

Code Generation
for
Transport Triggered Architectures

Code Generation for Transport Triggered Architectures

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof.ir. K.F. Wakker,
in het openbaar te verdedigen ten overstaan van een commissie,
door het College van Dekanen aangewezen,
op maandag 5 februari 1996 te 16.00 uur
door

Jan HOOGERBRUGGE

informatica ingenieur
geboren te Capelle aan de IJssel

Dit proefschrift is goedgekeurd door de promotor:
prof.dr.ir. A.J. van de Goor

Toegevoegd promotor:
dr. H. Corporaal

De leden van de promotiecommissie zijn:

Rector Magnificus	Technische Universiteit Delft
prof.dr.ir. A.J. van de Goor	Technische Universiteit Delft
dr. H. Corporaal	Technische Universiteit Delft
dr.ir. H.E. Bal	Vrije Universiteit Amsterdam
prof.dr.ir. J. van Katwijk	Technische Universiteit Delft
prof.dr.ir. M.J. Plasmeijer	Katholieke Universiteit Nijmegen
prof.dr.ir. H.J. Sips	Universiteit van Amsterdam
prof.dr. H.A.G. Wijshof	Rijks Universiteit Leiden

Cover design by Jan Hoogerbrugge and Roger van der Laan

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Hoogerbrugge, Jan

Code Generation for Transport Triggered Architectures /
Jan Hoogerbrugge. – [S.l. : s.n.]. – Ill.

Thesis Technische Universiteit Delft. – With ref. – With
summary in Dutch

ISBN 90-9009002-9

Subject headings: code generation / computer architecture

Copyright © 1996 Jan Hoogerbrugge

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without permission of the author.

*This dissertation is dedicated to
the loving memory of my father
Marinus Hoogerbrugge*

Contents

Acknowledgements	v
1 Introduction	1
1.1 Application Specific Processors	2
1.2 Transport Triggered Architectures	4
1.3 Motivation	5
1.4 Contributions	6
1.5 Thesis Overview	7
2 Transport Triggered Architectures	9
2.1 Instruction Level Parallel Processors	10
2.1.1 Superpipelining and Multiple Instruction Issue	10
2.1.2 Static and Dynamic Scheduling	13
2.1.3 Superscalars vs. VLIWs	15
2.1.4 Available ILP	18
2.2 Static ILP Exploitation	19
2.2.1 Scheduling Constraints	19
2.2.2 Scheduling Scopes	23
2.3 Transport Triggered Architectures	25
2.3.1 The Principle	25
2.3.2 An Example	28
2.3.3 Immediates	30
2.3.4 Control Flow	30
2.3.5 Conditional Execution	31
2.3.6 The Interconnection Network	32
2.3.7 Functional Units	34
2.4 Advantages and Disadvantages of TTAs	37
2.4.1 Implementation Advantages	37
2.4.2 Compiler Optimizations	38
2.4.3 Disadvantages	41

3	Basic Block Scheduling	43
3.1	Overview of the Compiler	43
3.1.1	The Front-End	44
3.1.2	The Back-End	46
3.1.3	Reading the Sequential Program and the Machine Description File	46
3.1.4	Transforming Irreducible CFGs into Reducible CFGs	47
3.1.5	Control Flow Analysis	47
3.1.6	Function Inlining and Loop Unrolling	47
3.1.7	Data Flow Analysis	50
3.1.8	Memory Reference Disambiguation	51
3.1.9	Register Allocation	54
3.2	The Basic Block Scheduler	57
3.2.1	List Scheduling for OTAs	58
3.2.2	List Scheduling for TTAs	60
3.2.3	Resource Assignment	61
3.2.4	Scheduling an Operation	63
3.2.5	TTA Specific Optimizations	66
4	Extended Basic Block Scheduling	69
4.1	Scheduling Scopes	69
4.2	Inter Basic Block Code Motion	72
4.3	Region Scheduling for OTAs	75
4.3.1	Importing Operations	75
4.3.2	The Operation Selection Heuristic	78
4.3.3	Importing a Compare Operation	79
4.3.4	Importing a Jump Operation	79
4.4	TTA Specific Issues	80
4.5	Discussion	83
5	Software Pipelining	87
5.1	Modulo Scheduling	88
5.1.1	Cyclic Data Dependency Graphs	89
5.1.2	Modulo Scheduling Constraints	90
5.1.3	Modulo Scheduling	90
5.2	Preprocessing Loops	94
5.2.1	If-conversion	94
5.2.2	Promotion	99
5.2.3	Delay Lines	100
5.2.4	Software Pipelining <i>While</i> Loops	102
5.3	TTA Specific Issues	103
6	Architecture and Compiler Evaluation	107
6.1	Methodology	107
6.2	Experiments	110

6.2.1	Speedup	110
6.2.2	Scheduling Scope	111
6.2.3	Scheduling Freedom	113
6.2.4	TTA Specific Optimizations	115
6.2.5	Register File Port Requirements	115
6.2.6	Partitioned Register Files	117
6.2.7	Multi-Way Branching and Guarding	119
6.2.8	Functional Unit Pipelining	121
6.2.9	Memory Reference Disambiguation	121
6.2.10	Multicasts	122
6.2.11	Partial Connectivity	123
6.2.12	Bypass Conflicts	125
6.2.13	Register Allocation	125
6.2.14	Conclusions	126
6.3	ILP Exploitation Bottlenecks	126
7	Design Space Exploration	133
7.1	The Design Process	134
7.1.1	Resource Optimization	136
7.1.2	Connectivity Optimization	139
7.2	Case Study: An ASP for MCCD	141
7.2.1	Special Functional Units	143
7.2.2	Resource Optimization	146
7.2.3	Connectivity Optimization	148
7.2.4	Miscellanea	149
7.2.5	Limitations	150
7.3	Related Work	151
8	Conclusions	155
8.1	Summary	155
8.2	Current Status of the Compiler	160
8.3	Perspective	160
8.4	Future Work	162
	Bibliography	165
A	Partial Loop Unrolling	179
A.1	Motivating Example	179
A.2	The Algorithm	180
A.3	Evaluation	182
	Samenvatting	183
	Curriculum Vitae	185

Acknowledgements

First of all I wish to thank prof. Ad van de Goor and Henk Corporaal for their guidance during the four years in which I performed the research described in this dissertation. I appreciate the freedom they gave me to do the research I am interested in.

Furthermore, I would like to thank my fellow AIOs and ex-AIOs within the MOVE project Robert Portier, Andy Verberne, Johan Janssen, Roger Jansen, Jeroen Hordijk, Paul van der Arend, Paul Stravers, and Reinoud Lamberts. Roommates Wilco van Hoogstraeten and Wiebe Cnossen for all the fun we had during our work. Prof. Stamatis Vassiliadis for his rich experience and view on computer architecture. Jaap Hoekstra for explaining me everything about department politics. System administrator Jean-Paul van der Jagt and his successor Tobias Nijweide for providing an excellent working computer environment. Students Theo Baan, Erwin Abrahamse, and Bas van Houte for their contributions to my research. Rogier Wolff for providing the MCCD application that I used in chapter 7.

Finally, I would like to thank my friends and family for supporting me and letting me think about other things than code generation for transport triggered architectures.

Jan Hoogerbrugge

Rotterdam, February 1996

Introduction

1

This dissertation describes the results of research performed within the MOVE Project at Delft University of Technology. The MOVE Project aims at designing *application specific processors* (ASPs) based on a new novel computer architecture paradigm called *transport triggered architectures* (TTAs). ASPs are processors designed especially for one particular application in order to improve their cost/performance. They are often embedded in all kinds of electronic systems. To reduce design costs, an ASP is usually designed according to a *template* architecture. Such a template should be flexible, scalable, and cost efficient. TTAs fulfill these requirements. TTAs are similar to very long instruction word (VLIW) processors in that they provide statically scheduled instruction level parallelism (ILP) in order to improve performance in a scalable and cost efficient way. The difference is that they are not programmed by instructions specifying multiple operations but by instructions specifying data transports. This improves flexibility, scalability, and cost efficiency, but also complicates the already complex compilation process. The main theme of this dissertation is to demonstrate that efficient compilation for TTAs is very well possible. This is done by developing a compiler for TTAs.

This chapter describes ASPs briefly in section 1.1, and TTAs in section 1.2. Section 1.3 gives our motivation for this research, section 1.4 enumerates the major contributions, and section 1.5 gives an overview of the remaining chapters of this thesis.

1.1 Application Specific Processors

One of the first decisions a designer of a processor based electronic system has to make is choosing between a standard off-the-shelf processor and an ASP, specially designed for the application in question. A standard processor is intended for a large class of applications and usually contains hardware that is not effectively used by the given application; and in addition, it misses hardware that could be very useful. For example, a standard processor might contain an expensive multiplier which is a waste of chip area and power consumption when the application seldom performs multiplications (e.g., less than 0.1% of all executed operations). As an example of hardware functionality that might be missing, consider an application that manipulates bit-oriented data. For such an application instructions such as ‘find first bit set’ and ‘count number of bits set’ are very welcome since they are easy to implement in hardware and may result in a significant speedup.

If the decision turns out in favor of an ASP the next question becomes: how should it be designed and implemented? A full custom design offers optimal flexibility, performance, chip area, and power consumption at the price of a long and expensive design process. This is only acceptable for high volume production. A method to reduce design costs is to build ASPs according to a *template*. For example, all ASPs have a four stage pipeline, have 32-bit wide words, and are big endian. An ASP is designed by instantiating the *architectural parameters* of the template. Examples of architectural parameters are the number of general purpose registers, cache sizes, the operation set, the amount of instruction level parallelism, and operation latencies. By designing ASPs according to a template, we reduce design costs by giving up some design freedom. This is similar to programming in a high level language instead of in assembly language or designing hardware in a hardware description language instead of designing at gate or layout level. Obviously, the usefulness of a templated ASP design system depends largely on the flexibility of the template and the efficiency of the tools for generating the processor and the code for the processor.

In our opinion a system for templated ASP design should consist of at least the following three components: (1) a processor generator, (2) a code generator, and (3) a design space explorer (see figure 1.1).

The processor generator

The processor generator is responsible for generating VLSI layout for the ASP according to the architectural parameter set. There are several ways to do this. One can use a silicon compiler that generates a layout based on a parameterized processor description. The parameters of the processor description are directly related to the architectural parameters. Another method is to use param-