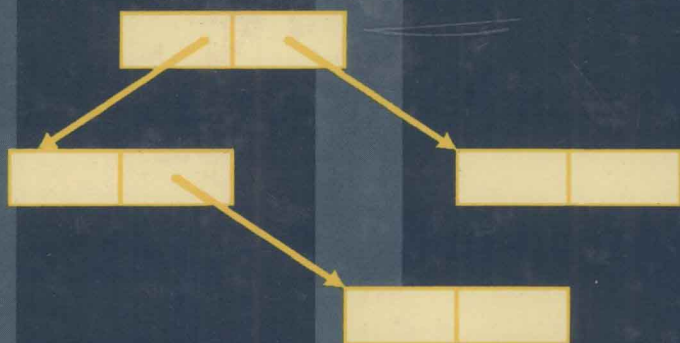




An Architecture for Combinator Graph Reduction



Philip John Koopman, Jr.

An Architecture for Combinator Graph Reduction

Philip John Koopman, Jr.

*Harris Semiconductor
Melbourne, Florida*



ACADEMIC PRESS, INC.
Harcourt Brace Jovanovich, Publishers

Boston San Diego New York
London Sydney Tokyo Toronto

This book is printed on acid-free paper. ∞

Copyright © 1990 by Academic Press, Inc.
All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system, without permission in writing from the publisher.

ACADEMIC PRESS, INC.
1250 Sixth Avenue, San Diego, CA 92101

United Kingdom Edition published by
ACADEMIC PRESS LIMITED
24-28 Oval Road, London NW1 7DX

Library of Congress Cataloging-in-Publication Data

Koopman, Phil, date.

An architecture for combinator graph reduction / Philip John

Koopman, Jr.

p. cm.

Includes bibliographical references and index.

ISBN 0-12-419240-8 (alk. paper)

1. Computer architecture. 2. Functional programming (Computer science) 3. Graph grammars. I. Title.

QA76.9.A33K66 1990

004.2'2'015115—dc20

90-38168
CIP

Printed in the United States of America
90 91 92 93 9 8 7 6 5 4 3 2 1

**An
Architecture
for
Combinator Graph
Reduction**

To my parents

Preface

This book is based on my Ph.D. thesis for the Electrical and Computer Engineering Department at Carnegie Mellon University. It is the result of a computer engineer's journey into the realm of Computer Science theory and programming language implementation. The point of view taken is that of an engineer, and focuses on how to solve a problem (in this case, fast combinator reduction) efficiently. The book is split into two major areas. The first area is the development of the TIGRE graph reducer, along with performance measurements on a variety of machines. The second area is an architectural analysis of TIGRE's behavior.

This research would not have been possible without the support of two faculty members to cover the two areas of the research. Dan Siewiorek has helped me mature as an architect, provided guidance for the engineering half of the thesis, and was supportive when I decided to pursue an unusual (for an engineer) research direction. Peter Lee introduced me to combinator reduction, and provided encouragement, software support, and expert editing assistance. The other members of my thesis committee, Rob Rutenbar and Tom Hand, also helped guide the course of the research. John Dorband at NASA/Goddard gave me the funding support and freedom I needed to perform the research (which was funded by NASA/Goddard under contract NAG-5-1046).

During the quest for my degree, many people have helped in ways large and small. Some of the main contributors are: my wife, Mary, for her support during the stressful times, and tolerance of late nights/early mornings; Glen Haydon, who inspired my interest in threaded architectural techniques and provided helpful insight into the Ph.D. process; and Dom Carlino, who has provided sage advice and encouragement.

Contents

List of Tablesxi
List of Illustrationsxiii
Preface	xv
1. Introduction	1
1.1. OVERVIEW OF THE PROBLEM AREA	1
1.2. ORGANIZATION OF THIS BOOK	3
2. Background	5
2.1. PROBLEM DEFINITION	5
2.1.1. Lazy Functional Programming	5
2.1.2. Closure Reduction and Graph Reduction	6
2.1.3. Performance Inefficiencies	8
2.2. PREVIOUS RESEARCH	9
2.2.1. Miranda	9
2.2.2. Hyperlazy Evaluation	10
2.2.3. The G-Machine	10
2.2.4. TIM	11
2.2.5. NORMA	12
2.2.6. The Combinatorgraph Reducer	12
2.2.7. Analysis and Summary	13
2.3. APPROACH OF THIS RESEARCH	14
3. Development of the TIGRE Method	15
3.1. THE CONVENTIONAL GRAPH REDUCTION METHOD	15
3.2. FAST INTERPRETIVE EXECUTION OF GRAPHS	17
3.3. DIRECT EXECUTION OF GRAPHS	19
4. Implementation of the TIGRE Machine	25
4.1. THE TIGRE ABSTRACT MACHINE	25
4.1.1. Hardware Definition	25

4.1.2. TIGRE Assembly Language	26
4.1.3. A TIGRE Compiler	30
4.2. MAPPING OF TIGRE ONTO VARIOUS EXECUTION MODELS	31
4.2.1. Mapping of TIGRE Onto the C Execution Model	31
4.2.2. Mapping of TIGRE Assembly Language Onto a VAX	33
4.2.3. Mapping of TIGRE Assembly Language Onto a MIPS R2000	35
4.2.4. Translation to Other Architectures	36
4.3. TIGRE ASSEMBLER DEFINITIONS OF COMBINATORS	37
4.3.1. Non-Strict Combinators	37
4.3.1.1. 1-Projection Combinators	37
4.3.1.2. Simple Graph Rewriting Combinators	38
4.3.2. Strict Combinators	38
4.3.2.1. Totally Strict Combinators	39
4.3.2.2. Partially Strict Combinators	40
4.3.3. List Manipulation Combinators	41
4.3.4. Supercombinators	43
4.4. SOFTWARE SUPPORT	45
4.4.1. Garbage Collection	45
4.4.2. Other Software Support	47
5. TIGRE Performance	49
5.1. TIGRE PERFORMANCE ON VARIOUS PLATFORMS	49
5.1.1. TIGRE Performance for the Turner Set	51
5.1.2. TIGRE Performance for Supercombinator Compilation	53
5.2. COMPARISONS WITH OTHER METHODS	54
5.2.1. Miranda	54
5.2.2. Hyperlazy Evaluation	55
5.2.3. The G-Machine	55
5.2.4. TIM	56
5.2.5. NORMA	56
5.3. TIGRE VERSUS OTHER LANGUAGES	57
5.3.1. Non-Lazy Language: T Version 3.0	57
5.3.2. Imperative Language: MIPS R2000 C Compiler	58
5.4. ANALYSIS OF PERFORMANCE	59
6. Architectural Metrics	63
6.1. CACHE BEHAVIOR	63
6.1.1. Exhaustive Search of the Cache Design Space	63
6.1.2. Parametric Analysis	69

Contents	ix
6.1.2.1. Write Allocation	70
6.1.2.2. Cache Size	72
6.1.2.3. Block Size	73
6.1.2.4. Associativity	75
6.1.2.5. Replacement Policy	76
6.1.2.6. Write-Through Policy	76
6.1.3. A Desirable Cache Strategy	77
6.2. PERFORMANCE OF REAL HARDWARE	78
6.2.1. Simulation Results for a DECstation 3100	78
6.2.2. Comparison with Actual Measurements	82
6.3. DYNAMIC PROGRAM BEHAVIOR	84
6.3.1. Heap Memory Use	84
6.3.2. Stack Memory Use	86
7. The Potential of Special-Purpose Hardware	89
7.1. DECSTATION 3100 AS A BASELINE	89
7.2. IMPROVEMENTS IN CACHE MANAGEMENT	90
7.2.1. Copy-Back Cache	90
7.2.2. Increased Block Size	90
7.2.3. Prefetch on Read Misses	91
7.3. IMPROVEMENTS IN CPU ARCHITECTURE	92
7.3.1. Stack Unwinding Support	92
7.3.2. Stack Access Support	93
7.3.3. Doubleword Store	94
7.4. PERFORMANCE IMPROVEMENT POSSIBILITIES	94
8. Conclusions	97
8.1. CONTRIBUTIONS OF THIS RESEARCH	97
8.2. AREAS FOR FURTHER RESEARCH	98
Appendix A. A Tutorial on Combinator Graph Reduction	.101
A.1. FUNCTIONAL PROGRAMS	.101
A.2. MAPPING FUNCTIONAL PROGRAMS TO LAMBDA CALCULUS	.102
A.3. MAPPING LAMBDA CALCULUS TO SK- COMBINATORS	.103
A.4. MAPPING SK-COMBINATOR EXPRESSIONS ONTO A GRAPH	.105
A.5. THE TURNER SET OF COMBINATORS	.117
A.6. SUPERCOMBINATORS	.120
A.7. INHERENT PARALLELISM IN COMBINATOR GRAPHS	.121

Appendix B. Selected TIGRE Program Listings	123
B.1. REDUCE.H	123
B.2. KERNEL.C	126
B.3. TIGRE.S	129
B.4. MIPS.S	136
B.5. HEAP.H	144
B.6. HEAP.C	145
 References	 149
 Index	 153

List of Tables

Table 5-1. TIGRE performance on a variety of platforms	50
Table 5-2. Benchmark listings	52
Table 5-3. TIGRE speedups using supercombinator compilation . . .	53
Table 5-4. Performance of TIGRE versus Miranda	54
Table 5-5. Performance of TIGRE versus Hyperlazy evaluation . . .	55
Table 5-6. Performance of TIGRE versus TIM	56
Table 5-7. Performance of TIGRE versus NORMA	56
Table 5-8. TIGRE performance compared to T3.0	57
Table 5-9. TIGRE performance compared to C	58
Table 5-10. C program listings for comparison with TIGRE	60
Table 6-1. Cache performance simulation results for TIGRE on a MIPS R2000	65
Table 6-2. Baseline for parametric analysis	69
Table 6-3. TIGRE performance with varying cache write allocation strategy	70
Table 6-4. TIGRE performance with varying cache associativity . .	76
Table 6-5. TIGRE performance with varying cache replacement policies	76
Table 6-6. TIGRE performance with varying cache write-through strategy	77
Table 6-7. Baseline for DECstation 3100 analysis	79
Table 6-8. Performance with varying cache write allocation strategy	79
Table 6-9. Cache performance with varying cache associativity . . .	80
Table 6-10. Cache performance with varying cache write-through strategy	82
Table 6-11. TIGRE use of heap memory	84
Table 6-12. TIGRE use of stack memory for fib	86
Table 7-1. Summary of TIGRE DECstation 3100 performance characteristics	90
Table 7-2. Summary of possible performance improvements	95

Table A-1. Non-strict members of the Turner combinator set . . .	117
Table A-2. Turner Set optimizations	119

List of Illustrations

Figure 2-1. Evolution of lazy functional program implementation techniques	13
Figure 3-1. Basic structure of a node	15
Figure 3-2. Example for expression $((+ 11) 22)$	16
Figure 3-3. Example using indirection nodes for constants	17
Figure 3-4. Example using LIT nodes instead of indirection nodes for constants	18
Figure 3-5. Example with tag fields removed	19
Figure 3-6. An example TIGRE program graph, emphasizing the left spine	20
Figure 3-7. A TIGRE program graph with only subroutine call pointers	21
Figure 3-8. VAX assembly language implementation of a TIGRE expression	22
Figure 4-1. A block diagram of the TIGRE abstract machine	26
Figure 4-2. The S' combinator	29
Figure 4-3. Mapping of the TIGRE abstract machine onto C	31
Figure 4-4. Mapping of the TIGRE abstract machine onto a VAX 8800	34
Figure 4-5. Mapping of the TIGRE abstract machine onto a MIPS R2000	35
Figure 4-6. The IF combinator	41
Figure 4-7. The P combinator	42
Figure 4-8. The U combinator	43
Figure 4-9. The \$FIB supercombinator	44
Figure 6-1. TIGRE performance with varying cache size	72
Figure 6-2. TIGRE performance with varying cache block size	74
Figure 6-3. Cache performance with varying cache size	80
Figure 6-4. Performance with varying cache block size	81

Figure A-1. The function and argument structure of a node106
Figure A-2. A function argument pair106
Figure A-3. A shared subtree107
Figure A-4. Graph to add 11 and 22107
Figure A-5. Operation of the I combinator107
Figure A-6. Operation of the K combinator108
Figure A-7. Operation of the S combinator109
Figure A-8. An addition example110
Figure A-9. Doubling function110
Figure A-10. Doubling function applied to argument111
Figure A-11. Reduction step 1112
Figure A-12. Reduction step 2112
Figure A-13. Reduction step 3113
Figure A-14. Reduction step 4113
Figure A-15. Reduction step 5114
Figure A-16. Reduction step 6114
Figure A-17. Reduction step 7115
Figure A-18. Reduction step 8115
Figure A-19. Reduction step 9115
Figure A-20. Reduction step 10116
Figure A-21. Reduction step 11116
Figure A-22. Operation of the B combinator118
Figure A-23. Operation of the C combinator119

Chapter 1

Introduction

This chapter contains both an overview of the problem area to be discussed and an overview of the structure of the rest of the book.

1.1. OVERVIEW OF THE PROBLEM AREA

Functional programming provides a new way of writing programs and a new way of thinking about problem solving (Backus 1978). A specific advantage of functional programs is the fact that they are easy to reason about, since they can be viewed as mathematical specifications of algorithms, and are therefore amenable to automatic verification techniques. Also, there is a belief in some circles that functional programs are easier to write than other programs. This is because functional programming languages provide powerful higher-order composition mechanisms which are not found in conventional imperative languages such as C. Furthermore, the combination of these mentioned qualities can lead to reliable software systems (Hughes 1984). Although the foundations of functional programming have been known for some time (Curry & Feys 1968, Landin 1966, Reynolds 1972), most of what we know about the field has been discovered in the last ten years. Therefore, the potential benefits of using functional programming techniques are still largely unexplored.

Lazy evaluation (Henderson & Morris 1976, Freedman & Wise 1976) of functional programs allows the use of powerful programming structures such as implicit coroutining and infinitely long lists. Unfortunately, the power and flexibility of lazy evaluation has, in the past, been associated with extreme inefficiency when executing programs. It is common for programs to be 100 times slower in a lazy functional language than in an imperative language such as C.* Because programs written in these languages execute so slowly, it is difficult to build a large software base to gain experience in using the languages. And, without a large software and user base, it will be difficult to gain insights on the

* Actual comparisons will be given in a later chapter.

appropriateness of lazy functional programming languages for solving real problems.

One important evaluation strategy for lazy functional programming languages is *graph reduction*. Graph reduction involves converting the program to a lambda calculus expression (Barendregt 1981), and then to a graph data structure. One method for implementing the graph data structure is to translate the program to combinators (Curry & Feys 1968). A key feature of this method is that all variables are abstracted from the program. The program is represented as a computation graph, with instances of variables replaced by pointers to subgraphs which compute values. Graphs are evaluated by repeatedly applying graph transformations until the graph is irreducible. The irreducible final graph is the result of the computation. In this scheme, the rewriting of the graph data structure, also called *combinator graph reduction*, is the method used to execute the program.

A great allure of combinator graph reduction is that it may provide an automatic approach to parallel computation, since the available parallelism of a program compiled to a graph is directly represented by the graph structure (Peyton Jones 1987). Such parallelism tends to be fine-grained, where each quantum of work available is small in size. Overhead in managing resources and task scheduling can quickly dominate the performance of a fine-grained parallelism system, so it is important to find a scheme in which overhead is kept low to achieve reasonable speedups.

Traditionally, it has been assumed that advanced programming languages (and in particular functional programming languages) require radically different, non-vonNeumann architectures for efficient execution. This book explores mapping functional programming languages onto conventional architectures using a combination of techniques from the fields of computer architecture and implementation of advanced programming languages.

The tools of the computer architect shed new light on the behavior of this special class of programs. *The results shown here suggest that the advanced programming languages being explored by computer scientists do not adhere to the normal expectations of computer architects*, and may eventually force a reevaluation of architectural tradeoffs in system design. An important point of the findings presented here is that the combination of architectural features required for efficiency may be relatively inexpensive, yet omitted from even recent machines because of relative unimportance for conventional programming language execution.

1.2. ORGANIZATION OF THIS BOOK

The book examines existing methods of evaluating lazy functional programs using combinator reduction techniques, implementation and characterization of a means for accomplishing graph reduction on uniprocessors, and analysis of the potential for special-purpose hardware implementations.

Chapter 2 provides a background on functional programming languages and existing implementation technology. The reader who is not familiar with the field may wish to read Appendix A, which is a tutorial on combinator graph reduction. Chapter 2 also contains a summary of important previous work on the combinator reduction approach to evaluating lazy functional programming languages.

Chapter 3 describes the TIGRE methodology for implementing combinator graph reduction. The description is in the form of a progression of techniques which are added to a graph reduction mechanism based on previously used methods. The general flow of the incremental improvements starts with conventional graph reduction methods, moves on to a fast interpretation scheme for combinator graphs, refines the method to a direct execution scheme for combinator graphs, and then discusses supercombinator compilation methods for improved performance.

Chapter 4 describes the TIGRE abstract machine, which is used to implement the graph reduction methodology described in Chapter 3. TIGRE may be described in terms of an abstract architecture and abstract assembly languages. These abstract definitions have been mapped efficiently onto real languages and architectures, including machine-independent C code and assembly language implementations for the VAX family and the MIPS R2000 processor.

Chapter 5 gives the results of performance measurements of TIGRE on a variety of platforms. These results are compared with available results for other combinator reduction strategies and against the performance of imperative languages.

Chapter 6 discusses architectural metrics for TIGRE executing on the MIPS R2000 processor. The architectural metrics include a simulation of cache behavior, combinator execution frequency, and various dynamic metrics such as heap allocation statistics.

Chapter 7 explores the potential for special-purpose hardware to yield further speed improvements. In order to maintain some basis in reality, modifications to the MIPS R2000 architecture as implemented in the DECstation 3100 platform are proposed, along with predicted speed improvements.