

09 AH
A3741



COMPILATION OF PARALLEL APPLICATIONS VIA
AUTOMATED TRANSFORMATION OF BMF
PROGRAMS

By
Brad Alexander
March 28, 2006

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SCHOOL OF COMPUTER SCIENCE
UNIVERSITY OF ADELAIDE

March 28, 2006

Preface

Transformation is crucial to any program improvement process. Highly transformable notations pave the way for the application of deep and pervasive program improvement techniques. Functional programming languages are more amenable to transformation than their more traditional imperative counterparts. Moreover, functional programs specify only true dependencies between values, making improvements that reveal and exploit parallelism much easier. Some functional programming notations are more transformable than others. Bird-Meertens-Formalism (BMF) is a functional notation that evolved as a medium for transformational program development. A substantial, and growing, body of work has created novel tools and techniques for the development of both sequential and parallel applications in BMF.

Formal program development is at its most useful when it can be carried out automatically. Point-Free BMF, where programs are expressed purely as functions glued together with higher-order operators, provides enhanced scope for automated development because many useful transformations can be expressed as easily applied re-write rules. Moreover, realistic sequential and parallel static cost models can be attached to BMF code so the relative merits of applying various transformations can be accurately assessed.

In spite of its potential merits there has been little work that has utilised point-free BMF, in a pervasive manner, as a medium for automated program improvement. This report describes a prototype implementation that maps a simple point-wise functional language into point-free BMF which is then optimised and parallelised by the automated application of, mostly simple, rewrite rules in a fine-grained and systematic manner. The implementation is shown to be successful in improving the efficiency of BMF code and extracting speedup in a parallel context. The report provides details of the techniques applied to the problem and shows, by experiment

and analysis, how reductions in high data-transport costs are achieved. We also describe techniques used to keep the optimisation task tractable by alleviating the hazard of case-explosion.

The report is structured according to the stages of the compilation process, with related work described at the end of each chapter. We conclude with our main finding, namely, the demonstrated feasibility and effectiveness of optimisation and parallelisation of BMF programs via the automated application of transformation rules. We also restate techniques useful in achieving this end, the most important of which is the substantial use of normalisation during the optimisation process to prepare code for the application of desirable transformations. We also present a brief summary of potential future work including the introduction of more formally described interfaces to some of the transformative rule-sets, the automatic production of annotated proofs and a facility to display static estimates of the efficiency code during transformation.

Contents

Preface	iii
Declaration	v
Acknowledgments	vi
1 Introduction	1
1.1 Distributed Parallel Computing	1
1.2 Implementing a model	2
1.3 This work	4
1.3.1 The role of BMF in this project	7
1.4 Related work	9
1.5 Preview	12
2 Adl	13
2.1 The Adl language	13
2.1.1 Background	13
2.1.2 Main features	14
2.1.3 Program layout	15
2.1.4 Declaration sequences	16
2.1.5 Types	17
2.1.6 Conditional expressions and iteration	22
2.2 Future enhancements	24
2.3 Summary	25
3 Bird-Meertens Formalism	26
3.1 Introduction to Bird-Meertens Formalism	26

3.1.1	Some introductory transformations	27
3.1.2	General Transformations and Program Structures	28
3.1.3	Functions working with list-like data	30
3.1.4	Homomorphisms	33
3.2	The role of BMF in the Adl project	36
3.3	Summary	37
4	Adl to BMF translation	39
4.1	What the translator does	39
4.2	Overview of translation	41
4.3	The rules of translation	45
4.3.1	The source and target syntax	45
4.3.2	Interpreting the rules	52
4.3.3	The rules	56
4.4	Performance	75
4.4.1	Measuring sequential performance	75
4.4.2	Examples of performance of translator code	79
4.5	Related Work	87
4.6	Conclusions	88
5	Data Movement Optimisation	90
5.1	Chapter outline	90
5.1.1	How the optimiser is described	90
5.1.2	Stages of optimisation	90
5.1.3	Common elements and separate elements	92
5.2	The common strategy	92
5.3	Common tactics	96
5.3.1	Making it easy to ignore uninteresting code	97
5.3.2	Keeping code predictable	99
5.4	Vector optimisation	106
5.4.1	Strategy for vector optimisation	106
5.4.2	Vector Optimisation of Map	112
5.4.3	Optimising non-map functions	137
5.5	Tuple optimisation	143
5.5.1	The filter expression	144

5.5.2	The rules of tuple optimisation	146
5.6	Findings	153
5.6.1	Performance	153
5.6.2	Findings related to building the optimiser definition	167
5.7	Related work	173
5.7.1	Optimisations related by goal	173
5.7.2	Optimisations related by technique	178
5.8	Summary and Future Work	182
5.8.1	Future work	183
6	Parallelisation and Targetting	184
6.1	Preview	184
6.1.1	Design choices for parallelisation	185
6.1.2	Design choices for code generation	187
6.1.3	Summary of our design choices	188
6.2	Parallelisation	188
6.2.1	Overview of the parallelisation process	188
6.2.2	The rules of parallelisation	195
6.3	Code generation	216
6.3.1	Preliminary implementation	216
6.3.2	Memory Management Costs	217
6.4	Results	219
6.4.1	Methodology	220
6.4.2	The simulator	220
6.4.3	Experiments	221
6.4.4	Experiment 1: <code>map_map_addconst.Ad1</code>	223
6.4.5	Experiment 2: simple reductions	229
6.4.6	Experiment 4: <code>finite_diff_iter.Ad1</code>	238
6.4.7	Experiment 5: <code>remote.Ad1</code>	245
6.5	Related work	249
6.6	Conclusions and Future work	251
7	Conclusions and Future work	253
7.1	Primary findings	253
7.2	Secondary findings	254

7.2.1	Incremental transformation offers advantages	254
7.2.2	Catch-all rules, if overused can lead to poor performance	255
7.2.3	The importance of normalisation	255
7.2.4	Complex transformation rules are best avoided	256
7.2.5	Observations relating to Centaur	256
7.3	Future work	257
A	Glossary	259
B	The transpose function	268
B.1	Implementation of parametric transpose	268
B.1.1	Transposing non-rectangular vectors	269
B.2	Source code of an implementation	271
B.3	Source code for <code>transpose.scm</code>	272
B.4	Test results	283
C	Sorted and Sortable indexing functions	286
C.1	Assumptions	286
C.2	The set of valid address functions	287
C.3	A partial ordering on address functions	288
C.4	Extracting address functions from index-generators	289
C.4.1	Condensing the output of <code>ExtractAddr</code>	289
C.5	The property of being sorted	290
C.6	From sortable to sorted	290
D	The time-space model	292
D.1	Syntax	292
D.2	Semantics	295
D.2.1	The trace generator	296
D.2.2	Interpreter rules	296
D.2.3	Auxiliary function definitions	297
D.3	Interpreter Rules	299
D.3.1	Scalar Functions	301
D.3.2	Length	301
D.3.3	Vector Indexing	301

D.3.4	The identity function	302
D.3.5	Function Composition	303
D.3.6	Map	303
D.3.7	Distl	304
D.3.8	Reduce	305
D.3.9	Scan	306
D.3.10	Alltup functions	307
D.3.11	Allvec functions	308
D.3.12	Select	309
D.3.13	Zip	309
D.3.14	If	310
D.3.15	While	311
D.3.16	Priffle	311
D.3.17	Mask	312
D.3.18	Transpose	313
D.4	Extracting the trace	314
E	Nested Parallelism	316
E.1	Methodology	316
E.2	Using nesting to extend block partitioning to more than one dimension	319
E.2.1	Nested block partitioning in Adl	321
E.2.2	Nesting other structures	324
E.2.3	Simpler block-partitioning	325
E.3	Using nested parallelism to exploit parallelism over static irregular structures	325
E.3.1	A simple schema	326
E.3.2	Nesting other functions in an irregular context	329
E.4	Using nested parallelism to exploit parallelism over evolving irregular structures.	329
E.4.1	Using work migration to implement dynamically evolving irregular nested parallelism	330
E.4.2	Using flattening to implement dynamically evolving irregular nested parallelism	331
E.5	Conclusion	332

F	Costs of malloc and free	333
F.1	Factors	333
F.2	Methodology	333
F.3	Experimental results	336
F.3.1	Raw instruction execution speeds	336
F.3.2	The cost of malloc	336
F.3.3	The cost of free	343
F.4	Conclusion	346