplitsen van de berekeningen de verwachte winst uiteraard niet mag teniet doen.

4.3.4 Het algoritme

Het algoritme bestaat uit twee stappen. Eerst wordt voor een aantal methodes uit de oproepgraaf bepaald of ze wel of niet in de hoofdpartitie thuis horen. Dit kan gebeuren op basis van verschillende criteria: manuele selectie door een ontwerper of op basis van gegevens van een prestatieschatter. In deze eerste stap krijgt slechts een deel van de methodes in de graaf een bestemming, voor de andere methodes hangt de uiteindelijke partitie waarin ze terechtkomen af van de tweede stap. Die tweede stap is een heuristisch zoekalgoritme dat de beste partitionering zoekt die de minste communicatie tussen de partities oplevert. Het algoritme voor de tweede stap is weergegeven in figuur 4.8.



Figuur 4.5: Handmatige selectie van rekenknopen in de oproepgraaf.

Selectie van methodes uit de oproepgraaf

De ontwerper start het partitioneringsproces door handmatig twee types methodes uit de oproepgraaf te selecteren. Een eerste reeks methodes zijn methodes die expliciet in een afzonderlijke partitie terecht moeten komen, bijvoorbeeld omdat hun functionaliteit geschikt wordt geacht voor hardwareversnelling. Een tweede reeks zijn knopen die zeker op de hoofdprocessor moeten worden uitgevoerd. In de pseudocode van figuur 4.8 wordt dit per knoop respectievelijk aangeduid als `n.voorkeur = HW` of `n.voorkeur = SW`. Figuur 4.5 toont hoe dit er dan uit ziet op de communicatiegraaf. De lichtgrijze knopen a en g zijn geselecteerd voor de hoofdprocessor, knopen d en f komen volgens de ontwerper in aanmerking voor hardware-implementatie.

Voor alle andere knopen moet dus nog een oplossing gevonden worden: ofwel gaan zij naar de softwarepartitie en worden ze uitgevoerd door de processor, ofwel gaan zij naar de hardwarepartitie en komen ze terecht op de FPGA in een co-processor.

Partitionering opbouwen van onderop

Rekening houdend met de eerder geschetste specifieke eigenschappen van de hardwareversnelde JVM, is het interessant om geen los-

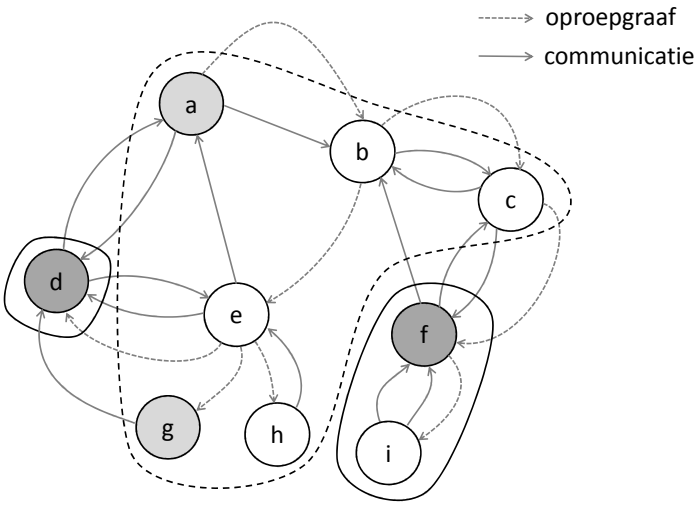
se methodes over te hevelen naar de hardware, maar om subbomen van de oproepgraaf samen te houden en ze dus ofwel volledig naar de co-processor te verplaatsen, dan wel ze volledig in de software te houden.

Dit concept van partitioneren volgens subbomen geeft, na de handmatige selectie van knopen, aanleiding tot een partitionering met een minimale afsplitsing van methodes naar de co-processor op de FPGA en een partitionering met een maximale afsplitsing. Bij de minimale afsplitsing worden enkel de methodes die geschikt zijn voor hardwareversnelling en al hun kinderen in de dynamische oproepgraaf, afgesplitst. Op figuur 4.6 is te zien dat in dit geval drie methodes naar de hardware verhuizen: methodes d, f en i. In de praktijk betekent dit dat er twee co-processors moeten worden ontworpen: één voor methode d en een tweede co-processor die gezamenlijk methodes f en i implementeert.

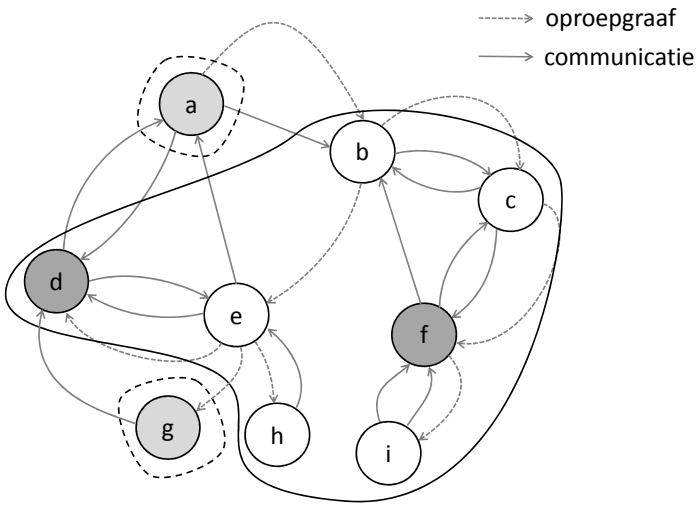
Omgekeerd is de partitionering van de maximale afsplitsing zó opgebouwd dat enkel de methodes die expliciet zijn gemarkeerd als niet geschikt voor hardware-implementatie, overblijven op de hoofdprocessor, samen met hun kinderen voor zover die niet meegenomen kunnen worden in de hardware-partitie.

Concreet wordt deze partitionering van maximale afsplitsing bereikt door, vertrekkend vanaf de methodes die in hardware geïmplementeerd worden in de minimale partitionering, stelselmatig op te klimmen in de oproepgraaf en alle ouders van die methodes mee op te nemen in de hardwarepartitie, zonder echter methodes op te nemen die expliciet gemarkeerd zijn om in software te blijven. In de eerste stap wordt methode c opgenomen in de hardware-partitie omdat al haar kinderen (in casu f) ook al in de hardwarepartitie zitten. Vanuit d wordt methode e meegenomen, samen met al haar kinderen, behalve g omdat g expliciet gemarkeerd is om niet mee te verhuizen naar de hardware. Eens e opgenomen is in de hardwarepartitie, wordt de stap gezet naar methode b. De resulterende partitionering met maximale afsplitsing staat weergegeven in figuur 4.7. Uiteindelijk blijven enkel a en g over in software, alle andere methodes worden verhuisd naar de FPGA waar één specifieke co-processor ontworpen wordt die hun functionaliteit omvat.

De optimale partitionering ligt ergens tussen de partitionering van de minimale afsplitsing en die van de maximale afsplitsing. Op het hierboven beschreven pad dat van onderop vertrekt en zo stapsgewijze de maximaal afgesplitste partitionering bereikt, moet in elke



Figuur 4.6: Minimale afsplitsing. In deze partitionering worden uiteindelijk drie methodes in de hardware geïmplementeerd op twee co-processors. Methode d zit in de eerste co-processor, methodes f en i worden samen geïmplementeerd in een tweede co-processor.



Figuur 4.7: Maximale afsplitsing. Bij deze partitionering worden enkel methodes a en g in de software uitgevoerd. Alle andere methodes komen op een specifieke co-processor terecht.

stap geëvalueerd worden of de verhouding tussen interne en externe comunicatie beter wordt, met dus meer interne communicatie of minder externe, door er nog een knoop bij te nemen (met zijn eventueel meeverhuizende kinderen). Wanneer dat niet meer het geval is, voor geen enkel van de mogelijke paden omhoog in de oproepgraaf, wordt het proces stopgezet. De partitionering die op dat ogenblik bereikt is, wordt door het algoritme naar voor geschoven als geschikte statische partitionering.

Pseudocode voor dit zoekalgoritme staat in figuur 4.8. Dit codefragment veronderstelt dat voor alle knopen `n` in de communicatiegraaf `G` die manueel geselecteerd werden voor implementatie op de hardwareversneller of op de hoofdprocessor, het veld `n.voorkeur` is ingevuld. Daarna start het eigenlijke zoekalgoritme via `partitioneer(root)`, met `root` de top van de dynamische oproepgraaf.

Hierbij dient opgemerkt dat deze geschikte partitionering niet noodzakelijk optimaal is. Om effectief de optimale verhouding tussen interne en externe communicatie op te zoeken, dient men effectief alle mogelijke partitioneringen tussen die van de minimale en de maximale afsplitsing te evalueren en met elkaar te vergelijken.

4.3.5 Partitioneren op basis van profileringsdata

De aandachtige lezer zal opgemerkt hebben dat de statische partitionering die in deze paragraaf wordt voorgesteld louter gebaseerd is op profileringsdata. Er kwam helemaal geen statische analyse aan te pas. Men zou zich kunnen afvragen of dit geen probleem is.

Door het feit dat de partitionering tot stand komt zonder enige vorm van statische analyse, kan nooit gegarandeerd worden dat deze partitionering in alle omstandigheden de beste, meest optimale, oplossing oplevert. Het is immers mogelijk dat er in de praktijk een methode-oproep gebeurt die niet in de opgemeten oproepgraaf voorkomt. De oproepgraaf is immers gebaseerd op de profilering van één enkele uitvoering van het programma. Of het zou kunnen dat er een communicatiestroom ontstaat tussen twee knopen in de graaf die tijdens profilering niet optrad. Met die situatie is dan vanzelfsprekend geen rekening gehouden in de partitionering.

Louter afgaan op profileringsdata voor de functionele partitionering van een systeem kan dus leiden tot een niet-optimale efficiëntie van de gevonden oplossing. De profilering en de partitio-

```
proc partitioneer(n):
    voor elke knoop m in n.kinderen:
        partitioneer(m)

    switch n.voorkeur:
        case HW:
            recursiefToevoegen(n, HW)
        case SW:
            recursiefToevoegen(n, SW)
        case NULL:
            if aantal(n.kinderen) > 0:
                maakEenKeuzeEnPasToe(n)
end

proc recursiefToevoegen(n, p):
    if n.partitie == NULL:
        n.partitie := p
        voor elke knoop m in n.kinderen:
            recursiefToevoegen(m, p)
end

proc maakEenKeuzeEnPasToe(n):
    beschouw volgende subgrafen:
        subgraaf_HW: n + al zijn kinderen m met m.partitie == HW of NULL
        subgraaf_SW: n + al zijn kinderen m met m.partitie == SW of NULL
        subgraaf_NULL: alle kinderen van n

    voor elke subgraaf g in { subgraaf_HW, subgraaf_SW, subgraaf_NULL }
        bereken verhouding interne/externe communicatie tussen g en G\g
        en kies subgraaf g waarvoor verhouding minimaal is

    switch minimum
        case subgraaf_HW: recursiefToevoegen(m, HW)
        case subgraaf_SW: recursiefToevoegen(m, SW)
        case subgraaf_NULL: n.partitie := NULL
end
```

Figuur 4.8: Algoritme voor statische partitionering

neringstool die hier beschreven werd, zal de ontwerper slechts een hint geven voor een mogelijke, communicatiebewuste opdeling van het programma.

De gevonden partitionering zal misschien niet altijd de meest efficiënte zijn, ze zal wel een correcte partitionering zijn die leidt tot een werkende implementatie van het systeem. Polyvalente en flexibele platformen zoals bijvoorbeeld Faes' hardwareversnelde JVM zullen ervoor zorgen dat het gepartitioneerde systeem perfect blijft werken, ook als tijdens de uitvoering een onvoorziene situatie optreedt. Extra communicatie vormt geen probleem voor de werking van het systeem omdat alle data perfect toegankelijk is voor eender welke rekenknoop (zie verder in hoofdstuk 5).

De efficiëntie kan evenwel licht dalen als de functionaliteit van het programma (de code) niet optimaal verdeeld is, maar mits een dynamische plaatsing van de data (de Java-objecten) kan dat zelfs deels opgevangen worden. Een onvoorziene methode-oproep is op zich ook geen probleem omdat het systeem zo opgebouwd is dat de hardware elke mogelijke Java-methode, zelfs ongeziene Java-methodes, kan oproepen via een *callback* naar de software van de JVM.

Op platformen waar geen enkele vorm van *callback*-mechanisme voorzien is om problemen op te vangen die optreden met de functionele partitionering of met bijkomende communicatie tijdens de uitvoering van het programma, kan men dus niet blindelings vertrouwen op het advies van de in deze paragraaf beschreven partitioneerder.

Toch blijft het interessant om op basis van profileringsdata te partitioneren. Enerzijds omdat een dergelijke functionele opdeling van het programma een interessante hint is voor manuele partitionering of optimalisatie van de partitionering. Een goede start is altijd belangrijk. Anderzijds omdat deze partitionering rekening houdt met de 'gebruikelijke' communicatiekost die werkelijk in het systeem zal optreden in de gevallen die tijdens het profileren aan bod kwamen.

Wanneer de profilering dus gebeurt op basis van realistische scenario's en wanneer men er de aandacht op vestigt dat alle meest gebruikelijke scenario's effectief aan bod zijn gekomen tijdens de profilering, dan geeft dit een realistisch beeld van de communicatie. Dit is vaak niet het geval bij een communicatieschatting of berekening op basis van statische analyse omdat die enkel een conservatief beeld oplevert. Een statische analyse garandeert dat de communica-

tie nooit meer zal zijn dan de opgegeven waarden of aantallen. Deze schatting zal dus bij definitie een overschatting zijn.

4.4 Dynamische partitionering

Statische partitionering zoals voorgesteld in de voorgaande paragraaf is zeer ruim toepasbaar. Maar voor sommige toepassingen wordt het in de toekomst misschien ook mogelijk en zelfs voordelig om de functionele opdeling uit te stellen tot op het moment van de uitvoering van het programma. Hieronder schets ik welke toepassingen kunnen profiteren van dynamische partitionering en aan welke specifieke eigenschappen het hardwareplatform moet voldoen om dit mogelijk te maken.

De technieken voor *runtime* hardware-generatie staan vandaag nog niet ver genoeg om volledige Java-methodes tijdens de uitvoering te kunnen vertalen naar hardware. Dit hoofdstuk over dynamische partitionering schept daarom louter het kader waarbinnen mijn technieken voor statische partitionering in de toekomst uitgebreid kunnen worden naar een volkomen dynamische aanpak binnen het JIT-compilatieproces.

4.4.1 Voordelen van dynamische partitionering

In hoofdstuk 2 haalde ik al enkele voorbeelden aan die aantonen dat het soms nuttig is om een aantal beslissingen uit te stellen tot het laatste moment, tijdens de uitvoering van het programma, in plaats van alles op voorhand te bepalen en vast te leggen.

De belangrijkste reden hiervoor was het onvoorspelbare gedrag van sommige toepassingen. Het gedrag van een programma hangt vaak af van de concrete invoer. De werklast kan behoorlijk groter of kleiner zijn voor specifieke invoersets. Daarnaast is er ook een invloed mogelijk op het functieverloop van een programma, zodat de oproepgraaf van het programma niet altijd gelijk blijft.

Omdat programma's niet altijd op dezelfde manier reageren of functioneren, kan een statische compiler vooraf dus niet altijd de gepaste beslissingen nemen. Sommige optimalisaties kunnen immers in de ene uitvoering wel winst opleveren, terwijl ze in een andere uitvoering nadelig uitdraaien. Een goed voorbeeld hiervan is de vraag om bepaalde methode-oproepen al dan niet te inlijnen, meteen

één van de belangrijkste dynamische optimalisaties in het Dynamo-project bij Hewlett-Packard (Bala et al., 2000).

Methodes inlijnen heeft als duidelijke voordeel dat de extra kost van een methode-oproep bespaard wordt, maar het nadeel is dat er code voor gedupliceerd moet worden. Het is dus niet zinnig om alle methodes in te lijnen, dit zou immers leiden tot een te drastische toename van de grootte van het programma, een explosie als het ware. De compiler moet hierbij dus selectief te werk gaan, maar omdat die compiler op voorhand moeilijk kan inschatten hoe vaak elke methode vanuit elke andere methode opgeroepen wordt, kan de compiler niet afwegen of de duplicatie van de code al dan niet zin heeft. Door deze beslissing dynamisch te nemen en pas na het veelvuldig uitvoeren van een specifieke methode-oproep vanuit een andere methode, over te gaan op hercompilatie met inlijnen van de opgeroepen methode, kon Dynamo de uitvoeringstijd van sommige benchmarkprogramma met 20% reduceren.

Het vraagstuk van de partitionering is gelijkaardig: het draait immers allemaal om de verhouding tussen berekeningen en communicatie. Nu eens is het afsplitsen van een stukje code naar een afzonderlijke partitie een goed idee, wanneer de hoeveelheid berekeningen voldoende is om de bijkomende communicatiekost te verantwoorden, dan weer is het veel beter om hetzelfde stukje code bij het hoofdprogramma op dezelfde partitie te houden. Uit metingen voor een DNA-alignatie-algoritme in het kader van het FlexWare-project (Meeus, 2007) blijkt dat deze verhouding zeer sterk kan verschillen afhankelijk van de lengtes van de DNA-sequenties die vergeleken moesten worden. Voor sequenties korter dan enkele tientallen elementen, is hardwareversnelling volstrekt zinloos. Het is dus niet zo vreemd om dit mee te nemen bij dynamische partitioneringsbeslissingen.

4.4.2 Platformen voor dynamische partitionering

Een platform moet aan vier cruciale voorwaarden voldoen om dynamische partitionering mogelijk te maken.

Centrale sturing en controle. Het platform moet een centrale rekenknoop bevatten die het overkoepelende werk kan doen om de nodige parameters op te meten die dienen als basis voor de partitionering: communicatieprofilering tijdens de uitvoering. Deze reken-

knoop zal dan ook instaan voor de coördinatie en synchronisatie tussen de verschillende partities. Hij zal op het juiste moment de juiste methodes in het programma afleiden naar de correcte co-processor.

Terugval-principe. Er wordt dynamisch beslist om stukken code af te splitsen van de hoofdpartitie of om ze niet af te splitsen. Om dit systeem in alle omstandigheden vlekkeloos te doen werken is het dus belangrijk dat het platform minstens één centrale rekenknoop bevat die in staat is om het volledige programma zelf uit te voeren, zonder partitionering en hulp van co-processors.

Meerdere rekenknopen. Partitionering werkt uiteraard pas als er één of meer andere rekenknopen aanwezig zijn om taken over te nemen van de hoofdprocessor. Bovendien moeten deze rekenknopen in staat zijn om specifieke methodes sneller en efficiënter uit te voeren dan de hoofdprocessor. Anders heeft partitionering geen zin.

Communicatieplatform. Het moet mogelijk zijn voor de hoofdprocessor en de co-processors om met elkaar te communiceren. In de hardwareversnelde JVM werkt dit op basis van gedeeld geheugen. Het systeem moet ervoor zorgen elke methode in het programma aan de data kan die ze nodig heeft, ook als deze methode in deze specifieke uitvoering niet op de hoofdprocessor, maar op een andere rekenknoop wordt uitgevoerd.

4.4.3 Verwante aanpakken in de literatuur

Combinatie van statische en dynamische partitionering

In het zJava-project onderzochten (Chan & Abdelrahman, 2004) de automatische parallellisatie van Java-programma's. Het centrale idee in dit project is het combineren van statische en dynamische methodes om potentieel parallellisme tussen methodes in het Java-programma uit te buiten. Het zJava *runtime system* start een nieuwe draad op voor elke methode-oproep. Deze draad wordt asynchroon uitgevoerd met de hoofd-draad van het programma die de *main*-methode bevat. Tijdens het compileren wordt een statische analyse uitgevoerd die het werk voorbereidt. Van elke methode wordt bijgehouden welke afhankelijkheden er zijn ten opzichte van andere methodes. Tijdens de uitvoering worden die afhankelijkheden dan

op het juiste moment afgedwongen door de juiste synchronisatiemechanismen.

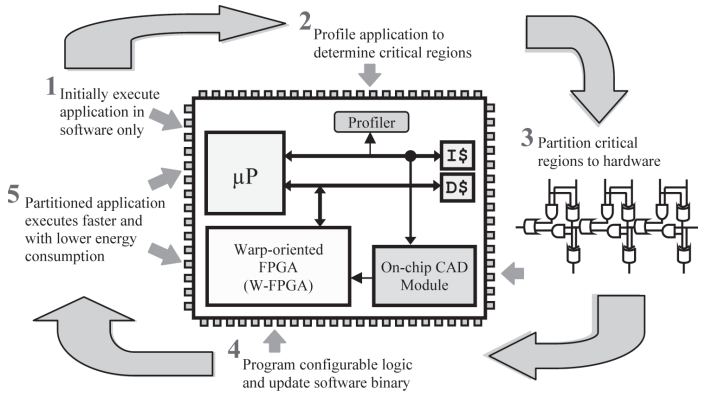
Chan & Abdelrahman behaalden met deze aanpak een schaalbare versnelling gebruik makend van 1, 2, 3 of 4 kernen van een Sun quadcore multiprocessor. Het zJava-project was erg succesvol: voor een aantal benchmarkprogramma's werd de ideale versnelling bereikt, gelijk aan het aantal gebruikte rekenkernen.

Volledig dynamische partitionering

De eerste succesvolle aanpak voor een volledig dynamische hardware/software partitionering is wellicht het concept van de WARP-processor (Lysecky et al., 2006) in figuur 4.9. Deze nieuwe processorarchitectuur is ontworpen om de snelheid en het energieverbruik van een standaard softwareprogramma te optimaliseren. WARP doet dit volledig transparant voor de programmeur. Er is geen speciale compiler of ander voorbereidend werk voor nodig omdat het systeem werkt met klassieke software *binaries*. Tijdens de uitvoering identificeert de profileerder, die ingebed is in de WARP-processor, de hete code in het programma. Op de chip zit ook een CAD-module die voor deze codefragmentjes een specifiek hardwareblokje uitwerkt en de ingebouwde FPGA correct configureert. Eens dat gebeurd is, worden de instructies van dit codefragment in de instructie-*cache* vervangen door een oproep naar het hardwareblok op de FPGA.

Het concept achter WARP werd gelanceerd door Vahid et al. (2001). De basis waarop de versnelling van het programma in de hardwareblokken steunt, is constantenpropagatie in de datastroomgraaf. De eerste echte WARP-processor is het resultaat van het samenkomen van zeer divers onderzoek waaronder dat van decompilatie van *binaries* (Stitt & Vahid, 2005) en snelle synthese van de hardware (Stitt & Vahid, 2007) alsook de ontwikkeling van een speciale FPGA-structuur die het mogelijk maakt om zeer snelle plaatsings- en routeringsalgoritmes te gebruiken.

Het eindresultaat was erg succesvol: voor een reeks van benchmarks in de ingebedde software werd een versnelling bereikt van een factor 6,3 met een vermindering van het energieverbruik tot 66%.



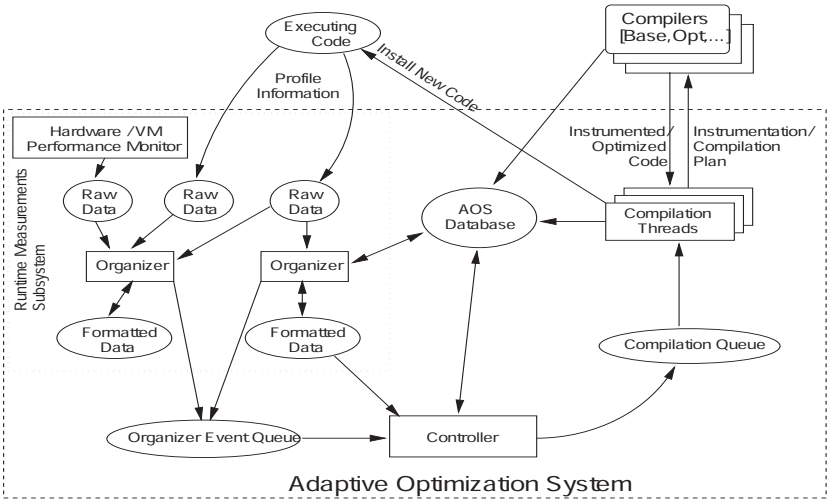
Figuur 4.9: Overzicht van de WARP-processor, één van de eerste platformen met ondersteuning voor dynamische hardware/software partitionering. Bron: Lysecky et al. (2006).

4.4.4 Dynamische partitionering in de hardwareversnelde Java virtuele machine

Faes' hardwareversnelde JVM is gebaseerd op de Jikes Research Virtual Machine (RVM) (Alpern et al., 2000). Deze JVM bevat het adaptieve optimalisatiesysteem van figuur 4.10 dat bedoeld is om een snelle, maar toch efficiënte JIT compilatie van de Java-programma's mogelijk te maken (Arnold et al., 2000). Het bestaat uit een profileringseenheid, het *runtime measurements subsystem*, dat tijdens de uitvoering van het programma allerlei statistieken bijhoudt over die uitvoering. Op basis van de data die dit systeem oplevert, wordt de JIT-compilatie gestuurd. Zoals beschreven in paragraaf 2.2 zorgt deze compiler voor een dynamische vertaling van de Java-bytecodes naar rechtstreeks uitvoerbare machinecode.

Dit gebeurt in verschillende stappen: op het eerste niveau gaat het over een rechttoe-rechtaan vertaling die ervoor zorgt dat er weinig tijd verloren gaat voor het compilatieproces. Het resultaat van deze *baseline*-compiler is relatief inefficiënte machinecode.

Vanaf het moment dat het adaptieve optimalisatiesysteem te weten komt dat een methode veelvuldig wordt uitgevoerd en/of dat de implementatie van die methode in de machinecode wel zeer inefficiënt is, wordt er beslist om de methode te hercompileren met extra compilatie-optimalisaties. Deze nieuwe compilatie neemt typisch meer tijd in beslag dan de *baseline* compiler, maar omdat ze tot



Figuur 4.10: Adaptief optimalisatiesysteem in de Jikes RVM. Bron: Arnold et al. (2000).

een meer efficiënte implementatie leidt en omdat de code vaak wordt uitgevoerd, kan die extra compilatietijd in veel gevallen teruggewonnen worden. Merk op dat dit op voorhand nooit helemaal zeker is. Hiervoor zijn twee redenen: ten eerste kan de snelheid van de nieuwe implementatie niet altijd correct worden voorspeld. Ten tweede is het feit dat code in het verleden (tot nu toe) vaak werd uitgevoerd geen absolute garantie dat dat in de toekomst nog steeds het geval zal zijn.

Dit concept van meertrapscompilatie kan uitgebreid worden met een extra optimalisatiestap die een specifieke hardwareversneller instantieert op de FPGA wanneer blijkt dat dit nuttig is. Hiervoor moet de JIT-compiler allereerst een inzicht hebben in de communicatie in het systeem. Daarvoor volstaat het een profileerder voor het meten van het communicatieprofiel tussen objecten en methodes, in te bouwen in de JVM. De werking van een dergelijke profileerder werd concreet uitgewerkt in hoofdstuk 3.

Daarnaast moet de JIT-compiler toegang hebben tot een bibliotheek van *bit streams* voor de FPGA die elk een hardwareblok voorstellen dat een specifieke methode kan vervangen. Deze bibliotheek dient eveneens een schatting van de uitvoeringstijd van die methode in hardware te bevatten. Op basis van die schatting kan de JIT-

compiler dan, net zoals dat bij de software-optimalisaties het geval is, dynamisch beslissen of het al dan niet nuttig is om het hardwareblok effectief te instantiëren.

Bruneel & Stroobandt (2008b) werkten een methode uit om zeer snel gespecialiseerde hardwareblokken te genereren. De tijd die nodig is om een dergelijk circuit te genereren is van dezelfde orde als de tijd die nodig is om de FPGA te configureren. In hun aanpak wordt een hardware-implementatie van een gegeven functionaliteit met willekeurige inputs, snel en efficiënt geoptimaliseerd voor het specifieke geval waarbij een aantal van die inputs gedurende langere tijd een vaste waarde hebben. Een concreet voorbeeld hiervan is een FIR-filter. De coëfficiënten van het filter liggen lange tijd vast en veranderen slechts af en toe, de waarden die gefilterd moeten worden, veranderen voortdurend. Op basis van een generiek filter, kan men met de aanpak van Bruneel & Stroobandt (2008b), een gespecialiseerd filter maken voor de coëfficiënten die op dat ogenblik nodig zijn. Deze aanpak is veelbelovend in de context van de hardwareversnelde JVM omdat ze het mogelijk zou maken om relatief snel en tijdens de uitvoering van het programma, hardwareblokken te genereren.

Een nog verder gevorderde versie van dynamische partitionering in de hardwareversnelde JVM kan ontstaan wanneer uit het niets volledige hardwareblokken gegenereerd kunnen worden door de JIT-compiler. In dat geval wordt hardwareversnelling pas echt een nieuwe optimalisatiestap in het meertraps-compilatieproces en dan is de hardwareversnelling ook werkelijk transparant voor de programmeur. Voorlopig blijft dit nog wel even toekomstmuziek, maar onder meer het werk van Beck & Carro (2005), toont aan dat het niet geheel onmogelijk is. Voor codefragmentjes van enkele Java-bytecodes, slaagden zij erin om geoptimaliseerde instructies te genereren op een gedeeltelijk herconfigureerbare Very Long Instruction Word (VLIW)-architectuur.

4.5 Besluit

In een gedistribueerd rekensysteem is het van uitermate groot belang dat een goede partitionering wordt gekozen. Die partitionering bepaalt immers in grote mate de communicatiekost in de uiteindelijke implementatie en die kost is vaak de belemmerende factor voor een optimaal ontwerp.

In dit hoofstuk heb ik twee technieken voorgesteld om tot een communicatiebewuste partitionering van de functionaliteit van het programma te komen.

De eerste techniek, statische partitionering, is bedoeld als ondersteuning voor een ontwerper die met de hand een programma partitioneert. Eens die ontwerper een aantal methodes heeft gemarkeerd als geschikt of ongeschikt voor afsplitsing naar een co-processor, zoekt het algoritme op basis van het communicatieprofiel van het programma, een gepaste partitionering met een zo laag mogelijke communicatiekost.

Dynamische partitionering, tot slot, is een uitbreiding van het statische algoritme voor gebruik in de hardwareversnelde JVM als bijkomende stap in het JIT-compilatieproces. Dit is een interessante en veelbelovende techniek, zeker voor toepassingen met in grote mate invoerafhankelijk communicatiegedrag. Enkel een dynamische aanpak kan hierop immers adequaat reageren.

Hoofdstuk 5

Communicatiebewuste plaatsing van objecten in het geheugen

Dit hoofdstuk gaat over een specifieke vorm van hardware/software co-ontwerp waarbij een klassieke computer wordt uitgebreid met een Field Programmable Gate Array (FPGA). Op de computer draait een Java Virtuele Machine (JVM) die het software-gedeelte van het programma uitvoert. Tegelijkertijd beheert de JVM ook de hardware en zal de JVM voor specifieke methodes de hardware aanroepen. Hier bestudeer ik hoe de JVM geheugenallocatie kan optimaliseren voor de verschillende geheugens in het systeem.

5.1 Invloed van geheugenbeheer op de totale communicatie in het systeem

5.1.1 Platform voor co-ontwerp in Java

Hardwareversnellers of andere toepassingsspecifieke co-processors worden vaak gebruikt om de prestaties van rekenintensieve programma's te verbeteren. In verschillende domeinen werden zo significante versnellingen genoteerd: multimedia (Eeckhaut et al., 2007; Lysecky et al., 2006; Vassiliadis et al., 2004), bio-informatica (Faes et al., 2006; Maddimsetty et al., 2006) en vele andere toepassingen die gebruik maken van kleine rekenintensieve kernen met veel intern parallellisme.

De eerste pogingen tot hardware/software co-ontwerp (Ernst et al., 1993; Gupta & De Micheli, 1993) ontstonden uit software die geschreven was in talen die dicht bij de hardware staan zoals C of C++ of varianten ervan. Stap voor stap werden delen van een programma vervangen door snellere, specifieke hardwareblokken tot men uiteindelijk een hardware/software-systeem verkreeg dat aan de snelheidsvereisten voldeed.

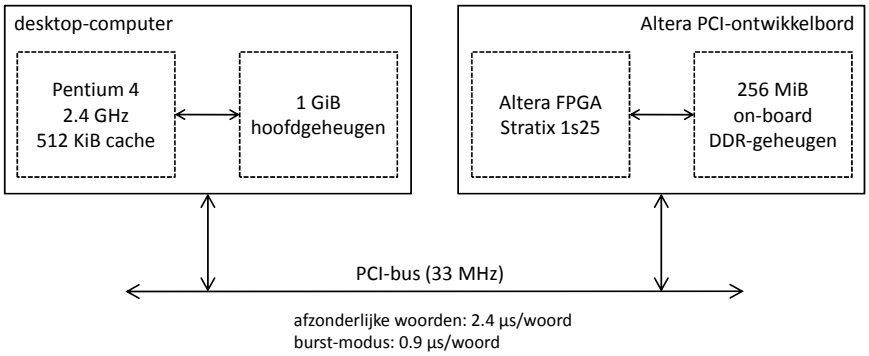
Meer recent vonden gelijkaardige technieken voor hardware/software co-ontwerp ook hun weg naar de Java-wereld voor de versnelling van Java-programma's (Helaihel & Olukotun, 1997). Daarbij werden hoofdzakelijk twee onderzoekslijnen gevolgd. Enerzijds is het mogelijk om de JVM zelf te versnellen. Anderzijds kan men er ook voor kiezen om slechts enkele specifieke methodes uit het programma te versnellen.

Het versnellen van de volledige JVM heeft als voordeel dat het volledig transparant is voor de programmeur. In de tweede aanpak moet de programmeur vaak zelf nog de communicatie en synchronisatie tussen de hardwareversneller en de software regelen.

Het grote voordeel van de tweede aanpak is dat de hardware enkel gebruikt wordt voor het versnellen van die specifieke methodes waar een significante versnelling verwacht kan worden, zoals bijvoorbeeld blijkt uit het werk van onder meer Hakkennes & Vassiliadis (2001) en Eeckhaut et al. (2007). Minder hardwarevriendelijke methodes blijven dan in software.

De hardwareversnelde JVM (Faes et al., 2004) die ik al vermeldde in hoofdstukken 2 en 4, combineert het beste van de twee werelden. Enerzijds levert deze JVM een volledig transparante oplossing voor de programmeur. Anderzijds wordt in deze aanpak geen kostbare hardware verspild voor algemene functies van de JVM die inefficiënt zijn in hardware, waaronder de geheugensanering door de *garbage collector*.

De hardwareversnelde JVM is zo aangepast dat hij de volledige controle heeft over de hardware en dus ook de synchronisatie perfect en transparant kan beheren. Het gaat zelfs nog een stap verder, de JVM van Faes et al. (2004) is in theorie in staat om tijdens de uitvoering functionaliteit die eerst in software werd uitgevoerd op de generieke processor, dynamisch te migreren naar een geschikte hardwareversneller van zodra een geschikte versneller beschikbaar is. Uitvoering in hardware kan dan beschouwd worden als een bijkomende stap in de Just-in-Time (JIT) compiler. De hardwareblokken die



Figuur 5.1: Architectuur van de hardwareversnelde JVM

in een dergelijke aanpak nodig zijn, kunnen ofwel ingeladen worden uit een vooraf beschikbare bibliotheek (Borg et al., 2006), ofwel *on-the-fly* gegenereerd worden aan de hand van methodes ontwikkeld door Beck & Carro (2005), Bruneel et al. (2007); Bruneel & Stroobandt (2008a) en Lysecky et al. (2006). Deze technieken zijn momenteel nog niet voldoende ontwikkeld om elke Java-methode volledig om te zetten naar hardwareversnellers, maar deze veelbelovende technieken zullen in de toekomst ongetwijfeld uitgebreid worden en ze zullen daardoor nog aan belang winnen.

5.1.2 Niet-uniforme geheugenstructuur

Om prestatieredenen hebben zowel de hoofdprocessor als de hardwareversneller elk hun eigen lokaal geheugen. Beide geheugens vormen samen de Java *heap* binnen het standaard *shared-memory* geheugenmodel van Java (Faes et al., 2007). Onze transparante hardwareversnelde JVM zorgt ervoor dat alle geheugentoegangen, naar alle objecten, correct worden afgehandeld, onafhankelijk van het specifieke geheugen waarin die objecten zich bevinden. De plaatsing van objecten in één van beide geheugens in het systeem heeft echter een cruciale impact op de algemene systeemprestaties. Uit figuur 5.1 blijkt immers dat de hardwareversneller en de hoofdprocessor met elkaar verbonden zijn over een relatief traag communicatiemedium waardoor geheugentoegangen van de hardwareversneller naar het hoofdgeheugen van de computer of vanuit de hoofdprocessor naar geheugen op het FPGA-bord erg duur, en dus te vermijden zijn.

De plaatsing van objecten in de gedistributeerde Java *heap* is dus

een belangrijke taak van de JVM. De JVM moet er immers voor zorgen dat alle objecten geplaatst worden in dat deel van het geheugen dat het dichtst staat bij de rekenkern (hoofdprocessor of hardwareversneller) die deze objecten het vaakst nodig heeft. Zo zullen gegevens die louter door een van beide rekenkernen gebruikt worden, steeds lokaal staan zodat de communicatie over het trage netwerk geminimaliseerd wordt. Het zoeken van de optimale geheugenlocatie is het centrale thema van dit hoofdstuk.

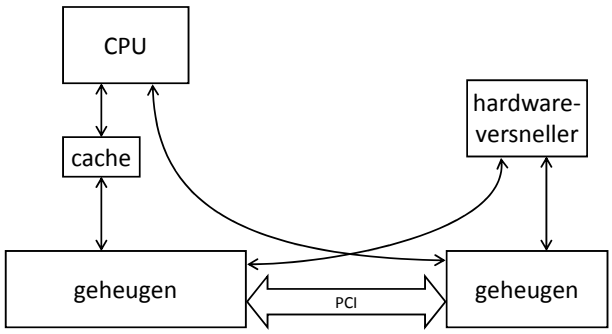
5.1.3 Oplossing voor het dataplaatsingsprobleem

Het is niet mogelijk om het dataplaatsingsprobleem op te lossen met statische analyse alleen omdat statische analyse enkel conservatief kan inschatten welke data louter gebruikt wordt door één specifieke methode. Het is zelfs sterker dan dat: gedeelde data wordt zelden symmetrisch gedeeld tussen verschillende componenten in het systeem. De relatieve verhouding tussen de datatoegangen vanuit een component en die vanuit een andere is vaak data-afhankelijk en daardoor zeer moeilijk in te schatten met statische analyse tijdens het compilatieproces. Tenslotte is er ook nog de situatie waarbij functionaliteit dynamisch wordt gemigreerd tussen de generieke hoofdprocessor en toepassingsspecifieke hardwareversnellers. Hierdoor wordt het gebruik van een dynamische aanpak onvermijdelijk.

In wat volgt beschrijf ik eerst in detail het geheugenmodel van de hardwareversnelde JVM (paragraaf 5.2). Daarna stel ik verschillende technieken voor communicatiebewust geheugenbeheer voor in paragraaf 5.3. Daarin wordt ernaar gestreefd om voor elk Java-object de optimale geheugenlocatie te bepalen.

5.2 Geheugenmodel van de hybride architectuur

Het hybride hardware/software-platform van de hardwareversnelde JVM gebruikt een gedeeld-geheugenmodel dat zowel de hoofdprocessor als de hardwareversneller toegang geeft tot alle objecten. De Java *heap* is verdeeld tussen het hoofdgeheugen en lokaal geheugen op de hardwareversneller. De *garbage collector* in deze JVM is uitgebreid om rekening te houden met objecten en referenties naar die objecten in beide geheugens, ook de referenties die de hardwareversneller bijhoudt (Faes et al., 2005). Of nieuwe objecten in het



Figuur 5.2: Geheugenmodel van de hardwareversnelde JVM. De CPU heeft een *cache* naar het hoofdgeheugen op de computer, maar toegangen van en naar de hardwareversneller op de FPGA gaan nooit via de *cache*, maar altijd rechtstreeks naar de FPGA over de PCI-bus. De FPGA bevat sowieso geen algemene *cache*-infrastructuur, zodat alle geheugentoegangen naar het lokale geheugen of naar het hoofdgeheugen van het systeem rechstreeks moeten verlopen.

hoofdgeheugen worden geplaatst of in lokaal geheugen bij de hardwareversneller zou dus moeten afhangen van de toegangspatronen naar die objecten. Dit is precies de focus van de algoritmes voor communicatiebewuste geheugenallocatie die in paragraaf 5.3 worden voorgesteld.

Hoewel in object-georiënteerde talen zoals Java de koppeling tussen data van objecten en de functionaliteit (methodes) beschreven in die objecten, conceptueel erg sterk is, worden in deze aanpak de beslissing over het plaatsen van de data van de objecten en het verdelen van de functionaliteit tussen processor en hardwareversneller volsterkt afzonderlijk behandeld. Zo is het mogelijk dat sommige methodes van een bepaalde klasse door de hardwareversneller worden uitgevoerd en andere methodes door de hoofdprocessor. Deze aanpak is vergelijkbaar met wat in een conventionele JVM gebeurt: ook daar zal men de functionaliteit van het programma, de bytecode, slechts op één plaats in het geheugen opslaan, terwijl de interne toestand van elk object, vanzelfsprekend, per object wordt bijgehouden.

Het hier beschouwde platform bevat een *cache* voor de geheugentoegangen naar het hoofdgeheugen. Deze cache zit tussen het hoofdgeheugen en de hoofdprocessor. Er wordt verondersteld dat

cache-coherentie wordt voorzien voor het geval de hardwareversneller schrijft naar objecten in het hoofdgeheugen, bijvoorbeeld met een coherente HyperTransport-bus. Tijdens de communicatiemetingen veronderstel ik dat niet-lokale geheugentoegangen, van de hardwareversneller naar objecten in het hoofdgeheugen of van de hoofdprocessor naar geheugen op de FPGA, niet gecachet worden. Dit is een realistische veronderstelling omdat de meeste FPGA's geen *cache*-geheugen hebben.

De CPU heeft typisch wel een *cache*-geheugen. In deze aanpak wordt verondersteld dat het *cache*-geheugen enkel de toegangen tussen de CPU en het hoofdgeheugen *cachet*. De toegangen van de CPU naar geheugen op de FPGA verlopen steeds rechtstreeks, zonder *caching*. Er wordt over de PCI-bus geen *cache*-coherentie verzekerd voor het geheugen op de FPGA. Dit wordt schematisch weergegeven in figuur 5.2.

Merk op dat de *cache* van de CPU een *write-through cache* moet zijn. De FPGA krijgt over de PCI-bus immers rechstreeks toegang tot het hoofdgeheugen, niet tot de *cache* van de CPU. Wijzigingen die de CPU wil aanbrengen in het hoofdgeheugen moeten dus niet enkel naar de *cache* geschreven worden, maar moeten meteen ook zichtbaar worden in het hoofdgeheugen zelf.

Ten opzichte van specifieke implementaties die wel *caching* voorzien van niet-lokale geheugentoegangen, zal er uiteraard meer communicatie zijn in de schatting die in dit werk gemaakt wordt zonder *caching*. Het verschil zal afhangen van de *hit rate* van deze *cache*. Niettemin zullen de technieken voor het minimaliseren van communicatie, die verder in dit hoofdstuk besproken worden, ook op platformen met *caching* hun nut kunnen bewijzen: de hoeveelheid communicatie zal nog steeds dalen en misschien kan zelfs de *cache* verkleind worden.

Dit hoofdstuk blijft beperkt tot het eenmalig (communicatiebewust) plaatsen van objecten in het hoofdgeheugen, dan wel in het lokale geheugen op de hardwareversneller. De mogelijkheid om objecten tijdens die uitvoering te verplaatsen van het ene geheugen naar het andere, wordt in dit werk buiten beschouwing gelaten. Het migreren van objecten zou een bijkomende prestatiewinst kunnen opleveren. Men moet er echter rekening mee houden dat het verplaatsen van objecten ook een extra kost met zich meebrengt. Het is een punt van verder onderzoek om na te gaan of de behaalde prestatiewinst opweegt tegen deze bijkomende kost. Ook een tussenvorm is moge-

lijk waarbij de migratie enkel wordt toegepast op de langst levende objecten. De meeste objecten in Java hebben immers een vrij korte levensduur, wat het terugverdienen van de migratiekost uiteraard bemoeilijkt.

5.3 Strategieën voor dataplaatsing

Twee verschillende strategieën kunnen onderscheiden worden: een aanpak gebaseerd op een vooraf opgenomen profiel van de datatoegangen in het programma en een dynamische, zelflerende strategie. In de eerste aanpak worden alle lees- en schrijfoperaties naar het geheugen geteld tijdens een voorafgaande, afzonderlijke profileringsfase. In elke volgende uitvoering van hetzelfde programma wordt de data onmiddellijk geplaatst in het optimale geheugen volgens de opgemeten statistieken. De zelflerende aanpak probeert dynamisch het gebruikspatroon van elk object te schatten op basis van opgemeten patronen voor eerder aangemaakte objecten tijdens de huidige uitvoering van het programma. Deze twee technieken zal ik vergelijken met een referentie-algoritme dat volstrekt geen rekening houdt met de communicatiekost en met een statische techniek voor lokale dataplaatsing waarbij gepoogd wordt de communicatiekost te verminderen zonder effectief toegangspatronen naar de data op te meten.

Deze strategieën voor het plaatsen van objecten in het geheugen, zorgen voor een drastische vermindering in het aantal niet-lokale geheugentoegangen. Het dynamische, zelflerende algoritme levert voor programma's uit de SPECjvm (Standard Performance Evaluation Corporation, 1998, 2008) en DaCapo-benchmarks (Blackburn et al., 2006) een vermindering op van gemiddeld 43%, bij statische lokale dataplaatsing is dat 12%. Beide methodes presteren goed voor verschillende benchmarks. Wanneer deze technieken gecombineerd worden en per benchmark de beste van de twee wordt gebruikt, levert dat een gemiddelde winst op van 53% (paragraaf 5.4.2).

De algemene regel zou moeten zijn dat objecten steeds in het geheugen worden geplaatst dat het dichtst staat —wat betreft communicatiekost— bij de component, de hoofdprocessor of de hardwareversneller, die het object het vaakst nodig heeft. Door deze regel te volgen, verbetert de datalokaliteit van het programma en zal een grote fractie van alle geheugentoegangen bestaan uit lokale

toegangen. Daardoor vermindert ook de totale communicatie en dus ook de geassocieerde communicatiekost.

Deze optimale plaatsing van alle objecten kan onmogelijk bepaald worden tijdens de uitvoering van het programma. Daarvoor zijn twee belangrijke redenen. Ten eerste kan men niet in de toekomst kijken en is het daardoor onmogelijk om op voorhand te weten hoe een nieuw aangemaakt object gebruikt zal worden. De beslissing over het plaatsen van een object zal steeds gebaseerd zijn op andere informatie zoals vooraf opgemeten profielen of gebruikspatronen van eerder gealloceerde objecten. Een tweede reden is dat er veel te veel objecten zijn om alle statistieken per object bij te houden. Daardoor moeten plaatsingsbeslissingen op een geclusterde manier genomen worden. In dit werk worden de statistieken bijgehouden per creatieplaats: dat is de lijn in de broncode van het programma waar een object wordt aangemaakt.

Hierbij wordt verondersteld dat objecten die worden aangemaakt op dezelfde creatieplaats gelijkaardige gebruikspatronen hebben. Daarom kunnen deze objecten in hetzelfde geheugen worden geplaatst zonder al te veel prestatie te moeten inboeten ten opzichte van de situatie waarbij men voor elk object afzonderlijk de optimale locatie bepaalt. Bovendien kunnen de opgemeten toegangspatronen van eerder aangemaakte objecten in eenzelfde creatieplaats gebruikt worden om de optimale plaats te bepalen van later aangemaakte objecten. Deze toegangspatronen kunnen opgemeten worden op twee manieren: ofwel door profilering van het programma tijdens de uitvoering ofwel tijdens een afzonderlijke, voorafgaande profileringsfase.

In de meeste gevallen worden op een creatieplaats objecten aangemaakt van hetzelfde datatype en met erg gelijkaardige toegangspatronen. Sommige specifieke software-ontwerpspatronen wijken hiervan af. Een belangrijk voorbeeld hiervan zijn de zogenaamde *class factories* waarbij één enkele creatieplaats objecten aanmaakt van verschillende types die bovendien in een verschillende context en op een verschillende manier gebruikt zullen worden. Niettemin blijkt uit de resultaten in paragraaf 5.4 dat het verband tussen de creatieplaats en de toegangspatronen naar de objecten erg sterk is voor die objecten die de meeste niet-lokale geheugentoegangen veroorzaken.

In dit hoofdstuk vergelijk ik verscheidene algoritmes voor communicatiebewust geheugenbeheer: een referentie-algoritme, profielgebaseerde optimale dataplaatsing, lokale allocatie en een zelflerend

algoritme voor dataplaatsing. Deze algoritmes verschillen in complexiteit van de implementatie, in het feit of de allocatie adaptief is of vast, en in het feit of de dataplaatsing gebaseerd wordt op dynamisch opgemeten informatie dan wel op informatie uit een afzonderlijke, voorafgaande profilering.

5.3.1 Referentie-algoritme

Een eerste benadering is het referentie-algoritme waarbij alle objecten worden geplaatst in het hoofdgeheugen omdat de meeste methodes op de hoofdprocessor uitgevoerd worden. Alle geheugentoegangen die uitgaan van de hardwareversneller zullen dus niet-lokale geheugentoegangen zijn. Daarom geeft dit referentie-algoritme behoorlijk grote communicatiekosten, hoewel uit paragraaf 5.4 blijkt dat dit algoritme voor sommige benchmarks toch nog behoorlijk goed presteert, vergelijkbaar met communicatiebewuste strategieën. De complexiteit om dit algoritme te implementeren is uiteraard zeer laag omdat in wezen geen enkele beslissing genomen moet worden. De *runtime overhead* voor deze strategie is nul.

5.3.2 Optimale dataplaatsing

Op basis van de gemeenschappelijke toegangspatronen voor alle objecten die op dezelfde plaats in de broncode aangemaakt werden, gemeten over de volledige duur van een programma, kan de optimale geheugenlocatie voor elk van die creatieplaatsen bepaald worden. Deze strategie is uiteraard niet helemaal optimaal omdat de objecten niet elk afzonderlijk beschouwd worden. Maar omdat het praktisch nauwelijks haalbaar is, gezien de grote aantallen objecten, om dit per object te bepalen, lijkt het redelijk om het optimum slechts per creatieplaats te bepalen. Dit vormt dan ook de ideale situatie waarmee ik de andere strategieën in dit hoofdstuk zal vergelijken.

Deze strategie vergt een afzonderlijke profileringsfase voor het bepalen van de globale toegangspatronen per creatieplaats (Bertels et al., 2008). In dit werk heb ik de volledige toegangspatronen opgemeten voor een specifieke uitvoering van verschillende Java-programma's. Alternatief zou men ook toegangspatronen over verschillende uitvoeringen kunnen samenstellen om een meer algemeen beeld te krijgen. Deze strategie voor optimale dataplaatsing vergt geen *runtime overhead* omdat de beslissing over het plaatsen van elk object reeds vastligt voordat de uitvoering start.

```
function lokaleDataplaatsing(CreatiePlaats cp, Rekenknoop rekenknoop):
    if rekenknoop == HOOFDPROCESSOR:
        return HOOFDPROCESSOR
    else:
        return FPGA
end

function zelfLerend(CreatiePlaats cp, Rekenknoop rekenknoop):
    if cp.tellerHoofdProcessor >= cp.tellerFpga :
        return HOOFDPROCESSOR
    else:
        return FPGA
end

proc registreerGeheugenToegang(Rekenknoop rekenknoop, Object object):
    CreatiePlaats cp := getCreatiePlaats(object)
    if rekenknoop == HOOFDPROCESSOR:
        cp.tellerHoofdProcessor ++
    else:
        cp.tellerHoofdFpga ++
end
```

Figuur 5.3: Pseudocode voor dataplaatsingsalgoritmes

5.3.3 Lokale dataplaatsing

Veel objecten hebben slechts een korte levensduur. Ze worden in veel gevallen bijna uitsluitend gebruikt door de methode die deze objecten heeft aangemaakt. Deze vaststelling vormt het uitgangspunt voor de strategie van de lokale dataplaatsing waarbij elk object wordt aangemaakt in het geheugen dat het dichtst staat bij de component die het object heeft aangemaakt. Dit wordt weergegeven in de pseudocode in figuur 5.3. De informatie die nodig is om deze strategie te implementeren is rechtstreeks beschikbaar terwijl het programma loopt. De implementatie van lokale dataplaatsing is dan ook eenvoudig en ze vergt geen extra rekentijd ten gevolge van instrumentatie tijdens de uitvoering. Ook vooraf is er voor deze strategie geen afzonderlijke profileringsfase nodig.

Dit concept van lokale dataplaatsing is verwant aan de aanpak van Lattanzi et al. (2005) die ik in paragraaf 5.5 uitgebreid bespreek.

5.3.4 Zelflerend algoritme

Het zelflerende algoritme laat de Virtuele Machine (VM) tijdens de uitvoering beslissen waar elk object geplaatst wordt op basis van de toegangspatronen van eerder aangemaakte objecten. Een zelflerend algoritme is bijzonder nuttig in een dynamische omgeving zoals de hardwareversnelde JVM die tijdens de uitvoering kan beslissen om functionaliteit op de hoofdprocessor uit te voeren, dan wel op een specifieke hardwareversneller.

De JVM zal voortdurend alle geheugentoegangen tellen vanuit zowel de hoofdprocessor als de hardwareversneller naar alle objecten in beide geheugens. Dit kan bijvoorbeeld gebeuren door (bemonsterende) instrumentatie of door rechtstreekse profilering in de hardware. Voor elke creatieplaats worden twee tellers bijgehouden, een voor de processor en een voor de hardwareversneller, die elk bijhouden hoeveel toegangen er waren naar de objecten die op die creatieplaats aangemaakt werden. Op elk ogenblik vertelt het eenvoudig vergelijken van beide tellers welke component deze objecten het vaakst gebruikt heeft tot op dat ogenblik in de uitvoering. Nieuwe objecten die op een creatieplaats worden aangemaakt, worden dan geplaatst in het geheugen dat het dichtst staat bij de component met de meeste toegangen. Dit wordt weergegeven in de pseudocode in figuur 5.3. Op het einde van het programma bereiken alle tellers dezelfde waarde die verkregen wordt bij de profileringsfase voor optimale dataplaatsing (paragraaf 5.3.2). De plaatsingsbeslissing voor nieuwe objecten in het zelflerende algoritme zal dus convergeren naar dezelfde beslissing van de optimale dataplaatsing. Deze convergentie gebeurt doorgaans vrij snel, zoals blijkt uit de paragraaf 5.4, waardoor deze strategie de hoeveelheid niet-lokale datatoegangen sterk kan reduceren.

Voor de praktische implementatie van de tellers die nodig zijn voor dit algoritme, zijn er verschillende mogelijkheden. Dit kan gebeuren via bytecode-instrumentatie, zoals de profileerder die besproken werd in hoofdstuk 3. Het is echter efficiënter om de tellers aan te passen in de JVM zelf. De JVM beschikt immers over alle informatie waardoor een extra teller bijhouden relatief snel kan gaan. Elke mogelijke tijdswinst is van belang voor de praktische implementatie van dit zelflerende algoritme, een argument dat veel minder belangrijk was voor het opbouwen van het communicatieprofiel in hoofdstuk 3.

Tabel 5.1: Cruciale eigenschappen van de verschillende strategieën

strategie	referentie	optimaal	lokaal	zelflerend
beslissing	vooraf	vooraf	uitvoering	uitvoering
toegangspatronen	neen	ja	neen	ja
adaptief	neen	neen	ja	ja
runtime overhead	neen	neen	neen	ja

5.4 Resultaten

Ik heb de verschillende technieken voor dataplaatsing geëvalueerd met behulp van de SPECjvm (Standard Performance Evaluation Corporation, 1998, 2008) en DaCapo benchmark suites (Blackburn et al., 2006). Hierbij werd van de programma's die zowel in SPECjvm98 als in SPECjvm2008 opgenomen zijn, enkel de recentste versie gebruikt. Benchmarks *sunflow* en *crypto.signverify* werden ook buiten beschouwing gelaten omdat ze minder dan 100 objecten aanmaken.

In een eerste profileringsfase heb ik voor elke benchmark de tien heetste methodes bepaald, d.w.z. de methodes die verantwoordelijk zijn voor de grootste fractie van de uitvoeringstijd. Voor al mijn simulaties heb ik verondersteld dat er voor elk van die tien heetste methodes een geschikte hardwareversneller bestaat. Het was immers in het tijdsbestek van dit doctoraatsonderzoek onmogelijk om voor elk van de benchmarks en alle hete code hardwareversnellers te implemeneteren. Voor de tien hardwareversnelde methodes geeft dat in principe 2^{10} mogelijke partitioneringen van de functionaliteit. Maar die werden niet alle 1024 geëvalueerd. Ik heb me beperkt tot experimenten waarbij voor elke benchmark 10 verschillende partitioneringen aan bod kwamen met daarbij telkens de n heetste methodes in hardware, met n van 1 tot 10. Dat geeft 10 verschillende partitioneringen waarvoor steeds elke strategie uitgetest werd. Omdat de resultaten voor die 10 partitioneringen gelijklopend waren, geef ik in dit hoofdstuk enkel de grafieken voor het geval waarbij de 10 hardwareversnelde methodes ook allemaal efffectief op de FPGA werden uitgevoerd.

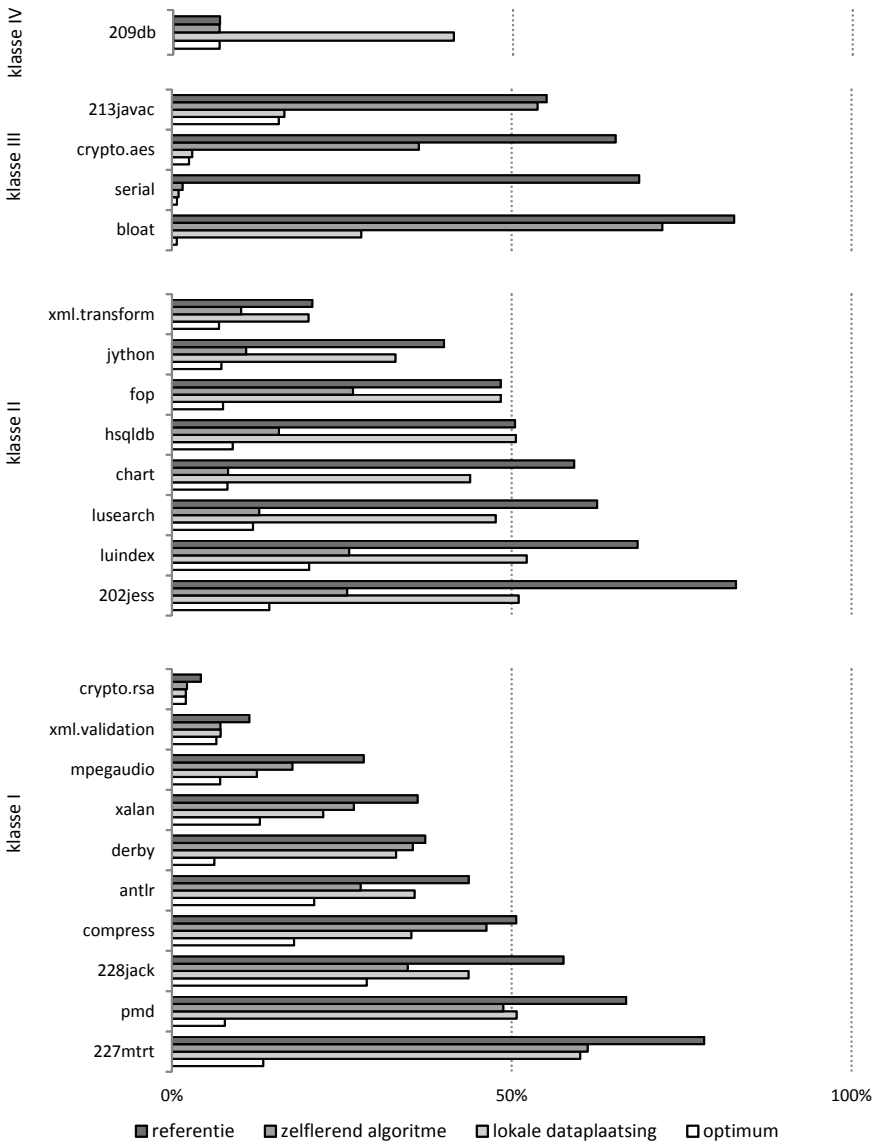
5.4.1 Het aantal niet-lokale geheugentoegangen

Mijn eerste experiment is een algemene vergelijking van de vier verschillende strategieën voor geheugenallocatie over alle benchmarks. Hiervoor werden alle benchmarks geprofileerd waarbij alle geheugentoegangen geregistreerd werden. Op basis van die metingen van de geheugentoegangen berekende ik het aantal lokale en niet-lokale geheugentoegangen tijdens de volledige uitvoering van de benchmarks in de veronderstelling dat elk van de 10 heetste methodes op de hardwareversneller werd uitgevoerd.

In de referentie-aanpak waarbij alle objecten in het hoofdgeheugen van de processor worden opgeslagen, zijn alle geheugentoegangen vanuit de hardwareversnelde methodes dus niet-lokale geheugentoegangen. Als sommige van die objecten in het geheugen van de versneller geplaatst worden, veranderen toegangen naar die objecten in lokale toegangen voor de hardwareversneller, terwijl de toegangen vanuit de hoofdprocessor nu niet-lokaal (en dus duurder) worden. Voor de optimale dataplaatsing is het dus duidelijk dat de hoeveelheid niet-lokale geheugentoegangen steeds kleiner zal zijn dan bij de referentie-aanpak. De andere twee strategieën leiden in de meeste gevallen ook tot een drastische vermindering van het aantal niet-lokale geheugentoegangen, al kan dat voor die strategieën niet ontegensprekelijk gegarandeerd worden.

Figuur 5.4 toont het relatieve aantal niet-lokale geheugentoegangen voor de verschillende strategieën voor dataplaatsing voor alle programma's van de SPECjvm en DaCapo benchmark suites. In dit experiment werden de tien heetste methodes van elk programma op de hardwareversneller geplaatst. Uit deze vergelijking blijkt dat de benchmarks op basis van deze resultaten ingedeeld kunnen worden in vier categorieën.

Programma's in klasse I, II en III vertonen het gewenste gedrag: optimale dataplaatsing levert uiteraard de beste prestaties op, de referentie-aanpak scoort het slechtst en lokale dataplaatsing en de zelflerende strategie liggen daartussenin. Klasse I bevat de benchmarks waarvoor het zelflerende algoritme en lokale dataplaatsing vergelijkbare resultaten opleveren. Bij benchmarks in klasses II en III verschilt het aantal overblijvende niet-lokale datatoegangen voor deze twee strategieën drastisch. Voor klasse II zijn er minstens 50% meer niet-lokale toegangen voor lokale dataplaatsing dan voor de zelflerende strategie. Voor deze benchmarks is het zelflerende algoritme dus de beste keuze. In klasse III is het net omgekeerd: lokale



Figuur 5.4: Vergelijking van het relatieve aantal niet-lokale geheugentoegangen voor de verschillende strategieën voor dataplaatsing, getest op alle benchmarks in de veronderstelling dat elk van de 10 heetste methodes op de hardwareversneller werd uitgevoerd.

dataplaatsing presteert significant beter, het zelflerende algoritme levert immers steeds meer dan 50% meer dure geheugentoegangen.

De uitschieter in de lijst is *209db* die tot klasse IV behoort: in dit programma worden heel veel objecten aangemaakt door de hete methodes op de versneller terwijl die objecten later voornamelijk door de hoofdprocessor gebruikt worden. Concreet gaat het over methode `Database.set_index()` die arrays van het type `Entry[]` aanmaakt voor het indexeren van een database. Deze objecten worden bijzonder vaak gebruikt door methodes die op de hoofdprocessor worden uitgevoerd en ze worden na creatie niet meer gebruikt door de hardwareversneller. Daarom presteert het algoritme voor lokale dataplaatsing zeer slecht voor *209db*. De referentie-aanpak waarbij alle objecten in het hoofdgeheugen worden geplaatst, ook deze `Entry[]`-arrays is hier dus wel zeer efficiënt. Het zelflerende algoritme en de optimale plaatsing zijn nog net iets beter dan het referentie-algoritme omdat zij ook gebruik kunnen maken van het feit dat enkele objecten enkel intern binnen de hardwareversneller worden gebruikt. Dit is onder meer het geval voor de `StringBuffer` in `ValidityCheckOutputStream.strip(int, InputStream)`.

5.4.2 Beste presterende techniek per benchmark

Uit figuur 5.4 blijkt dat geen enkele techniek voor het communicatiebewust plaatsen van objecten het best presteert voor alle benchmarks. Voor de benchmarks in klasse II is het zelflerende algoritme duidelijk de beste keuze, voor klasse III presteert lokale dataplaatsing beter. Hoewel de benchmarks in klasse I een vergelijkbaar resultaat opleveren voor beide technieken, kan ook voor deze benchmarks een best presterende techniek bepaald worden.

Tijdens een éénmalige, initiële profilering kan bepaald worden tot welke klasse een programma behoort en welke techniek voor het plaatsen van de objecten de beste prestatie oplevert. Vervolgens kan men die beste techniek toepassen bij de eigenlijke uitvoering van het programma.

Tabel 5.2 toont voor elk van de geteste benchmarkprogramma's welke techniek de beste prestaties oplevert. In de tabel staat in de eerste kolom de fractie van niet-lokale geheugentoegangen voor de referentie-aanpak en in de tweede kolom de fractie in het geval van de best presterende techniek. De laatste kolom in de tabel toont de

Tabel 5.2: Best presterende techniek voor elke benchmark

benchmark	referentie	beste aanpak		winst
209db	7%	7%	referentie	0%
bloat	83%	28%	lokaal	66%
serial	69%	1%	lokaal	99%
crypto.aes	65%	3%	lokaal	95%
213javac	55%	17%	lokaal	70%
202jess	83%	26%	zelflerend	69%
luindex	69%	26%	zelflerend	62%
lusearch	63%	13%	zelflerend	79%
chart	59%	8%	zelflerend	86%
hsqldb	50%	16%	zelflerend	69%
fop	48%	27%	zelflerend	45%
jython	40%	11%	zelflerend	73%
xml.transform	21%	10%	zelflerend	51%
227mtrt	78%	60%	lokaal	23%
pmd	67%	49%	zelflerend	27%
228jack	58%	35%	zelflerend	40%
compress	51%	35%	lokaal	30%
antlr	44%	28%	zelflerend	36%
derby	37%	33%	lokaal	12%
xalan	36%	22%	lokaal	38%
mpegaudio	28%	13%	lokaal	56%
xml.validation	11%	7%	zelflerend	37%
crypto.rsa	4%	2%	lokaal	52%

procentuele vermindering van het aantal niet-lokale geheugentoegangen.

De keuze voor een communicatiebewuste strategie voor het plaatsen van objecten in het geheugen, die aangepast is aan het programma, zorgt voor een significante daling van het aantal niet-lokale, en dus dure, geheugentoegangen. Gemiddeld resulteert deze aanpak in een verbetering met 53%, zoals blijkt uit tabel 5.2.

5.4.3 Hoe snel leert het zelflerend algoritme?

In een tweede experiment heb ik de convergentie van het zelflerende algoritme onderzocht. Dit algoritme plaatst objecten daar waar ze

het vaakst nodig waren in het verleden. In het ideale geval convergeert het algoritme: voor elke creatieplaats bereikt het de optimale geheugenlocatie voor de objecten aangemaakt op die creatieplaats. In deze paragraaf wordt beschreven hoe het algoritme zich in de praktijk gedraagt. Cruciaal daarbij is het begrip 'convergentie' dat hieronder wordt gedefinieerd. Na de definitie, beschrijf ik hoe die convergentie in de praktijk opgemeten werd. Tot slot worden de resultaten besproken.

Zelflerend algoritme in de praktijk

Het zelflerende algoritme houdt voor elke creatieplaats, dus voor elke lijn in de broncode waarop nieuwe objecten worden aangemaakt, twee tellers bij. De ene teller telt het aantal geheugentoegangen vanuit de hoofdprocessor naar de objecten aangemaakt op deze creatieplaats. De andere teller doet hetzelfde voor de hardwareversneller. Telkens als er een nieuw object wordt aangemaakt, zal de JVM beide tellers met elkaar vergelijken en het object aanmaken in het geheugen van de rekenknoop (processor of versneller) die vergelijkbare objecten het vaakst heeft gebruikt.

Afhankelijk van het gedrag van het programma zijn er verschillende scenario's.

Het ideale scenario is dat waarbij alle objecten van een creatieplaats slechts door één van beide rekenknopen worden gebruikt. Het eerste object wordt altijd geplaatst in het hoofdgeheugen van de processor. Als dit object en alle volgende objecten enkel en alleen door de hardwareversneller wordt gebruikt, dan zal de teller voor de hoofdprocessor op 0 blijven staan, met als gevolg dat alle volgende objecten van dit type meteen in het geheugen van de hardwareversneller zullen worden geplaatst.

Een heel ander scenario is dat waarbij objecten van een creatieplaats door beie rekenknopen ongeveer even vaak worden gebruikt. Het eerste object wordt opnieuw in het hoofdgeheugen geplaatst. Wanneer bij het aanmaken van het tweede object van dezelfde creatieplaats, uit de tellers blijkt dat het eerste object net iets vaker gebruikt werd door de hardwareversneller dan door de processor, dan wordt het tweede object in het geheugen van de FPGA geplaatst. Het gedrag van het programma kan ervoor zorgen dat beide rekenknopen deze objecten gebruiken en dat de twee tellers afwisselend de overhand nemen, en als het ware haasje-over spelen. Daardoor zul-

len voor die creatieplaats een deel van de objecten in het hoofdgeheugen terechtkomen en een deel in het geheugen op de FPGA. Er komt geen duidelijke winnaar uit te bus. Op zich hoeft dat geen probleem te zijn want in deze situatie blijven er sowieso nog relatief veel niet-lokale datatoegangen over, ook bij lokale dataplaatsing of in de referentie-aanpak.

Definitie van convergentie

Voor elke creatieplaats kan men zich de vraag stellen of het algoritme ooit tot rust komt. Of er met andere woorden een punt in de tijd is vanaf wanneer alle objecten van die creatieplaats in hetzelfde geheugen worden geplaatst.

Daaruit volgt deze definitie: het zelflerende algoritme *convergeert* voor een gegeven creatieplaats, als en slechts als, er een punt in de tijd bestaat waarna één van beide tellers, tot op het einde van het programma, hoger blijft dan de andere teller.

Voor creatieplaatsen waarvoor het algoritme convergeert, definieer ik de *convergentietijd*, als het totaal aantal objecten dat werd aangemaakt op die creatieplaats vóór convergentie. Een creatieplaats met convergentietijd 0, is dus een creatieplaats waarvoor alle objecten in hetzelfde geheugen werden geplaatst. Merk op dat bij de definitie van convergentietijd alle objecten worden geteld die aangemaakt worden voor convergentie optreedt: zowel de objecten die in het 'optimale geheugen werden geplaatst als objecten die 'fout werden geplaatst.

Experiment voor het opmeten van convergentie

Voor elk van de benchmarkprogramma's werd per creatieplaats, nagegaan of ze convergeert en wat de convergentietijd is. Om inzicht te krijgen in de snelheid van convergentie over alle benchmarks en creatieplaatsen heen, heb ik de creatieplaatsen gesorteerd volgens hun convergentietijd. Na het sorteren werd al duidelijk dat het algoritme voor de meeste creatieplaatsen vrij snel tot convergentie komt: er worden vaak maar enkele objecten aangemaakt voor convergentie.

Na sorteren kan een histogram worden getekend van deze creatieplaatsen volgens hun convergentietijd. Dit geeft echter niet het hele plaatje. Er is immers een wereld van verschil tussen een creatieplaats die zeer snel convergeert, maar weinig belang heeft omdat

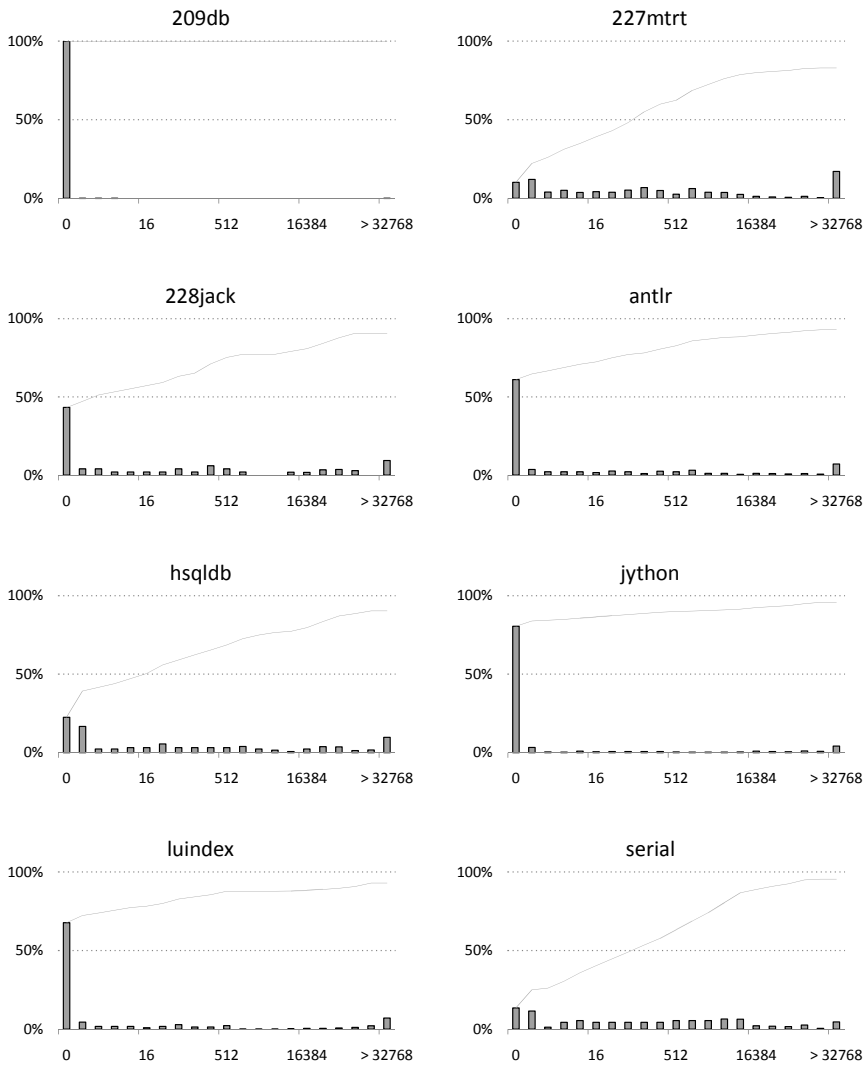
er weinig objecten worden aangemaakt en een traag convergerende creatieplaats waar wel veel objecten bijhoren.

Figuur 5.5 toont die convergentie voor enkele representatieve benchmarks. In dit histogram staan staan alle creatieplaatsen gewogen volgens het aantal objecten die op deze creatieplaatsen aangemaakt worden. De horizontale as toont de convergentietijd. De vertikale as geeft het aantal objecten als fractie van het totaal aantal objecten aangemaakt in het volledige programma. Dit geeft aan hoe belangrijk de creatieplaatsen zijn. Tot slot geeft figuur 5.5 ook een cumulatieve curve voor de relatieve fractie van objecten. Merk op dat de cumulatieve curve niet altijd op 100% eindigt. Er blijft bij de meeste benchmarkprogramma's nog een fractie van objecten over die aangemaakt worden op een creatieplaats die de convergentie niet bereikt of zeer traag (> 32768).

Bespreking van de resultaten

Uit de vergelijking van de verschillende strategieën voor alle benchmarks, bleek al dat voor benchmark *209db* het hoofdgeheugen de optimale geheugenlocatie is voor het grootste deel van de objecten, ook de objecten die door de hardwareversneller aangemaakt worden. Omdat het zelflerende algoritme standaard beslist om alle objecten in het hoofdgeheugen te plaatsen, worden alle objecten vanaf de start van het programma al correct geplaatst. In figuur 5.5 komt dit tot uiting als een balk van 99,96% op 0: voor bijna alle creatieplaatsen wordt geen enkel object fout geplaatst. Om die reden was *209db* dan ook een buitenbeentje in figuur 5.4.

Het gedrag van benchmark *227mtrt* is ingewikkelder. Voor 10% van alle objecten is het optimale geheugen het hoofdgeheugen, net zoals voor benchmark *209db* geeft dit een balkje van 10% op 0 in het histogram. Voor sommige creatieplaatsen wordt het eerste object in het hoofdgeheugen geplaatst, maar wordt dat vervolgens vaker door de hardwareversneller gebruikt dan door de hoofdprocessor. Daarom zal het zelflerende algoritme alle volgende objecten in het geheugen van de hardwareversneller plaatsen, wat voor die objecten inderdaad de optimale keuze is. Dus enkel het eerste object aangemaakt op elk van deze creatieplaatsen werd fout geplaatst. Voor benchmark *227mtrt* staan deze creatieplaatsen in voor 12% van alle objecten zoals blijkt uit de balk bij 1 in figuur 5.5. Voor andere creatieplaatsen werden meerdere objecten aangemaakt alvorens er duide-



Figuur 5.5: Op de horizontale as in dit histogram staan de creatieplaatsen geclusterd volgens hun convergentietijd, gedefinieerd als het aantal objecten dat door deze creatieplaatsen werden aangemaakt vóór convergentie van het algoritme. De balken op de verticale as geven het totaal aantal aangemaakte objecten van de creatieplaatsen weer, als fractie van het totaal aantal objecten in het hele programma, en wijzen zo op het relatieve belang van de creatieplaatsen. Elk histogram toont ook een cumulatieve curve voor de relatieve fractie van objecten. Voor de meeste programma's leert het zelflerende algoritme erg snel voor de meeste creatieplaatsen.

lijk meer toegangen komen vanuit de ene component, de processor of de hardwareversneller dan vanuit de andere, zodat een correcte en finale beslissing kan worden genomen. Het laatste balkje in het histogram toont dat de creatieplaatsen waarvoor het algoritme niet (of uitzonderlijk traag) convergeert naar de optimale locatie, instaan voor 17% van alle objecten in *227mtrt*.

Tot slot blijkt uit figuur 5.5 dat het zelflerende algoritme relatief snel convergeert. De fractie van objecten die aangemaakt worden op een creatieplaats die niet convergeert is nooit groter dan 20%. Dat maximum wordt bereikt in benchmark *201compress*, niet getoond op figuur 5.5.

5.4.4 Zelflerend algoritme in de praktijk

De vorige paragrafen hebben aangetoond dat het zelflerende algoritme heel goed presteert voor de meeste benchmarks. Door het dynamische en adaptieve karakter kan dit algoritme ook gebruikt worden in situaties waar het geheugengedrag van het programma niet gekend is tijdens het compileren. Dit is het geval voor toepassingen die een data-afhankelijk gedrag hebben of in toepassingen waar de beslissing om de functionaliteit op de hoofdprocessor dan wel op een specifieke hardwareversneller uit te voeren pas tijdens de uitvoering van het programma wordt genomen. Dat laatste is ook het geval wanneer hardwareversnellers *on-the-fly* worden ontwikkeld, als uitbreiding van het conventionele JIT-compilatieproces.

Het enige nadeel van de aanpak is het feit dat deze techniek vrij duur is. Alle geheugentoegangen moeten worden geprofileerd en voor elke toegang moeten de tellers van de desbetreffende objecten worden aangepast. Dit verhoogt de uitvoeringstijd met een factor tien of meer, behalve op platformen waar eventueel *hardware performance counters* zouden kunnen worden gebruikt. Maar dergelijke tellers die rechtstreeks in de hardware zijn ingebouwd, zijn uiteraard niet altijd aanwezig. Bovendien komen ook deze tellers met een zekere kost in de extra hardware-oppervlakte die ze vereisen.

Het is duidelijk dat dit onaanvaardbaar is omdat de te verwachten prestatieverbetering door verbeterde dataplaatsing voor de meeste benchmarks immers minder groot is.

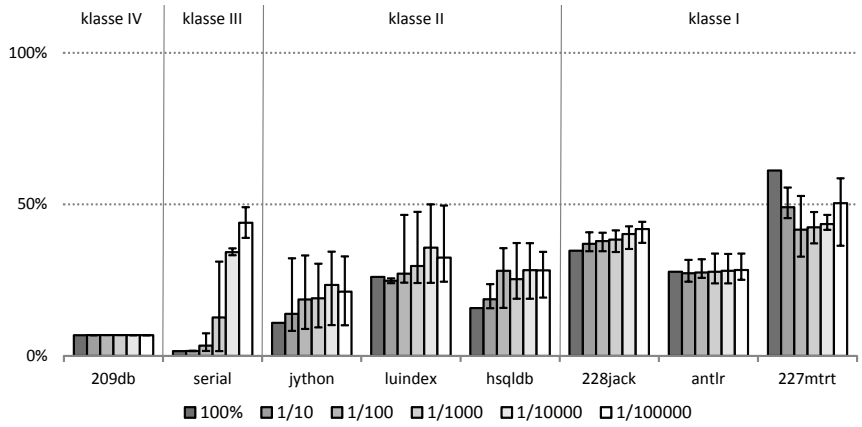
De kost voor het profileren van alle geheugentoegangen kan sterk worden verminderd door over te schakelen op een bemonsterende aanpak waarbij slechts een bepaald deel van de geheugentoe-

gangen wordt geïnstrumenteerd. In hoofdstuk 3 werd de techniek van *reservoir sampling* gebruikt voor het opmeten van het volledige communicatieprofiel van een Java-programma. Het zelflerende algoritme voor communicatiebewust geheugenbeheer, beschreven in dit hoofdstuk, bouwt incrementeel een communicatieprofiel op dat bovendien ook alle datastructuren bevat waarlangs die communicatie verloopt. Hiervoor is de techniek van *reservoir sampling* niet geschikt. Het zelflerende algoritme moet immers al van bij de start beslissingen kunnen nemen en kan niet wachten tot een reservoir van n monsters is volgelopen. Daarom gebruik ik voor de experimenten in dit hoofdstuk een klassieke $1/n$ bemonstering, waarbij elke lees- of schrijfoperatie in het geheugen met een probabiliteit van $1/n$ in rekening wordt gebracht voor het communicatieprofiel.

Omdat de toegangspatronen met de grootste invloed op de prestatie van het programma, net die patronen zijn met het grootste aandeel in de totale communicatie, en omdat die patronen slechts langzaam evolueren in de tijd, zou men verwachten dat het nemen van een zeer kleine steekproef van die geheugentoegangen, voor de meeste benchmarks, weinig invloed heeft op het gedrag van het zelflerende dataplaatsingsalgoritme. Deze intuïtieve veronderstelling wordt in de praktijk bevestigd, zoals blijkt uit figuur 5.6. Deze figuur toont, voor een aantal benchmarks, het relatieve aantal niet-lokale geheugentoegangen in het programma op basis van zelflerende dataplaatsing, uitgevoerd met vijf verschillende bemonsteringsniveaus, variërend van 1/10 tot 1/100.000. Voor elke bemonsteringsfrequentie werden 100 onafhankelijke experimenten uitgevoerd. De balken in de grafiek tonen de gemiddelde resultaten per bemonsteringsniveau, terwijl de foutbalken de minimale en maximale waarde aangeven over deze 100 onafhankelijke experimenten. Hierop kan men dus aflezen hoeveel niet-lokale toegangen er overbleven in het programma bij de minst en de meest gunstige bemonstering.

Voor *209db* leiden zowel het referentie-algoritme als de zelflerende aanpak tot een bijna-optimale dataplaatsing. Dit resultaat wordt nauwelijks beïnvloed als het zelflerende algoritme werkt met bemonsterde metingen van de geheugentoegangen. Benchmark *serial* behoort tot klasse III waarvoor lokale dataplaatsing beter werkt dan de zelflerende aanpak. Met een bemonsteringsfrequentie van 1/100 presteert het zelflerende algoritme nog redelijk goed, maar bij een grotere bemonsteringsfrequentie daalt de prestatie zeer snel.

Voor benchmarks in klasse II (*jython*, *luindex* en *hsqldb*) is het



Figuur 5.6: Fractie van niet-lokale geheugentoegangen voor een representatieve selectie van benchmarks. Deze grafiek vergelijkt de resultaten van het zelflerende algoritme op basis van een volledige instrumentatie van het programma met dataplaatsing op basis van bemonstering met vijf verschillende bemonsteringsfrequenties van 1/10 tot 1/100.000. Bemonstering doet het aandeel niet-lokale toegangen stijgen, maar de prestaties blijven meestal aanvaardbaar.

zelflerende algoritme de beste keuze die leidt tot de laagste fractie van niet-lokale geheugentoegangen, dicht tegen het optimum. Jammer genoeg sluit de extra tijd die, tijdens de uitvoering, nodig is voor het opmeten van de communicatie, elke effectieve prestatieverbetering uit in een realistisch experiment. Maar voor de benchmarks in klasse II is dat geen enkel probleem, want de prestaties van de zelflerende dataplaatser blijven intact, zelfs voor een bemonsteringsfrequentie van 1/100.000 waarbij de *overhead* van de profilering uiteraard drastisch verminderd is.

De laatste reeks van benchmarks (*228jack*, *antlr* and *227mtrt*), die behoren tot klasse I, tonen een vergelijkbaar resultaat. Zelfs met een bemonsteringsfrequentie van 1/100.000 is het prestatieverlies minimaal.

Benchmark *227mtrt* is een buitenbeentje waarbij de prestaties zelfs verbeteren wanneer bemonstering wordt gebruikt met bemonsteringsfrequentie boven 1/10.000. Het zelflerende algoritme werkt niet goed voor objecten die aangemaakt worden op een specifieke creatieplaats maar die verschillende toegangspatronen hebben: een aantal van die objecten worden gebruikt door de hardwareversnel-

ler terwijl anderen vaker gebruikt worden door de hoofdprocessor. Al deze gebruikspatronen worden immers geclusterd op dezelfde creatieplaats. Daardoor kunnen de verschillen tussen deze objecten niet adequaat in rekening gebracht worden door het zelflerende algoritme. Zoals beschreven in paragraaf 5.3, kan dit gedrag het gevolg zijn van specifieke programmeerpatronen zoals bijvoorbeeld een *class factory*. In het geval van *227mtrt* wordt het probleem niet veroorzaakt door een *class factory*. Benchmark *227mtrt* is een *raytracer*. Op verschillende creatieplaatsen worden objecten van het type `Point` aangemaakt die later gebruikt worden aan beide kanten van de hardware/software-grens. Welke methode gebruik maakt van een specifiek `Point` hangt uiteraard meer af van het beeld dat geraytracet wordt dan van de creatieplaats. Dit is een goed voorbeeld van data-afhankelijk gedrag.

Algemeen blijkt uit dit experiment dat een bemonsteringsfrequentie van 1/1000 of zelfs 1/10.000 de hoeveelheid niet-lokale geheugentoegangen niet significant doet toenemen. Tegelijkertijd wordt daardoor ook de *overhead* van de profilering, die soms oploopt tot ongeveer 100 keer de oorspronkelijke uitvoeringstijd, drastisch verlaagd met een factor 10.000, tot slechts 1% van de uitvoeringstijd. Dit maakt praktisch gebruik van het zelflerende algoritme mogelijk, waarbij de prestatieverhoging door vermindering van het aantal dure externe geheugentoegangen sterk opweegt tegen de kleine verhoging van de uitvoeringstijd ten gevolge van het profileren van geheugentoegangen.

5.4.5 Impact op de uitvoeringstijd

Uit de vorige paragrafen is het duidelijk dat deze strategieën voor dataplaatsing het aantal externe geheugentoegangen drastisch verminderen. In deze paragraaf toon ik aan dat dit ook effectief leidt tot een aanzienlijke vermindering van de uitvoeringstijd van de benchmarks.

Voor alle vorige experimenten in dit hoofdstuk, heb ik de benchmarkprogramma's uitgevoerd op een JVM en geprofileerd, zodat ik bijvoorbeeld exact kon tellen hoeveel geheugentoegangen er waren vanuit de verschillende methodes naar de verschillende objecten. Praktisch was het niet mogelijk om voor elke benchmark verschillende hardwareversnellers te programmeren. Voor het bepalen van het aantal niet-lokale geheugentoegangen, was het ook niet nodig om

die hardwareversnellers effectief te implementeren, het volstond om geheugentoegangen vanuit een aantal methodes te tellen alsof die methodes op de hardware uitgevoerd werden. Het resultaat is een exacte, accurate simulatie.

Omdat er niet voor alle programma's en alle hete code in die programma's een hardwareversneller beschikbaar is, wordt in deze paragraaf de impact van een communicatiebewuste plaatsing van de objecten op de uitvoeringstijd berekend volgens onderstaande formule.

$$t_{\text{totaal}} = t_{\text{cp}} + t_{\text{rw}}(f_n \cdot t_n + (1 - f_n)) + t_{\text{sampling}} \tag{5.1}$$

Deze formule berekent t_{totaal}: de totale uitvoeringstijd in de hardwareversnelde JVM. t_{cp} staat voor de berekeningstijd, het deel van de uitvoeringstijd van het programma op een klassieke JVM dat gebruikt wordt voor de eigenlijke berekeningen. t_{rw} is het overige deel van de uitvoeringstijd dat besteed wordt aan geheugentoegangen. f_n is de fractie niet-lokale toegangen in het programma zoals opgemeten in paragraaf 5.4.1, t_n is de relatieve toegangstijd voor niet-lokale geheugentoegangen. t_n is afhankelijk van de communicatie-infrastructuur op de gebruikte architectuur. Dat kan een PCI-bus zijn zoals in figuur 5.1, maar ook snellere bussen of communicatiekanalen zijn uiteraard mogelijk. t_{sampling} is de *overhead* voor bemonstering.

Merk op dat bij de berekening van t_{totaal} wordt verondersteld dat er geen overlap is tussen de berekeningen (t_{cp}) en de communicatie (t_{rw}). Als dat op een concrete architectuur wel het geval is, geeft deze simulatie dus een overschatting van de totale uitvoeringstijd.

De totale uitvoeringstijd t_{totaal} omvat dus de communicatieoverhead die ontstaat wanneer een deel van de code wordt afgesplitst van de hoofdprocessor. De veronderstelling in deze berekening is dat de hardwareversneller *geen* versnelling oplevert, maar dat hij de code exact even snel uitvoert als de hoofdprocessor dat zou doen. Op die manier geeft de formule de vertraging weer veroorzaakt door het afsplitsen van code van de hoofdprocessor. Omgekeerd kan men hieruit ook afleiden welke versnelling de hardwareversneller minimaal moet halen om de extra communicatiekost terug te winnen.

Voor een benchmark van elke klasse in figuur 5.4 vergelijk ik de berekende uitvoeringstijden van het programma bij gebruik van de referentie-aanpak, lokale dataplaatsing en twee bemonsterende implementaties van het zelflerende algoritme in figuur 5.7. De uitvoeringstijd wordt genormaliseerd weergegeven, waarbij de uitvoeringstijd van het programma op de hoofdprocessor wordt be-

schouwd als referentiepunt. De figuur toont dus aan hoe sterk het programma zou vertragen, als gevolg van niet-lokale geheugentoegangen, als een deel ervan wordt uitgevoerd op een hardwareversneller met versnelling 1. Omgekeerd geeft de figuur dus aan welke versnelling de hardwareversneller zou moeten halen, als hij de vertraging ten gevolge van de bijkomende communicatiekost in de hardware/software co-ontwerpoplossing wil compenseren.

Voor *serial*, een benchmark uit klasse III met 69 % niet-lokale geheugentoegangen in de referentie-aanpak blijkt uit de volle lijn in de figuur 5.7 dat de uitvoeringstijd met een factor 2,85 toeneemt als de niet-lokale toegangen 10 keer zoveel tijd in beslag nemen als lokale toegangen. Hardwareversnelling voor *serial* is in deze architectuur dus pas zinvol als de co-processor op de FPGA een versnelling haalt van meer dan een factor 2,85.

In een architectuur waar de relatieve kost van niet-lokale toegangen 100 is zal de uitvoeringstijd zelfs toenemen met een factor 21, terwijl lokale dataplaatsing (stippellijn), de beste strategie voor *serial*, leidt tot een toename van slechts 18 %. De zelflerende strategie (streepjeslijn) leidt tot een vergelijkbare reductie van de niet-lokale toegangen als lokale dataplaatsing, maar deze strategie vergt wel een zekere verhoging van de uitvoeringstijd voor de instrumentatie van het programma.

In figuur 5.7 ben ik uitgegaan van een extra vertraging van een factor 10 wanneer alle geheugentoegangen geprofileerd worden of een factor 2 wanneer slechts 1/10 van de toegangen wordt opgemeten. Dit is veel minder dan de vertraging geïntroduceerd door de profilering die ik in hoofdstuk 3 heb gebruikt. Daar was de *overhead* van het profileren niet cruciaal. Voor de implementatie van het zelflerend algoritme wordt de profilering best ingebouwd in de JVM of via verticale instrumentatie met, bijvoorbeeld, Javana (Maebe et al., 2006b). Op die manier is een factor 10 of minder wel haalbaar.

De vorm van de curve van het zelflerende algoritme met bemonsteringsfrequentie 1/10 is vergelijkbaar met die van de lokale dataplaatsing, omdat ook de vermindering van het aantal niet-lokale geheugentoegangen vergelijkbaar is, maar de *overhead* van de instrumentatie maakt het programma nog steeds twee keer trager. Vanaf een bemonsteringsfrequentie van 1/10.000 is de *overhead* door instrumentatie verwaarloosbaar, maar dan bevat het programma wel weer iets meer niet-lokale geheugentoegangen, zoals uitgelegd in paragraaf 5.4.4.

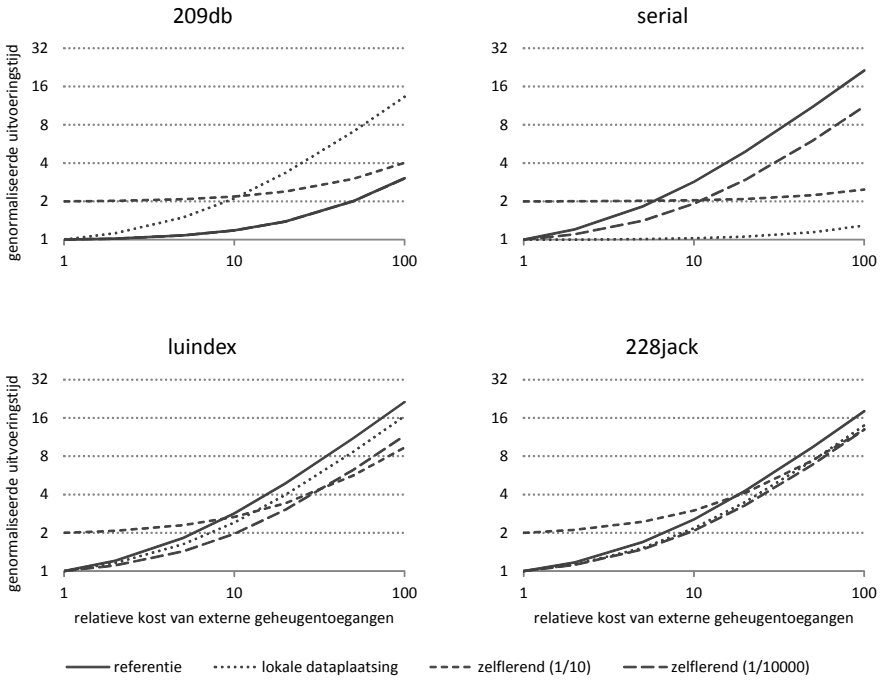
Net als in de vorige experimenten is benchmark *209db* opnieuw een speciaal geval: de referentie-aanpak presteert zeer goed, waardoor de uitvoeringstijd zelfs in het extreme scenario, waarbij niet-lokale toegangen 100 keer duurder zijn dan lokale toegangen, slechts toeneemt met een factor 3. Lokale dataplaatsing leidt tot een vertraging met een factor 13 in hetzelfde scenario. Het zelflerende algoritme reduceert het aantal niet-lokale toegangen in gelijke mate als de referentie-aanpak. Daarom valt de curve voor een bemonsteringsfrequentie van 1/10.000 —waar de *overhead* door instrumentatie verwaarloosbaar is— samen met die van de referentie-aanpak. Voor een bemonsteringsfrequentie van 1/10 wordt er geen betere plaatsing van de objecten gevonden, de beste plaatsing is immers die van de referentie-aanpak, en kan de extra vertraging ten gevolge van instrumentatie van het zelflerende algoritme dus niet terugverdiend worden.

Voor benchmarkprogramma's in klasse II presteert het zelflerende algoritme veel beter dan lokale dataplaatsing. Hiervan is *luindex* een mooi voorbeeld. De zelflerende aanpak geeft hier de grootste reductie van het aantal niet-lokale geheugentoegangen. Met een bemonsteringsfrequentie van 1/10.000 geeft dit al een verbetering van de effectieve uitvoeringstijd van het programma vanaf het moment waarop niet-lokale geheugentoegangen twee keer duurder zijn dan lokale toegangen. Wanneer de relatieve kost van niet-lokale toegangen meer dan 50 is, loont het zelfs de moeite om het zelflerende algoritme te gebruiken met een bemonsteringsfrequentie van 1/10.

Benchmark *228jack* is een voorbeeld van klasse I waarvoor het zelflerende algoritme en lokale dataplaatsing een vergelijkbare vermindering van het aantal niet-lokale geheugentoegangen bewerkstelligen. De referentie-aanpak doet de uitvoeringstijd toenemen met een factor 18 voor het meest extreme scenario, terwijl de uitvoeringstijd met beide alternatieve algoritmes slechts 13 keer groter wordt. Alle objecten in het meest optimale geheugen plaatsen levert hier dus een prestatiewinst op van ongeveer 27%.

5.5 Verwant werk

In de literatuur werden verschillende benaderingen voorgesteld om een generieke processor uit te breiden met toepassingsspecifieke hardwareversnellers. Het kernidee daarbij is steeds dat de hardwareversneller het mogelijk maakt om het inherente parallellisme



Figuur 5.7: Genormaliseerde uitvoeringstijd in functie van de relatieve kost van niet-lokale geheugentoegangen ten opzichte van lokale geheugentoegangen.

in de toepassing effectief uit te buiten. Daarin verschilt de aanpak die ik in dit werk voorstel niet van gelijkaardig werk. Het verschil zit in de manier waarop de communicatie tussen de processor en de hardwareversneller gebeurt. In sommige van de bestaande hybride systemen, zoals de MOLEN-processor (Vassiliadis et al., 2004) en de WARP-processor (Lysecky et al., 2006) zijn zowel de hoofdprocessor als de hardwareversneller verbonden met hetzelfde gedeelde geheugen. Dit staat in tegenstelling tot mijn aanpak en de aanpak beschreven door Lattanzi et al. (2005) waar het gedeelde geheugen bestaat uit verscheidene fysische geheugens met erg verschillende toegangstijden. In een dergelijke architectuur met niet-uniform gedeeld geheugen zijn strategieën om dynamisch objecten toe te wijzen aan de verschillende geheugens onontbeerlijk.

Lattanzi et al. (2005) verdeelt het geheugen in drie delen: een tweepoortsgeheugen dat toegankelijk is door beide componenten en twee eenpoortsgeheugens in de vorm van lokaal geheugen van elk van de twee componenten. Java-objecten die gebruikt worden door beide componenten worden specifiek in het tweepoortsgeheugen geplaatst. In een dynamische omgeving waarbij functionaliteit van het programma vlot kan verplaatst worden van de hoofdprocessor naar de hardwareversneller, leidt dit tot een situatie waarbij alle objecten die gebruikt worden door methodes die eventueel ooit op de hardwareversneller uitgevoerd zouden kunnen worden, in het tweepoortsgeheugen moeten worden geplaatst. Ook wanneer die methodes momenteel op de hoofdprocessor worden uitgevoerd. In de limiet vereist dit dat het volledige geheugen tweepoortsgeheugen wordt, wat de voordelen van éénpoortsgeheugen teniet doet.

De componenten in de hybride architectuur besproken in dit hoofdstuk hebben elk toegang tot hun eigen lokaal geheugen en tot het lokaal geheugen van de andere component, zij het dat die niet-lokale geheugentoegangen heel wat trager zijn. Ik heb hierboven aangetoond dat lokale dataplaatsing en het zelflerende algoritme het aantal niet-lokale geheugentoegangen en de bijbehorende communicatiekost drastisch kan doen dalen. Bovendien is er in deze aanpak geen fysisch gedeeld geheugen nodig zoals dat bij Lattanzi et al. (2005) wel het geval is.

De strategieën voor dataplaatsing die ik in dit hoofdstuk heb ontwikkeld doen de communicatiekost in de hybride architectuur significant dalen. Deze technieken komen goed van pas in het statische geval waarbij de verdeling van de functionaliteit tussen de hoofd-

processor en de hardwareversneller op voorhand vastligt. Het gedrag van het programma kan immers invoerafhankelijk zijn waardoor een vaste dataplaatsing, die dezelfde is voor alle uitvoeringen van het programma, niet optimaal is. In het geval van een dynamische, hybride architectuur met een herconfigureerbare hardwareversneller die door de JVM wordt gebruikt om *on-the-fly* hardware te genereren, zijn deze technieken voor dynamische dataplaatsing onmisbaar.

Het is vandaag nog niet mogelijk om voor elke willekeurige methode, tijdens de uitvoering van het programma, een hardwareversneller aan te maken, maar verschillende onderzoekers hebben toch al aangetoond dat het voor kleine stukjes code reeds mogelijk is in een *fine grain* aanpak. Een praktische implementatie hiervan kan gevonden worden in de WARP-processor (Lysecky et al., 2006), een nieuwe processorarchitectuur die bestaat uit een microprocessor en een FPGA. Tijdens de uitvoering van een programma op de microprocessor, wordt het programma geprofileerd om zo hete codefragmenten te ontdekken. Het gaat hier om relatief korte fragmentjes, van enkele instructies, waarvoor een hardwareversneller wordt gegenereerd op de FPGA. Een gelijkaardige methodologie werd ook toegepast op Java-programma's (Beck & Carro, 2005).

Een dergelijke aanpak voor het *on-the-fly* genereren van hardwareversnellers is voorlopig nog te fijnkorrelig om rechtstreeks in de hybride architectuur van dit werk ingepast te worden. Toch tonen de resultaten aan dat dynamische dataplaatsingstechnieken voor een dergelijke architectuur in de toekomst aan belang kunnen winnen.

5.6 Besluit

Hoewel toepassingsspecifieke hardwareversnellers de prestaties van een JVM aanzienlijk kunnen versnellen, wordt de effectieve prestatiewinst in de praktijk vaak beperkt door de extra communicatiekost die nodig is om die hardwareversnellers aan te spreken. In de zonet besproken hybride architectuur wordt deze kost veroorzaakt door de niet-uniformiteit van de toegangstijden tot het gedistribueerde geheugen dat gevormd wordt door het hoofdgeheugen van de processor en het lokale geheugen van de hardwareversneller.

Ik heb in dit hoofdstuk verscheidene technieken voorgesteld die de optimale geheugenlocatie zoeken voor elk Java-object, waardoor de communicatie kon verminderd worden met gemiddeld 43% voor

de programma's van de SPECjvm en DaCapo benchmarks met uitschieters tot 86%. Dit leidt tot een vermindering van de uitvoeringstijd van die programma's met een factor 1,64 op een architectuur waar niet-lokale geheugentoegangen 20 keer langzamer zijn dan lokale toegangen. Voor een eerste klasse van benchmarks, presteert het algoritme voor lokale dataplaatsing het best. Die techniek is daarenboven erg efficiënt te implementeren zonder dat er tijdens de uitvoering tijd verloren gaat door instrumentatie en profilering. Voor een tweede klasse van benchmarks kan mijn zelflerend algoritme de communicatie verder terugdringen door tijdens de uitvoering dynamisch de beste lokatie in het geheugen te kiezen op basis van opgemeten geheugentoegangen naar gelijkaardige objecten in het verleden. De kleine vertraging die veroorzaakt wordt door het opmeten van deze toegangspatronen, kan binnen de perken gehouden worden door het gebruik van bemonstering: zelfs met een bemonsteringsfrequentie van 1/10.000 blijkt het mogelijk om de communicatiekost significant terug te dringen. De prestatiewinst die gepaard gaat met de vermindering van het aantal niet-lokale geheugentoegangen is in dat geval vele malen groter dan het kleine verlies veroorzaakt door de vertraging van het profileren.

Hoofdstuk 6

Besluit

In dit laatste hoofdstuk kijk ik achterom naar de boeiende tocht die dit doctoraatsonderzoek voor mij geweest is en naar de concrete verwezenlijkingen die eruit zijn voortgevloeid. Eerst volgt een beknopte samenvatting van de drie belangrijkste bijdragen. Verder blikt dit besluit ook vooruit naar mogelijke nieuwe toepassingen voor de concepten uit dit proefschrift of potentiële uitbreidingen die een aanzet kunnen vormen voor verder onderzoek of valorisatie.

6.1 Samenvatting

Het uitgangspunt van dit doctoraatsproefschrift is de vaststelling dat communicatie cruciaal is in gedistribueerde rekensystemen. Door de evolutie naar een steeds groeiend aantal rekenkernen op één enkele chip, wordt communicatie tussen die kernen steeds belangrijker. Binnen een dergelijk gedistribueerd rekensysteem verloopt de communicatie op verschillende niveaus, elk met hun eigen karakteristieke prestatiekenmerken. Draden die op dezelfde kern worden uitgevoerd, kunnen communiceren via gedeeld geheugen in een *cache* of in een extern geheugen dat enkel via het netwerk bereikt kan worden. Draden die op verschillende kernen uitgevoerd worden, kunnen enkel informatie uitwisselen via een relatief trage netwerkverbinding.

De verhouding tussen interne en externe communicatie heeft een fundamentele impact op de uiteindelijke prestaties wanneer een programma wordt opgesplitst over de verschillende rekenkernen in het systeem. Een geschikte verdeling van zowel de berekeningen als de

data over de verschillende rekenkernen, is dan ook uitermate belangrijk om tot een efficiënt resultaat te komen.

6.1.1 Opmeten van communicatieprofielen

Om een goed beeld te krijgen van hoeveel communicatie er nodig is, heb ik een profileerder ontwikkeld die communicatieprofielen opmeet van Java-programma's. Daarin wordt de informatie over datastromen in het programma gecombineerd met de dynamische oproepgraaf van het programma.

Hierbij wordt een onderscheid gemaakt tussen twee types communicatieprofielen: een eerste dat enkel de inherente communicatie bevat tussen methodes in het programma zonder rekening te houden met de datastructuren die daarvoor gebruikt worden en een tweede profiel dat expliciet de communicatie opmeet tussen methodes en datastructuren (Bertels et al., 2008). Beide profielen hebben hun eigen specifieke toepassingen.

Het eerste communicatieprofiel kan door de ontwerper gebruikt worden als aanzet voor de partitionering van functionaliteit van programma's over meerdere rekenkernen of parallelle draden. De communicatie in een systeem is immers onafhankelijk van de specifieke implementatie en de inherente communicatie opgemeten in de geprofileerde, initiële implementatie, is dan ook een goede maat voor de communicatie in de uiteindelijke implementatie op een gedistribueerd rekensysteem.

Het tweede communicatieprofiel meet de communicatie van en naar datastructuren in Java. Het levert een goed beeld op van het gebruik van Java-objecten doorheen het programma en geeft dus meer informatie over concrete optimalisaties van de datalayout. Dit profiel vormt de basis voor mijn zelflerende algoritme om de plaatsing van objecten te optimaliseren.

Profileren brengt onvermijdelijk een vertraging met zich mee. Zeker voor het opmeten van het eerste type communicatieprofiel is de toename van de uitvoeringstijd enorm omdat er een grote boekhouding moet worden bijgehouden van alle lees- en schrijfoperaties en van een schaduwobject voor elk object in het geheugen. De profilering kan fors versneld worden door gebruik te maken van bemonstering, maar dat kan dan weer nadelig zijn voor de kwaliteit van het opgemeten communicatieprofiel.

In dit proefschrift gebruik ik met succes een bemonsteringstechniek uit de oude doos, *reservoir sampling*, om de overlast van het profileren te reduceren met een factor 9, met behoud van een aanvaardbare, en vooraf statistisch vastgelegde, nauwkeurigheid. Dit resultaat werd gevalideerd aan de hand van een hele reeks benchmarkprogramma's uit de SPECjvm benchmark suite (Standard Performance Evaluation Corporation, 1998).

6.1.2 Communicatiebewuste partitionering

De communicatieprofielen vormen de basis voor een communicatiebewuste partitionering van de functionaliteit van programma's waarbij de verhouding tussen interne en externe communicatie als eerste criterium wordt gebruikt.

Hierbij focus ik op een specifieke vorm van partitionering, met name het identificeren van delen van de functionaliteit van een systeem die geschikt zijn om ze af te zonderen van de kern van het systeem. Dit wordt uitgewerkt in het kader van de hardwareversnelde Java Virtuele Machine (JVM) waarbij een klassieke processor de initiële, hoofdpartitie, voor zijn rekening neemt en één of meerdere hardwareversnellers op een Field Programmable Gate Array (FPGA) als co-processor de afgezonderde partities voor hun rekening nemen. Faes et al. (2009) toonden immers aan dat een dergelijke aanpak voor de samenwerking tussen hardware en software interessante resultaten oplevert in de context van Java en de hardwareversnelde JVM.

Ik heb twee methodes uitgewerkt om programma's functioneel te partitioneren op basis van de opgemeten communicatieprofielen: statische partitionering en dynamische partitionering. Beide types hebben hun duidelijke voor- en nadelen en hun specifieke toepassingen.

Statische partitionering is bedoeld als ondersteuning voor een ontwerper die met de hand een programma partitioneert. In de oproepgraaf selecteert men een aantal methodes die geschikt zijn voor hardwareversnelling, bijvoorbeeld omdat ze veel intern parallellisme bevatten, en een aantal methodes die absoluut niet op de hardware uitgevoerd mogen worden, bijvoorbeeld omdat ze complexe controlestructuren bevatten.

De statische partitionering gaat dan incrementeel op zoek naar een communicatiebewuste partitionering die een geschikte grens tussen hardware en software vastlegt. Typisch worden dan niet enkel

de aangeduide methodes overgeheveld naar de co-processor, maar worden ook enkele omliggende methodes mee verplaatst. Wanneer die omliggende methodes zeer veel data uitwisselen met de hardwareversnelde methodes, is het immers beter dat die communicatie volledig op de co-processor kan gebeuren.

Dynamische partitionering is een uitbreiding van het statische algoritme voor gebruik in de hardwareversnelde JVM. De Just-in-Time (JIT)-compilatie in deze JVM opent immers perspectieven om hardwareversnelling te beschouwen als een bijkomende stap in het compilatieproces. Dit heeft een aantal voordelen. Zo is in een aantal toepassingen en voor een aantal methodes de verhouding tussen berekeningen en communicatie invoerafhankelijk. Een dynamische aanpak kan hier adequaat op reageren en in de ene uitvoering een concrete methode wel en in de andere uitvoering niet afleiden naar de co-processor op de FPGA.

6.1.3 Communicatiebewust geheugenbeheer

De hardwareversnelde JVM maakt het mogelijk om op een transparante manier aan hardware/software co-ontwerp te doen op een gemengd platform met een generieke processor en hardwareversnellers op FPGA als co-processor.

Om de prestaties te verbeteren beschikken zowel de hoofdprocessor als de hardwareversneller over hun eigen lokale geheugen. Beide fysieke geheugens zijn verenigd in het gedeelde-geheugenmodel van de JVM. Deze transparante hardwareversnelde JVM beheert de toegang tot alle objecten in het geheugen, ongeacht hun fysieke locatie. De hardwareversneller is meestal via een relatief traag communicatiemedium verbonden met de hoofdprocessor en het hoofdgeheugen. Daarom worden geheugentoegangen naar 'het andere geheugen' erg duur. Die moeten dus zo veel mogelijk vermeden worden.

Een belangrijke taak van de JVM is het zoeken van de meest geschikte geheugenlocatie van elk object in de gedistribueerde Java *heap*. De objecten zitten best in het geheugen dat het dichtst staat bij de processor (of versneller) die de data het vaakst gebruikt. Gegevens die enkel binnen een draad gebruikt worden, zullen zich dan steeds in het lokale geheugen bevinden met een minimalisatie van de externe communicatie tot gevolg. Voor data die gedeeld wordt tussen uitvoeringsdraden wordt gestreefd naar een communicatiebewuste geheugenallocatie.

Ik heb verschillende technieken voorgesteld om communicatiebewust geheugen toe te wijzen. Dynamisch bepaalt de JVM voor elk Java-object, de optimale geheugenlocatie. Mijn zelflerende aanpak probeert de gebruikspatronen voor elk object te schatten op basis van gemeten patronen voor objecten die eerder werden toegewezen (Bertels et al., 2009). Op basis hiervan kan het object geplaatst worden in het meest geschikte geheugen. Mijn strategieën voor het plaatsen van data in de geheugens leiden tot een vermindering van de hoeveelheid externe geheugentoegangen tot 86% (49% gemiddeld) voor de SPECjvm en DaCapo benchmarks (Blackburn et al., 2006; Standard Performance Evaluation Corporation, 1998, 2008).

6.2 Toekomstig werk

Tot slot wil ik een aantal potentieel interessante pistes voor verder onderzoek aanhalen. Mijn werk heeft aangetoond dat het nuttig en ook mogelijk is om binnen de JVM communicatie-optimalisaties door te voeren die uiteindelijk leiden tot een betere synergie tussen het programma en het hybride, gedistribueerde rekensysteem waarop het programma wordt uitgevoerd.

De concepten uitgewerkt in dit proefschrift zijn toegepast in de context van één specifiek en concreet platform, met name de hardwareversnelde JVM van Faes et al. (2009). De achterliggende ideeën kunnen echter in een ruimer kader hun nut bewijzen. In volgende paragrafen haal ik een aantal voorbeelden aan van toepassingen en de uitbreidingen die eventueel nodig zijn om die toepassingen effectief te verwezenlijken.

6.2.1 Op weg naar volautomatische hardwareversnelling

Het baanbrekende werk van Faes (2008) heeft geleid tot een effectieve implementatie van zijn hardwareversnelde JVM met concrete resultaten voor hardware/software co-ontwerp van een systeem voor het vergelijken van DNA-sequenties op basis van het algoritme van Smith & Waterman (1981). Hoewel de hardwareversneller die hiervoor werd ontwikkeld in de masterscriptie van Minnaert (2005) een zeer respectabele versnelling behaalde van de kritieke lussen in het algoritme, bleef daar na integratie in de hardwareversnelde JVM slechts een fractie van over. Een groot deel van de oorspronkelijk behaalde snelheidswinst ging immers verloren in de communicatie

tussen het Java-programma en de hardwareversneller op de FPGA. Het verbeteren van de communicatie speelt dus een cruciale rol in de optimalisatie van dit systeem.

In dit proefschrift heb ik simulaties gedaan van een groter aantal benchmarkprogramma's op een model van de hardwareversnelde JVM. Ook hier bleek het belang van communicatie. Figuur 5.7 toont, in functie van de vertraging veroorzaakt door het communicatiemedium tussen processor en co-processor, duidelijk aan hoeveel tijd verloren gaat in die communicatie. Voor het uiterste geval in mijn simulatie, moet de hardwareversneller op zich al minstens een prestatiewinst opleveren van een factor 16 vooraleer het systeem in zijn geheel enige snelheidswinst kan boeken. Met mijn zelflerende algoritme voor communicatiebewust geheugenbeheer, kan ik dit fors verbeteren. Zelfs vanaf een hardwareversnelling van een factor 3 wordt het dan efficiënter om de hardwareversnelling ook effectief te gaan gebruiken.

Deze simulaties geven dus aan dat het nuttig zou zijn om een demonstrator te maken die de bestaande hardwareversnelde JVM uitbreidt met mijn communicatiebewust geheugenbeheer. Een verdere stap zou de integratie van *on-the-fly* hardwaregeneratie zijn. Die stap maakt het immers mogelijk om het systeem te doen evolueren naar een volautomatische omgeving waarin hardwaregeneratie kan worden gezien als een extra optimalisatiestap in het JIT-compilatieproces. Daarin speelt dan ook de communicatiebewuste partitionering, uitgewerkt in hoofdstuk 4, zijn rol. De beslissing om een methode al dan niet in hardware uit te voeren is immers een vorm van dynamische hardware/software partitionering.

6.2.2 Systemen met meerdere rekenkernen

Het partitioneringsalgoritme dat ik heb uitgewerkt in hoofdstuk 4 deelt de functionaliteit van een programma op in meerdere partities die uitgevoerd kunnen worden door meerdere hardwareversnellers. Afhankelijk van de complexiteit van die hardwareversnellers, kunnen zij samen op één FPGA geplaatst worden, of zijn meerdere rekenkernen (FPGA's of andere types co-processors) nodig.

De aanpak voor communicatiebewuste plaatsing van objecten in het geheugen, uitgewerkt in hoofdstuk 5, is in dit werk enkel onderzocht voor het geval waarbij twee rekenkernen het werk uitvoeren: een hoofdprocessor en een FPGA. Verder onderzoek moet uitwijzen

in welke mate de uitbreiding naar meerdere hardwareversnellers op verschillende rekenkernen mogelijk is.

Cruciaal in de aanpak voorgesteld in dit proefschrift, is dat het systeem bestaat uit één centrale rekenknoop die alle co-processors aanstuurt en die ook het geheugenbeheer voor die co-processors regelt. Zo ontstaat een globaal gedeeld geheugen waarbij elke knoop, als dat nodig is, alle data in het systeem kan lezen of schrijven, uiteraard met een bijkomende communicatiekost voor niet-lokale data.

De aanpak voor lokale dataplaatsing kan eenvoudig in een dergelijk systeem geïmplementeerd worden: elke rekenknoop kan zijn eigen data in zijn eigen geheugen plaatsen. De keuze voor de plaatsing is bij deze aanpak immers binair: data wordt lokaal geplaatst op de knoop die data aanmaakt of niet-lokaal en dan komt het object in het centrale hoofdgeheugen terecht. Hieraan verandert niets ten opzichte van een systeem met slechts een hardwareversneller.

Het zelflerende algoritme kan ook aangepast worden voor meerdere rekenknopen. Voor elke creatieplaats in het programma moeten dan voor elke rekenknoop tellers bijgehouden worden die de geheugentoegangen vanuit die verschillende rekenknopen, tellen. Elk nieuw object dat wordt aangemaakt op een bepaalde creatieplaats, wordt dan in het geheugen geplaatst van de rekenknoop die tot dan toe de meeste lees- en schrijfoperaties uitvoerde naar die groep van objecten. De keuze is hier echter niet meer binair. Als rekenknoop n een object aanmaakt, dan zal het algoritme dit object niet noodzakelijk op rekenknoop n plaatsen of in het hoofdgeheugen. Het object kan ook rechtkomen op het geheugen van rekenknoop m.

Ik verwacht dat voor een groot aantal creatieplaatsen zal gelden dat de objecten aangemaakt op die creatieplaatsen hoofdzakelijk door één rekenknoop gebruikt zullen worden. Uit mijn experimenten blijkt dat lokale dataplaatsing relatief goed scoort voor een aantal benchmarks en dat is precies omwille van de aanwezige datalokaliteit die stelt dat objecten hoofdzakelijk gebruikt worden door de rekenknoop die ze aanmaakt.

Voor andere creatieplaatsen is het gebruik van de objecten wellicht meer verspreid over de rekenknopen. Als er geen duidelijke 'winnaar' is, dan zal het algoritme uiteraard minder goed presteren. De objecten worden dan geplaatst in het geheugen van de rekenknoop die eerder toevallig net iets meer datatoegangen heeft naar die objecten.

Als het systeem niet meer communiceert over een globale bus,

maar via een netwerk op de chip, biedt dat extra mogelijkheden. Men zou het algoritme kunnen uitbreiden om objecten niet meer altijd te plaatsen op de rekenknoop die ze het vaakst nodig heeft, maar ergens 'in het midden' tussen de rekenknopen die de objecten gebruiken. Dit biedt misschien een uitweg voor de situaties waarbij meerdere rekenknopen de objecten van een creatieplaats gebruiken. De tellers worden dan gewichtsfactoren die de objecten in meer of mindere mate aantrekken naar elke rekenknoop toe.

Als met bovenstaande aandachtspunten rekening wordt gehouden, verwacht ik dat mijn algoritme kan schalen tot een systeem met een aantal rekenkernen. Voor parallelle multicore-architecturen met honderden rekenkernen is mijn zelflerend algoritme niet geschikt. Het is echter wel mogelijk om een of enkele multicores als co-processor aan te sluiten op de hoofdprocessor. In dat geval kan mijn zelflerend algoritme de communicatie tussen de hoofdprocessor en elk van deze multicore co-processors regelen, terwijl elke multicore op zich intern zijn eigen geheugenbeheer uitwerkt.

6.2.3 Flexibele hardwareplatformen

Het onderzoek voorgesteld in dit proefschrift, past binnen het ruimere kader van FlexWare, een project voor strategisch basisonderzoek van IWT-Vlaanderen, met als missie het uitbuiten van flexibele hardwareplatformen voor massief parallelle toepassingen in de bio-informatica. In dit project onderzoeken IMEC, de bio-informatica-onderzoeksgroep PSB, Dekimo en mijn eigen onderzoeksgroep Paris, de mogelijkheid om systematisch en semi-automatisch het meest geschikte platform te bepalen voor de implementatie van een specifiek algoritme. Hierbij wordt een veelheid van platformen in overweging genomen, waaronder FPGA's, maar ook Digital Signal Processors (DSP's) en *coarse grain* herconfigureerbare processorarchitecturen.

Mijn profileerder en de daarop gebaseerde partitioneerder werden gebruikt in het kader van het eerste werkpakket van dit project. Samen met een onderzoeksluik over hoog-niveau-lustransformaties, draagt mijn werk er bij tot de extractie van parallellisme uit algoritmes. De opgemeten communicatieprofielen wijzen de ontwerper immers de weg naar een gepaste partitionering en een geschikte implementatie. De bedoeling is om na de platformonafhankelijke profilering en richtinggevende partitionering verder te gaan naar platformspecifieke optimalisaties op de verschillende processorarchitecturen

en op de FPGA die in de verdere werkpakketten aan bod komen.

6.2.4 PinComm: een uitbreiding voor C en C++

Op 1 januari 2008 startte, eveneens met steun van IWT-Vlaanderen, een ander strategisch basisonderzoeksproject, OptiMMA, dat gaat over de optimalisatie van Multiprocessor System-on-Chip (MPSOC)-platformen voor *event-driven* toepassingen. Net als in mijn eigen doctoraatsonderzoek, ligt de focus ook in het OptiMMA-project op de dynamiek van de programma's die een *runtime* aanpak onontbeerlijk maken. OptiMMA kiest hiervoor expliciet voor een aanpak gebaseerd op scenario's.

Heirman et al. (2009) ontwikkelden in het kader van dit project de profileerder PinComm die communicatieprofielen opmeet van C-programma's. Dit is een rechtstreekse implementatie van de concepten uitgewerkt in dit proefschrift, maar dan voor C en C++ in plaats van voor Java.

De communicatieprofielen worden in OptiMMA gebruikt voor partitionering en geheugenallocatie in een multicore context. Het inzicht verkregen in de communicatiestromen binnen een programma, maakt de weg vrij voor nieuwe toepassingen als communicatiebewuste parallellisering, *mapping* en configuratie van het *on-chip* communicatienetwerk.

6.3 Epiloog

In dit werk werd het aspect communicatie in een gedistribueerd rekensysteem op basis van een hoofdprocessor en een hardware co-processor, bestudeerd en geoptimaliseerd. Ik heb aangetoond dat communicatie een cruciale impact heeft op de algehele prestaties van een dergelijk systeem.

Ik heb profileringstechnieken ontwikkeld om communicatieprofielen in programma's op te meten. Het daardoor verkregen inzicht in de systeemcommunicatie heb ik met succes gebruikt voor communicatiebewuste partitionering van de functionaliteit tussen een processor en een hardwareversneller als co-processor.

Tot slot, stel ik met trots vast dat de hardwareversnelde JVM, dankzij mijn zelflerend algoritme voor geheugenbeheer, voor een aantal toepassingen heel wat interessanter is geworden. Door de

drastische vermindering van de communicatiestromen in het systeem, weegt de snelheidswinst door gebruik van een FPGA als coprocessor, nu in veel meer gevallen sterk op tegen de bijbehorende communicatiekost. Hierdoor komt efficiënte en volautomatische hardwareversnelling weer een stapje dichterbij.

Publicaties

Artikels in tijdschriften

1. **Peter Bertels**, Wim Heirman, Erik D'Hollander and Dirk Stroobandt. Efficient Memory Management for Hardware Accelerated Java Virtual Machines. *ACM Transactions on Design Automation of Electronic Systems*, Volume 14, Article 48, pp. 1–18, 2009.

2. Philippe Faes, **Peter Bertels**, Jan Van Campenhout and Dirk Stroobandt. Using Method Interception for Hardware/Software Co-Development. *Springer Design Automation for Embedded Systems*, Volume 13, Issue 4, pp. 223–243, 2009.

3. **Peter Bertels**, Michiel D'Haene, Tom Degryse and Dirk Stroobandt. Teaching Skills and Concepts for Embedded Systems Design. *ACM SIGBED Review*, 6(1):1–8, 2009.

Bijdragen op conferenties

1. **Peter Bertels** and Dirk Stroobandt. How Java Increases Flexibility and Run-Time Efficiency of MPSoC Systems. In *Proceedings of the 20th Annual ProRISC Workshop*, pp. 437–440, 2009.

2. **Peter Bertels**, Wim Heirman and Dirk Stroobandt. Strategies for Dynamic Memory Allocation in Hybrid Architectures. In *Proceedings of the ACM International Conference on Computing Frontiers*, pp. 217–220, 2009.

3. **Peter Bertels** and Dirk Stroobandt. Automatic Approach Towards Actor-Oriented Programming. In *Proceedings of the 19th Annual ProRISC Workshop*, pp. 16–19, 2008.

4. **Peter Bertels**, Wim Heirman and Dirk Stroobandt. Efficient Measurement of Data Flow Enabling Communication-Aware Parallelisation. In *Proceedings of the First International Forum on Next-Generation Multicore and Manycore Technologies (IFMT)*, pp. 43–49, 2008. ACM.

5. **Peter Bertels** and Dirk Stroobandt. Java and the Power of Multi-Core Processing. In *Proceedings of the 2nd International Conference on Complex, Intelligent and Software Intensive Systems*, pp. 627–631, 2008.

6. **Peter Bertels** and Dirk Stroobandt. Data Migration in Hardware Accelerated Java Virtual Machines. In *Proceedings of the 18th Annual ProRISC Workshop*, pp. 288–291, 2007.

7. **Peter Bertels**, Michiel D'Haene, Tom Degryse and Dirk Stroobandt. Gathering Skills for Embedded Systems Design. In *Proceedings of the 3th Workshop on Embedded Systems Education*, Embedded Systems Week, pp. 30–35, 2007.

8. Karel Bruneel, **Peter Bertels** and Dirk Stroobandt. A Method for Fast Hardware Specialization at Run-time. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, pp. 35–40, 2007.

9. **Peter Bertels** and Dirk Stroobandt. Profiling Based Estimation of Communication for System Partitioning. In *Proceedings of the 17th Annual ProRISC Workshop*, pp. 233–239, 2006.

10. **Peter Bertels** and Dirk Stroobandt. Heuristic for Profiling Bandwidths in Object Oriented Applications. In *Proceedings of the 16th Annual ProRISC Workshop*, pp. 475–478, 2005.

Overige publicaties

1. **Peter Bertels**. Analysing Communication Leads to More Efficient Systems. *Design Automation and Test in Europe*, 2009.

2. **Peter Bertels**. Automatic Approach towards Actor-oriented Programming. *Ninth FirW PhD Symposium*, pp. 157–158, 2008.

3. **Peter Bertels**. Java and the Power of Multi-Core Processing. *Eighth FirW PhD Symposium*, pp. 165–166, 2007.

4. **Peter Bertels** and Dirk Stroobandt. Memory Allocation in Hardware/Software Co-design. *Architecture and Compilers for Embedded Systems*, pp. 71–73, 2007.

5. **Peter Bertels**. Profilation directed design of embedded systems. *Sixth FirW PhD Symposium*, pp. 209–210, 2005.

Bibliografie

Adair, R. J., Bayles, R. U., Comeau, L. W., & Creasy, R. J. (1966). *A Virtual Machine System for the 360/40*. Technical Report 320-2007, IBM Corporation, Cambridge Scientific Center.

Allman, E. (2004). A conversation with James Gosling. *ACM Queue*, 2(5), 24–33.

Alpern, B., Attanasio, C. R., Barton, J. J., Burke, M. G., Cheng, P., Choi, J.-D., Cocchi, A., Fink, S. J., Grove, D., Hind, M., Hummel, S. F., Lieber, D., Litvinov, V., Mergen, M. F., Ngo, T., Russel, J. R., Sarkar, V., Serrano, M. J., Shepherd, J. C., Smith, S. E., Sreedhar, V. C., Srinivasan, H., & Whaley, J. (2000). The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 211–238.

Amer, I., Lucarz, C., Mattavelli, M., Roquier, G., Raulet, M., Deforges, O., & Nezan, J.-F. (2009). Reconfigurable Video Coding: the Video Coding Standard for Multi-core Platforms. *IEEE Signal Processing Magazine, Special issue on Multicore Platforms*, 26,(6), 113–123.

Arnold, M., Fink, S., Grove, D., Hind, M., & Sweeney, P. F. (2000). Adaptive optimization in the Jalapeño JVM. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (pp. 47–65). New York, NY, USA: ACM.

Aycock, J. (2003). A brief history of just-in-time. *ACM Comput. Surv.*, 35(2), 97–113.

Bala, V., Duesterwald, E., & Banerjia, S. (2000). Dynamo: a transparent dynamic optimization system. In *PLDI 2000: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (pp. 1–12). New York, NY, USA: ACM Press.

Bastoul, C. (2004). *Improving Data Locality in Static Control Programs*. PhD thesis, Université Paris 6.

Beck, A. C. S. & Carro, L. (2005). Dynamic reconfiguration with binary translation: breaking the ILP barrier with software compatibility. In *Proceedings of the 42nd annual Design Automation Conference (DAC)* (pp. 732–737). New York, NY, USA: ACM.

Bertels, P., Heirman, W., D'Hollander, E., & Stroobandt, D. (2009). Efficient memory management for hardware accelerated Java virtual machines. *ACM Transactions on Design Automation of Electronic Systems*, 14(4), 18.

Bertels, P., Heirman, W., & Stroobandt, D. (2008). Efficient measurement of data flow enabling communication-aware parallelisation. In *Proceedings of the International Forum on next-generation Multicore/manycore Technologies (IFMT)* (pp. 1–7). New York, NY, USA: ACM.

Bertels, P. & Stroobandt, D. (2006). Profiling based estimation of communication for system partitioning. In *Proceedings of the 17th Annual ProRISC Workshop* (pp. 233–239).

Beyls, K. (2004). *Software Methods to Improve Data Locality and Cache Behavior*. PhD thesis, Ghent University.

Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., & Wiedermann, B. (2006). The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (pp. 169–190). New York, NY, USA: ACM Press.

Borg, A., Gao, R., & Audsley, N. (2006). A co-design strategy for embedded Java applications based on a hardware interface with invocation semantics. In *Proceedings of the 4th international workshop on Java Technologies for Real-time and Embedded Systems (JTRES)* (pp. 58–67). New York, NY, USA: ACM.

Bornstein, D. (2008). Dalvik VM internals. In *Google I/O Developer Conference*.

Bougard, B., De Sutter, B., Verkest, D., Van der Perre, L., & Lauwereins, R. (2008). A coarse-grained array accelerator for software-defined radio baseband processing. *IEEE Micro*, 28(4), 41–50.

Box, D., Sells, C., & Miller, J. S. (2002). *Essential .NET, Volume 1, The Comon Language Runtime*. Addison-Wesley Professional.

Bruneel, K., Bertels, P., & Stroobandt, D. (2007). A method for fast hardware specialization at run-time. In K. Bertels & W. Najjar (Eds.), *Proceedings of the 2007 International Conference on Field Programmable Logic and Applications* (pp. 35–40). Amsterdam.

Bruneel, K. & Stroobandt, D. (2008a). Automatic generation of run-time parameterizable configurations. In U. Kebschull, M. Platzner, & T. J. (Eds.), *Proceedings of the International Conference on Field Programmable Logic and Applications* (pp. 361–366). Heidelberg: Kirchhoff Institute for Physics.

Bruneel, K. & Stroobandt, D. (2008b). Reconfigurability-aware structural mapping for LUT-based FPGAs. In *2008 International Conference on Reconfigurable Computing and FPGAs (ReConFig)* (pp. 223–8). Piscataway, NJ, USA: IEEE. 2008 International Conference on Reconfigurable Computing and FPGAs (ReConFig), 3-5 December 2008, Cancun, Mexico.

Bruneton, E., Lenglet, R., & Coupaye, T. (2002). ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*.

Burke, M. G., Choi, J.-D., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M. J., Sreedhar, V. C., Srinivasan, H., & Whaley, J. (1999). The Jalape no dynamic optimizing compiler for Java. In *Java '99: Proceedings of the ACM 1999 conference on Java Grande* (pp. 129–141). New York, NY, USA: ACM.

Calingaert, P. (1979). *Assemblers, Compilers, and Program Translation*. New York, NY, USA: W. H. Freeman & Co.

Catthoor, F., Wuytack, S., De Greef, E., Balasa, F., Nachtergaele, L., & Vandecappelle, A. (1998). *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Boston, USA: Kluwer Academic Publishers.

Chan, B. & Abdelrahman, T. S. (2004). Run-time support for the automatic parallelization of Java programs. *Journal of Supercomputing*, 28, 91–117.

Chen, Z. (2000). *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Dahm, M. (2001). *Byte Code Engineering with the BCEL API*. Technical report, Universität Berlin.

Deutsch, L. P. & Schiffman, A. M. (1984). Efficient implementation of the smalltalk-80 system. In *POPL 1984: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (pp. 297–302). New York, NY, USA: ACM.

Devos, H. (2008). *Loop Transformations for the Optimized Generation of Reconfigurable Hardware*. PhD thesis, Ghent University.

Eeckhaut, H., Devos, H., Lambert, P., De Schrijver, D., Van Lancker, W., Nollet, V., Avasare, P., Clerckx, T., Verdicchio, F., Christiaens, M., Schelkens, P., Van de Walle, R., & Stroobandt, D. (2007). Scalable, wavelet-based video: from server to hardware-accelerated client. *IEEE Transactions on Multimedia*, 9(7), 1508–1519.

Ernst, M. D. (2003). Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis* (pp. 24–27). Portland, OR, USA.

Ernst, R., Henkel, J., & Benner, T. (1993). Hardware-software cosynthesis for microcontrollers. *IEEE Design & Test of Computers*, 10(4), 64–75.

Faes, P. (2008). *An Object-Oriented Shared Memory Environment for Reconfigurable Hardware*. PhD thesis, Ghent University.

Faes, P., Bertels, P., Van Campenhout, J., & Stroobandt, D. (2009). Using method interception for hardware/software codevelopment. *Springer Design Automation for Embedded Systems*, 13(4), 223–243.

Faes, P., Christiaens, M., Buytaert, D., & Stroobandt, D. (2005). FPGA-aware garbage collection in Java. In *Proceedings of the international conference on Field Programmable Logic and Applications (FPL)* (pp. 675–680). Tampere, Finland: IEEE.

Faes, P., Christiaens, M., & Stroobandt, D. (2004). Transparent communication between Java and reconfigurable hardware. In T. Gonzalez (Ed.), *Proceedings of the 16th IASTED International Conference Parallel and Distributed Computing and Systems* (pp. 380–385). Cambridge, MA, USA: ACTA Press.

Faes, P., Christiaens, M., & Stroobandt, D. (2007). Mobility of data in distributed hybrid computing systems. In *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS)* (pp. 386). Los Alamitos, CA, USA: IEEE Computer Society.

Faes, P., Minnaert, B., Christiaens, M., Bonnet, E., Saeys, Y., Stroobandt, D., & Van de Peer, Y. (2006). Scalable hardware accelerator for comparing DNA and protein sequences. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems* (pp. 33–38). Hong Kong: ACM.

Fisher, J. A. (1983). Very long instruction word architectures and the eli-512. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture* (pp. 140–150). New York, NY, USA: ACM.

Georges, A., Buytaert, D., Eeckhout, L., & De Bosschere, K. (2004). Method-level phase behavior in Java workloads. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)* (pp. 270–287). New York, NY, USA: ACM.

Gheorghita, S. V., Palkovic, M., Hamers, J., Vandecappelle, A., Mamagkakis, S., Basten, T., Eeckhout, L., Corporaal, H., Catthoor, F., Vandeputte, F., & Bosschere, K. D. (2009). System-scenario-based design of dynamic embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1), 1–45.

Griesemer, R. (1999). Generation of virtual machine code at startup. In *Proceedings of the 14th annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)* (pp. 1–4).

Gupta, R. K. & De Micheli, G. (1993). Hardware-software cosynthesis for digital systems. *IEEE Design & Test of Computers*, 10(3), 29–41.

Hakkennes, E. A. & Vassiliadis, S. (2001). Multimedia execution hardware accelerator. *Journal of VLSI signal processing systems for signal image and video technology*, 28(3), 221–234.

Halfhill, T. R. (2000). Transmeta breaks x86 low-power barrier. *Microprocessor Journal*.

Heirman, W. (2008). *Reconfigurable Optical Interconnection Networks for Shared-Memory Multiprocessor Architectures*. PhD thesis, Ghent University.

Heirman, W., Stroobandt, D., Miniskar, N. R., & Wuyts, R. (2009). PinComm: A communication profiler to optimize embedded resource usage. In *Proceedings of the 20th Annual Workshop on Circuits, Systems and Signal Processing* (pp.9̃). Veldhoven, The Netherlands: Technology Foundation STW.

Helaihel, R. & Olukotun, K. (1997). Java as a specification language for hardware/software systems. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)* (pp. 690–697). Washington, DC, USA: IEEE Computer Society.

Kay, A. C. (1996). The early history of Smalltalk. *History of programming languages II*, (pp. 511–598).

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., & Griswold, W. G. (2001). An overview of aspectj. In *ECOOP 2001: Proceedings of the 15th European Conference on Object-Oriented Programming* (pp. 327–353). London, United Kingdom: Springer-Verlag.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., & Irwin, J. (1997). Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*: Springer-Verlag.

Lang, T. G., O'Quin II, J. T., & Simpson, R. O. (1986). Threaded code interpreter for object code. *IBM Technical Disclosure Bulletin*, 28(10), 4238–4241.

Lattanzi, E., Gayasen, A., Kandemir, M., Vijaykrishnan, N., Benini, L., & Bogliolo, A. (2005). Improving Java performance using dynamic method migration on FPGAs. *International Journal of Embedded Systems*, 1(3), 228–236.

Lewis, J. P. & Neumann, U. (2003). *Performance of Java versus C++*. Technical report, Computer Graphics and Immersive Technology Lab, University of Southern California.

Li, K.-H. (1994). Reservoir-sampling algorithms of time complexity $O(n(1 + log(N/n)))$. *ACM Transactions on Mathematical Software*, 20(4), 481–493.

Lindholm, T. & Yellin, F. (1999). *Java Virtual Machine Specification (second edition)*. Addison-Wesley.

Lysecky, R., Stitt, G., & Vahid, F. (2006). WARP processors. *Transactions on Design Automation of Electronic Systems*, 11(3), 659–681.

Maddimsetty, R. P., Buhler, J., Chamberlain, R. D., Franklin, M. A., & Harris, B. (2006). Accelerator design for protein sequence HMM search. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing* (pp. 288–296). New York, NY, USA: ACM.

Maebe, J., Buytaert, D., Eeckhout, L., & De Bosschere, K. (2006a). Javana: a system for building customized java program analysis tools. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (pp. 153–168). New York, NY, USA: ACM.

Maebe, J., Buytaert, D., Eeckhout, L., & De Bosschere, K. (2006b). Javana: a system for building customized Java program analysis tools. *ACM SIGPLAN Notices*, 41(10), 153–168.

Maebe, J., Ronsse, M., & De Bosschere, K. (2002). Diota: Dynamic Instrumentation, Optimization and Transformation of Applications. In *Compendium of Workshops and Tutorials Held in conjunction with the International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

May, C. (1987). Mimic: a fast System/370 simulator. In *SIGPLAN '87: Papers of the Symposium on Interpreters and interpretive techniques* (pp. 1–13). New York, NY, USA: ACM.

McGhan, H. & O'Connor, M. (1998). PicoJava: A direct execution engine for Java bytecode. *Computer*, 31(10), 22–30.

Meeus, W. (2007). *Report on the process of the implementation of the Smith-Waterman algorithm on the FPGA architecture*. Technical report, FlexWare, SBO project 060068, Institute for the Promotion of Innovation through Science and Technology in Flanders.

Microsoft (2008). *Virtualization from Data Center to Desktop*. Technical report, Microsoft.

Minnaert, B. (2005). Ontwerp van een hardwareversneller voor de vergelijking van dna-sequenties. Master's thesis, Ghent University.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8), 144–116.

Nagpurkar, P. (2007). *Analysis, detection, and exploitation of phase behavior in Java programs*. PhD thesis, University of California at Santa Barbara, Santa Barbara, CA, USA.

Nethercote, N. & Seward, J. (2007). Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (pp. 89–100). New York, NY, USA: ACM.

Nori, K. V., Ammann, U., Jensen, K., & Nageli, H. (1975). *The Pascal P compiler implementation notes*. Technical report, Eidgenössiche Technische Hochschule, Zürich.

Overgaard, M. (1980). Ucsd pascal™: a portable software environment for small computers. In *AFIPS '80: Proceedings of the May 19-22, 1980, national computer conference* (pp. 747–754). New York, NY, USA: ACM.

Porthouse, C. (2005). Jazelle for execution environments. ARM Whitepaper, available online.

Reinholtz, K. (2000). Java will be faster than C++. *SIGPLAN Notes*, 35(2), 25–28.

Schoeberl, M. (2008). A Java processor architecture for embedded real-time systems. *J. Syst. Archit.*, 54(1-2), 265–286.

Shaylor, N., Simon, D. N., & Bush, W. R. (2003). A Java virtual machine architecture for very small devices. In *LCTES '03: Proceedings*

of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems (pp. 34–41). New York, NY, USA: ACM.

Smith, J. E. & Nair, R. (2005). *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufman Publishers, Elsevier Inc.

Smith, T. F. & Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1), 195–197.

Standard Performance Evaluation Corporation (1998). Java Virtual Machine Benchmarks (SPECjvm1998).

Standard Performance Evaluation Corporation (2008). Java Virtual Machine Benchmarks (SPECjvm2008).

Stitt, G. & Vahid, F. (2005). A decompilation approach to partitioning software for microprocessor/fpga platforms. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe* (pp. 396–397). Washington, DC, USA: IEEE Computer Society.

Stitt, G. & Vahid, F. (2007). Binary synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3), 1–30.

Sun (2000). *J2ME Building Blocks for Mobile Devices*. Technical report, Sun Microsystems, 901 San Antonio Road, Palo Alto, CA 94303 USA.

Sun (2005a). *CLDC HotSpot Implementation Virtual Machine*. Technical report, Sun Microsystems, 4150 Network Circle, Santa Clara, CA 95054 USA.

Sun (2005b). *Sun powers Belgian e-government electronic identity card programme*. Technical report, Sun Microsystems, Guillemont Park, Minley Road, Blackwater, Camberley, Surrey, GU17 9QG, United Kingdom.

Vahid, F., Patel, R., & Stitt, G. (2001). Propagating constants past software to hardware peripherals in fixed-application embedded systems. *SIGARCH Comput. Archit. News*, 29(5), 25–30.

Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K., Kuzmanov, G., & Panainte, E. M. (2004). The MOLEN polymorphic processor. *IEEE Transactions on Computers*, 53(11), 1363–1375.

Vitter, J. S. (1985). Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1), 37–57.

VMWare (2008). *Building the Virtualized Enterprise with VMware Infrastructure*. Technical report, VMware, Inc.

Walpole, R. E. & Myers, R. H. (1993). *Probability and Statistics for Engineers and Scientists*. Prentice Hall.