Design and Optimisation of Scientific Programs in a Categorical Language

Thomas James Ashby

Doctor of Philosophy

Institute of Computing Systems Architecture
School of Physics and School of Informatics
University of Edinburgh

2005

Abstract

This thesis presents an investigation into the use of advanced computer languages for scientific computing, an examination of performance issues that arise from using such languages for such a task, and a step toward achieving portable performance from compilers by attacking these problems in a way that compensates for the complexity of and differences between modern computer architectures.

The language employed is Aldor, a functional language from computer algebra, and the scientific computing area is a subset of the family of iterative linear equation solvers applied to sparse systems. The linear equation solvers that are considered have much common structure, and this is factored out and represented explicitly in the language as a framework, by means of *categories* and *domains*. The flexibility introduced by decomposing the algorithms and the objects they act on into separate modules has a strong performance impact due to its negative effect on temporal locality. This necessitates breaking the barriers between modules to perform *cross-component optimisation*. In this instance the task reduces to one of *collective loop fusion* and *array contraction*. Traditional approaches to this problem rely on static heuristics and simplified machine models that do not deal well with the complex trade-offs involved in targeting modern computer architectures. To rectify this we develop a technique called *iterative collective loop fusion* that empirically evaluates different candidate transformations in order to select the best available. We apply our technique to programs derived from the iterative solver framework to demonstrate its effectiveness, and compare it against other techniques for collective loop fusion from the literature, and more traditional approaches such as using Fortran, C and/or high-performance library routines.

The use of a high-level categorical language such as Aldor brings important benefits in terms of elegance of expression, comprehensibility, and code reuse. Iterative collective loop fusion outperforms the other collective loop fusion techniques. Applying it to the iterative solver framework gives programs with performance that is comparable with the traditional approaches.

Acknowledgements

Firstly I would like to thank my two long-suffering supervisors Mike O'Boyle (computer science) and Tony Kennedy (physics), for their motivation, academic guidance, financial support and patience above and beyond the call of duty. Doing interdisciplinary research is by no means easy, but I hope that ultimately the venture was worth it for all concerned. I would also like to thank anyone who helped me in my work or commented on the thesis to help me improve it, including my viva panel Mark Bull (internal – EPCC) and Paul Kelly (external – Imperial College). On a more general note, my gratitude goes out to the administrative and support staff at the University of Edinburgh, and the people who built all the tools that I have relied on to do my work. This includes the creators of the dictation packages ViaVoice (IBM) and Dragon Naturally Speaking (virtually all the text in this document has been dictated), which, as well as being useful, have also helped to keep me amused with recognition errors, some of which were uncannily perceptive. My favourites include "much hilarity" for "modularity", "Al Gore/Algol" for "Aldor", and "awful banality" for "orthogonality".

Secondly, I would like to thank my friends – you know who you are. I especially need to mention two whose companionship along the torturous route to a doctorate meant a great deal to me – Henrik, for providing me with a regular dose of reality (*there is no spork*), and Björn, for coping with sharing an office with me as well as being a friend.

Lastly, there is my family and my partner, without whom I certainly would not have finished. Their support was always given without question, even when they had far, far greater burdens to carry than my own. Words fail me now, as they always will.

Dedicated to every unwitting innocent who has ever had the misfortune to ask me:
"So, what is it that you actually **do** *then?"*

Table of Contents

1 Introduction 1

1.1 Computational Science Domain 2

1.2 Language 3

1.3 Compiler Optimisations 4

1.4 Linear Systems 6

1.5 Thesis Outline 7

1.6 Contributions 9

2 Aldor 11

2.1 Fundamentals of the Language 11

2.1.1 Domains and categories 11

2.1.2 General features 14

2.1.3 The core language and the abstract machine 15

2.1.4 Basic libraries 17

2.1.5 Storage model 18

2.1.6 Purity and overloading 18

2.2 Abstract Machine and Compilation Model 19

2.2.1 Libraries and whole program optimisation 20

2.2.2 FOAM types and variables 21

2.2.3 Uniform representation rule 21

2.2.4 Memory model 22

2.2.5 Aldor domains and generators in FOAM 22

2.3 Compiler Implementation 23

2.3.1 Pre-existing optimisations 24
2.4 Summary 27

3 The Iterative Solvers **29**
3.1 Notation 29
3.2 Overview 30
3.2.1 The form of the problem 30
3.2.2 Krylov subspaces 30
3.2.3 Halting condition 31
3.2.4 Orthogonality conditions 31
3.2.5 Reduced (projected) system as interface 32
3.2.6 Operator structure 33
3.3 Generating Krylov Subspaces 34
3.3.1 The Arnoldi relation 34
3.3.2 Long and short recurrences 35
3.3.3 The two-sided Lanczos method 36
3.4 Orthogonality Conditions and Projected Systems 38
3.4.1 Orthogonality conditions and orthogonal Krylov bases 38
3.4.2 Orthogonality conditions and non-orthogonal Krylov bases . . 41
3.4.3 Orthogonality conditions and breakdowns 41
3.5 Solving the Projected System and Recovering the Solution 42
3.5.1 Search vectors 43
3.6 The Common Algorithms 46
3.6.1 Initial guess 46
3.6.2 Calculating the recurrence residual 46
3.6.3 Putting it all together 48
3.6.4 Preconditioning 49
3.7 Summary 50

4 Functional and Algebraic Language Optimisation **51**
4.1 Compilation of Functional Languages 51
4.1.1 Fine-grained function composition and recursion 52

4.1.2 Higher order control flow analysis 52
4.1.3 Polymorphism, boxing and modules 54
4.1.4 Arrays 56
4.1.5 Pure languages and the management of state 57
4.1.6 Compiler implementation 58
4.1.7 Fusion and loop restructuring in functional languages 59
4.2 Compilation of Numerical Computer Algebra Systems 60
4.3 Summary 61

5 Algorithm Framework **63**
5.1 Category Hierarchy 63
5.1.1 Linear algebra categories 67
5.1.2 Problem specific categories 71
5.2 Domain Implementation 76
5.2.1 Index functions, recurrences, and infinite sequences 76
5.2.2 Krylov space recurrence 77
5.2.3 Matrix of basis vectors 80
5.2.4 Tridiagonal matrix of recurrence coefficients 80
5.2.5 *QR* decomposition 81
5.2.6 Banded upper triangular factor 82
5.2.7 Lazy vector domain 82
5.2.8 Search vector recurrence 82
5.2.9 Matrix of search vectors 83
5.2.10 Glue code 83
5.3 Evaluation of Framework Design 85
5.3.1 Remaining issues 87
5.4 Summary 89

6 Linear Systems **91**
6.1 Partial Differential Equations and Their Discretisation 91
6.1.1 Grid numbering, matrix layout and stencils 92
6.2 Example Operators 94

6.2.1 The Laplacian-like simple operators 94
6.2.2 The Wilson-Dirac operator 95
6.3 Domain Implementation 98
6.3.1 The scalar domains 98
6.3.2 The simple stencil operator and vector domains 100
6.3.3 Subdomains for the Wilson-Dirac problem 101
6.3.4 Aggregate structures of subdomains 102
6.3.5 The Wilson-Dirac Operator and Vector Domains 105
6.4 Design Issues 106
6.4.1 Boundary conditions and indexing 106
6.5 Summary 107

7 Optimisation across Components 109
7.1 Basic Terminology and Formalisms 110
7.1.1 Loops 110
7.1.2 Dependencies between loop iterations 111
7.1.3 Statements and their dependencies 111
7.1.4 Dependence testing 113
7.1.5 Temporal Locality 113
7.2 Loop Fusion and Array Contraction 113
7.2.1 Loop Fusion 113
7.2.2 Array contraction 114
7.2.3 Effects of loop fusion and array contraction 115
7.3 Temporal Locality, Aldor and Iterative Solvers 116
7.3.1 Temporal locality of original programs 116
7.3.2 Finding opportunities to improve temporal locality 117
7.3.3 The impact of modularity 119
7.3.4 Applicability of proposed method 120
7.4 Summary 121

8 Iterative Collective Loop Fusion 123
8.1 Loop Dependence Graph 123

8.2 Collective Loop Fusion and Fusion Partitions 125
8.2.1 Array contraction 127
8.3 The Motivation for Search 127
8.3.1 Standard model based approach 128
8.3.2 Iterative optimisation 129
8.3.3 Previous approaches and this work 130
8.4 Iterative Collective Loop Fusion 132
8.4.1 Generating legal fusion partitions 133
8.4.2 Search heuristics and search space reduction 142
8.4.3 Algorithm for generating test cases 143
8.4.4 Code generation 144
8.5 Summary 145

9 Prototype and LDG Construction **147**
9.1 LDG Recovery 147
9.1.1 Control flow 148
9.1.2 Loop index variable ranges and strides 149
9.1.3 All statement dependencies 151
9.1.4 Guarded sections 151
9.2 Prototype Implementation 152
9.2.1 LDG data structures 153
9.2.2 Code generation 153
9.3 Issues with Prototype 154
9.3.1 Constant folding 154
9.3.2 Action of the inliner 155
9.3.3 Inlining of generated code 155
9.3.4 Emerging and unboxing hints 156
9.3.5 Data cache associativity 157
9.4 Summary 157

10 Experimental Results **159**
10.1 Example LDG 160

10.2 Enumeration of Fusion Partitions 163
10.3 Evaluation Environment 164
10.3.1 Machines 164
10.3.2 Compilers 165
10.3.3 BLAS routines 166
10.3.4 Generation of timing results 166
10.4 Iterative Search Experiments 167
10.4.1 3D operator 168
10.4.2 4D operator 173
10.4.3 Wilson-Dirac operator 177
10.4.4 Variability within (c, p) pair sets 180
10.4.5 Variability across setting 181
10.5 Control Experiments 184
10.5.1 Naive control 184
10.5.2 Other methods of collective loop fusion 184
10.6 Other methodologies 187
10.6.1 Fortran 3D stencil 187
10.6.2 Use of high-performance libraries 189
10.7 Discussion of Results 192
10.7.1 Interloop locality in Aldor 192
10.7.2 Search 192
10.7.3 Code generation 193
10.7.4 Local tuning and code generation 194
10.8 Summary 196

11 Conclusion **199**
11.1 Summary 199
11.2 Future directions 201
11.2.1 Framework design 201
11.2.2 Solver domains 203
11.2.3 Operators 203
11.2.4 Iterative collective loop fusion 203

11.2.5 Empirical results 204
11.2.6 Other optimisations 205
11.2.7 Other languages 206

A Conjugate Gradients and the Lanczos Type Product Methods 209
A.1 Conjugate gradients 209
A.2 The Lanczos type product methods 210
A.3 Functional parallelism and product methods 211

B Re-use in the Operator Application 213
B.1 3D Pure Stencil with One Level of Cache 214
B.2 4D and Concrete Operators 215
B.3 Operator Tiling in Practice 216

C Categories 217
C.1 Linear Algebra Categories 217
C.2 Problem Specific Categories 225

D Domains 231
D.1 Scalar Domains 232
D.2 Wilson-Dirac Subdomains 233
D.3 Vector and Operator Domains 236
D.4 Solver Domains 241

E Published Papers 253
E.1 "The Paraldor Project" – 2003 253
E.2 "A Modular Iterative Solver Package in a Categorical Language" – 2003 255
E.3 "Cross-Component Optimisation in a High Level Category Based Language" – 2004 256

Bibliography 259

Chapter 1

Introduction

The writers of scientific computing codes should ideally have a computer language that gives them brevity and elegance of expression, portability and performance. Elegance of expression implies a high level language with support for layering of abstractions and clear and concise exposition of the operations at any given level of abstraction, with a correspondence as close as possible to the original mathematical expression of the problem. Portability means the ability to reuse the same programs on different machines with the minimum of effort. This in turn implies maximum automation of the process of producing an executable for a given machine from the original source programs. Finally, performance suggests that the executables so produced ought to be as efficient as possible.

As a general rule, elegance of expression and portability are sacrificed for the sake of performance, and much work is done by hand, with authors writing codes in low-level languages and applying transformations on a per machine basis. However, as computer architectures get increasingly complex this process in itself becomes a difficult task, with many trade-offs that are usually impossible to analyse directly even for a single machine. This is analogous to the difficulties facing compiler writers, where the limitations of the traditional approach, which relies on simplified machine models, static heuristics to guide decisions and fixed orderings for applying different optimisations, are getting increasingly serious, especially when dealing with multiple architectures. These problems can be approached using a technique known as *iterative* or *feedback driven* compilation [16], which treats choosing the transformations to ap-

ply from some known set as a search problem, with the goal function being the actual performance of an executable. The increased compilation costs incurred by compiling and testing multiple versions of a program are acceptable within the domain of scientific computing, as they are outweighed by the expected returns over the lifetime of compute-intensive codes that run for long periods of time.

Interestingly, as a result of the difficulty of achieving good performance, portability and elegance of expression can be regained. Given that search techniques are employed, achieving performance across different architectures essentially comes for free as search is applied to a program on each architecture to find the optimisations that work. Similarly, given that the effective transformations are not known ahead of time, they are not directly present in the encoding of the problem, and so clarity is not lost.

This thesis presents an investigation of these issues starting from the construction of a framework for a group of scientific computing applications using a modern high level language, through to steps toward achieving portable performance using iterative optimisation. The emphasis is on representing the modularity of the algorithms cleanly and explicitly, and studying the optimisation issues that arise from the conjunction of the language, the modular style it encourages and the framework design.

1.1 Computational Science Domain

Finding answers to many important problems in scientific computing, such as modelling the evolution of physical systems, requires finding the solution to large systems of linear equations using numerical methods. This is the application area that will be investigated. The systems considered in this thesis equate to solving $Ax = b$ for known vector b and unknown x, with some square nonsingular matrix A that contains mostly zero entries (i.e. it is *sparse*) due to the problem from which the linear system is derived. This sparsity structure is usually exploited to save computer storage space and work by avoiding representing or manipulating the zero entries.

There are broadly two standard approaches to solving such systems, *direct* or *iterative* methods. Direct methods decompose the matrix A into factors that can be solved against trivially, while iterative methods produce a series of approximations to x until

a good enough approximation has been found. Direct methods have the advantage that once the factors have been produced, they can be reused for multiple right hand sides. When applied to sparse systems however, the manipulations of the matrix may cause entries that were previously zero to become nonzero (a process known as "fill–in"), which results in an increase in storage requirements if previously the nonzero entries were not stored. For very large, very sparse systems, this increase may be dramatic and unacceptable. Conversely, iterative methods only require matrix–vector products rather than direct manipulation of A. For a single right hand side, iterative methods may converge (find a good enough approximation) in some small number of steps and consequently require much less arithmetic than a full factorisation. Only requiring the matrix–vector products (and vector operations) also means that the methods are more natural for solving problems derived from approximations to continuous systems, as all the necessary steps have continuous counterparts. These benefits make the methods popular for certain applications, and thus an interesting subject for research.

1.2 Language

Conceptually, a large subset of the iterative methods can be thought of as the composition of various different algorithmic pieces, with different choices giving rise to the different algorithms. However, the algorithms are usually presented (and named) as separate entities with the pieces merged together (individual *recipes*), resulting in a confusingly large number of closely related methods with different numerical properties. Combining choices in this way is not only obfuscatory, but wasteful in terms of effort given that essentially the same algorithmic steps are programmed repeatedly across different iterative solvers. An alternative approach as pursued in this thesis is to design an *algorithmic framework* to keep the individual pieces as separate as possible and provide a means to join them together to create a given solver.

The goal of explicitly representing as much structure as possible implies that it would be a good idea to choose a language that naturally provides support for the activity. The language used in this thesis is called *Aldor*, and has its roots in the computer

algebra community as the "library language"[1] for the Axiom computer algebra system [48]. It is a self-contained functional language, and is similar to members of the ML family [4, 3]. Various algorithmic pieces used to construct the solvers can be represented as recurrences and coded as functions that carry state. The functional features of the language are used to provide a natural way to compose these recurrences to give the desired solver. In addition, the algorithms are naturally independent of the objects that they manipulate, such as matrices, vector spaces and scalars etc. Aldor is statically typed, and the type system is used throughout to encode the relationships between these lower level mathematical objects, as well as the algorithmic pieces.

Although there are other statically typed functional languages, the implementation of the type system and extra features such as overloading and *generators*[2] make the capturing of the structure particularly elegant. In addition, the design of the language and compiler is geared toward enabling efficient numerical programs, rather than solely concentrating on symbolic work which is typically the emphasis of other functional languages, and so provides a natural platform.

1.3 Compiler Optimisations

There are many different automatic compiler optimisation techniques that can be used to improve the performance of programs, ranging from low level (instruction selection, scheduling etc) to high level (inlining, loop restructuring etc). The goal of the investigation into optimisation conducted in this thesis is to concentrate on those aspects of the optimisation problem that are specific to the conjunction of the computational science application domain and the language – i.e. the way in which the framework is expressed in Aldor. To better describe and motivate what we concentrate on, there follows a quick summary of issues we do not deal with directly, along with the reasons why, followed by an outline of the actual subject covered.

The first group of problems that are not dealt with are the standard functional language issues that may have relevance to Aldor, but are not especially important for the application. These include storage management issues from the use of the garbage col-

[1] Aldor allowed the compiling of library functions to native machine code to improve performance.
[2] For a description of this language feature see Chapter 2.

lector and direct overhead from the use of advanced functional features. Aldor allows the explicit management of named objects thus making the whole subject of automatic storage management only tangentially relevant. The direct cost of functional language features (i.e. first class functions) stems from the overhead introduced by calling conventions necessary to support them, and the use of closures etc. Functional features are used in a simple way throughout the framework, with a handful of more complex uses to join separate recurrences to build the algorithms. The overhead for the simple instances is removed from the solver programs by the optimisations that already exist in the compiler, and the direct cost of the overhead for the remaining cases is inconsequential for the application at hand.

The second group of problems, which are not specific to either the language or the application, are the family of low level code generation techniques from standard compiler theory. These tend to be more machine oriented and applicable to any language. Also, the Aldor compiler achieves portability by generating C code, another factor in ruling out a number of low level optimisations as they cannot be directly expressed in a language such as standard C.

The subject that is actually tackled is the issue of the *indirect* costs of functional language features and modularity, and sits somewhere in between the two previous groups. This subject has received fairly limited attention from the functional language community, although they have developed some elaborate analyses to recover control and data flow information in the presence of higher order functions, and used these to attack certain specialised cases of indirect cost that would arise if the information was unavailable. However, the cost of the initial presence of higher order (or even simple) functions subsequently removed by optimisations (such as inlining) is not usually mentioned, despite the fact that it is clearly a significant obstacle to the generation of efficient code, especially when the original individual functions contain loops. The technique applied to this problem in this thesis is high level loop restructuring, specifically with regard to *temporal locality*[3], a characteristic that is strongly affected by the modular programming style that Aldor encourages.

High level restructuring compilers that target temporal locality have often used

[3]For a definition of this term, see Section 7.1.5.

source to source transformations and left low level optimisation to a native compiler [99, 15], so, in contrast to low-level techniques, this approach sits well with the current implementation of the compiler. Global (or collective) loop restructuring has relevance to both Aldor and the solver framework – the global loop structure is a result of their synthesis.

1.4 Linear Systems

Although the high-level structuring of the codes gives rise to the interesting optimisation problems, the solver algorithms do not specify the exact structure of the basic objects that they manipulate. Consequently, they do not constitute a code optimisation problem in isolation – they must first be paired with some implementation of a linear system to be solved.

The first two examples used in this thesis are systems modelled on those that arise from a direct discretisation of a simple partial differential equation problem. The third example is taken from Quantum Chromo Dynamics (QCD), a problem to which iterative solvers are frequently applied. QCD is interesting in that it has a very rich mathematical structure, is very compute intensive, and serves as a stimulus for research into algorithmic variations on iterative methods, tailored to exploit problem structure for obtaining maximum speed. This indicates that implementations have to be very efficient, whilst at the same time suggesting that a flexible framework to allow rapid investigation of novel algorithmic approaches would be valuable.

QCD problems are frequently run on large parallel computers, including several purpose-built machines over the years [23, 17], and are so numerically intensive that practitioners write specialised assembly level tools to get the maximum efficiency possible from a machine for a compute intensive production run[4]. As such, it is an interesting target to aim for – indeed, one version of the problem forms an application in the SPEC [5] CPU2000 benchmark suite (`wupwise`).

[4]The exercise is more like one of hardware/software co-design to get maximum performance for the specific problem when using custom designed machines.

1.5 Thesis Outline

The important aspects of the language with respect to the design of the framework and the optimisation issues are given in Chapter 2. Chapter 3 gives a brief summary of the necessary background on the structure of Krylov subspace based iterative solvers. The route taken is the derivation of the methods from an Arnoldi or Lanczos process coupled to an orthogonality condition. Chapter 4 discusses general previous work in the compilation of higher order language features and its ramifications for the handling of aggregate types such as arrays, the automatic management of storage with a garbage collector, and the wider implications for the overall design of compilers. It also covers some previous work on the compilation of computer algebra languages.

Chapter 5 highlights the important parts of the framework design with code extracts where relevant. A fuller listing is given in appendices C and D. The chapter begins by presenting a type hierarchy that captures the relationships between components at different levels and their individual interfaces, followed by a description of some example components that instantiate parts of the framework to give an iterative solver algorithm. Chapter 6 introduces the three examples of sparse linear systems used with the solver framework, and gives the essentials of how they are implemented. The emphasis is on those implementation details that are important for a discussion of optimisation issues.

If the individual functions associated with an abstract data type are taken as *components*, the optimisation problem is one of *cross-component optimisation*. In the specific instance considered in this thesis, the problem reduces to one of *loop fusion* with subsequent *array contraction*. The relevant basic terminology and formalisms are introduced at the beginning of Chapter 7, followed by a link to the structure of the iterative solver programs. This leads into a discussion of why fusion and contraction were used rather than any other transformations given the overall objective of targeting performance through temporal locality of data.

Chapter 8 builds on Chapter 7 by introducing the notion of collective loop fusion (and contraction) as a means of describing how the basic technique of loop fusion should be applied to a collection of loops. This is followed by a summary of previous work in the area and a detailed exposition of the new approach developed in this work.

The central issue in this chapter is the relation of the standard abstract model of the problem to concrete hardware – more specifically how trade-offs between transformations must be managed to get maximum performance, how different transformations that are ranked equal using an abstract goal function may actually have substantially different performance on a real machine, and how these discrepancies can be attacked using *iterative optimisation*. The resulting method is called *iterative collective loop fusion*.

The automatic construction of the data structure used for optimisation in this thesis would require several techniques that are not yet available in the Aldor compiler, and the first half of Chapter 9 outlines the necessary analysis. The second half introduces the prototype used to investigate the optimisation of the iterative solvers.

Chapter 10 describes the data structure used for optimisation derived from the most significant part of the code for a QMR algorithm – the two sided Krylov subspace procedure described in Chapter 5. This is followed by a description of the search through the space of the possible transformations using the iterative collective fusion technique for different combinations of machine, operator type (as described in Chapter 6) and data set size. The empirical results are compared against the case where no fusion is done at all, alternative techniques for collective loop fusion, and entirely different methodologies including an equivalent algorithm written in Fortran and specialised versions of the code where subsections have been replaced with combinations of C, assembly and high-performance binary BLAS routines.

Chapter 11 ties together the separate strands in the thesis into a conclusion and summarises the directions in which the work could be taken. The appendices contain code extracts (mentioned above), a discussion of how the framework relates to some other iterative solvers (Appendix A), a brief discussion of temporal locality for three and four dimensional regular stencils (Appendix B), and some notes on the two refereed conference papers and one workshop paper published during the course of this thesis (Appendix E).

1.6 Contributions

The contributions made in this thesis are as follows:

- A framework for explicitly representing the structure of and relationships between a subset of the family of iterative solver algorithms, written in Aldor.

- An examination of how a clean modular style applied to a problem domain such as the iterative solvers gives rise to temporal locality problems for cache based architectures, and how this problem can be addressed by expressing it in terms of a loop dependence graph and applying the transformations of loop fusion and array contraction (collective loop fusion).

- A demonstration of the importance of these issues to the combination of language and application, with speedups of up to 3.7 from transformations targeted at them.

- An empirical investigation of how the different choices of transformation affect the performance of the resulting code, with an emphasis on the inaccuracies that can be introduced by using abstract models of the problem.

- The embodiment of this approach into the technique we call *iterative collective loop fusion*. This approach gives speedups of up to 1.41 over well-known static approaches to the collective loop fusion problem.

- A comparison with alternative approaches, such as an equivalent program written in Fortran, or alternatives based on combinations of Aldor, C or assembly code and high-performance binary ATLAS BLAS routines.

Chapter 2

Aldor

This chapter summarises the language and its implementation, including features of the source language itself, the intermediate representation that is used during compilation and compilation strategy, including an outline of some existing optimisations. For further details, including the exact specifics of language syntax, see [93, 95]. The chapter serves to describe some of the important features that made Aldor our choice for this work, as well as enabling a discussion of the design, implementation and optimising transformation of the iterative solver applications themselves. A detailed description of why these features are important and how they relate to alternatives has been covered in previous work, and is summarised with references in Appendix E.

2.1 Fundamentals of the Language

The code in Figures 2.1 and 2.2 is used as an example throughout this chapter to illustrate some of the basic language features.

2.1.1 Domains and categories

Domains in Aldor are the mechanism used to implement abstract data types (ADTs) of a single implicit new type, or occasionally packages of functions on one or more explicit types. For an ADT, the underlying, hidden type of the domain elements is called the *domain representation.* Domains are typed by belonging to *categories.* A

```
define MyVectorCat(GroundField : Field) : Category == with {
  - : % -> %;                                               ++ negation
  + : (%, %) -> %;                                          ++ addition
  - : (%, %) -> %;                                          ++ subtraction
  * : (%, %) -> GroundField;                                ++ innerproduct
  apply : (%, SingleInteger) -> GroundField;                ++ element access
  set! : (%, SingleInteger, GroundField) -> ();             ++ element update

  default (v: %) - (w: %): % == v + (-w);
}
```

Figure 2.1: Aldor source code for a simple vector category with negation, addition, subtraction and inner product on the vectors, as well as individual element access and updating. The category is parameterised by the type of the vector elements, which must be a field.

category consists of a list of constants (or *exports*) that the domain must provide, i.e. make public to users of the domain, and these constants are either functions, types (one implicit type is created for an ADT, but a package may export several) or distinguished values of a type. As such, a category defines an interface and any domain belonging to it provides an implementation thereof.

A domain may belong to multiple categories, and only belongs to a named category if it is explicitly defined as such – that is, a domain never satisfies a category implicitly merely by exporting the constants that the category requires, unless the category is an anonymous one. If a domain is explicitly typed with a category, then only the constants defined in the category are exported, thereby allowing data/implementation hiding. If the domain satisfies multiple categories, then it exports the union of their constants. A domain that is not typed by a category implicitly belongs to the anonymous category that consists of a list of all the constants defined in the implementation of the domain.

2.1.1.1 Parameterisation and recursion

Both domains and categories can be parameterised, allowing the definition and typing of type constructors (parameterised domains are also known as *functors*). The argu-

```
MyVector : MyVectorCat(DoubleFloat) with {
  new : () -> %;
  dispose! : % -> ();
} == add {
  Rep == PrimitiveArray(DoubleFloat);
  import from Rep;
  vectorSize : SingleInteger := 10;

  new() : % == per new(vectorSize, 0);
  dispose!(e : %) : () == dispose!(rep e);
  apply(v : %, i : SingleInteger) : DoubleFloat == (rep v)(i);
  set!(v : %, i : SingleInteger,
       e : DoubleFloat) : () == (rep v)(i) := e;

  - (v: %) : % ==  {
     r := new();
     for i in 1..vectorSize repeat r(i) := - v(i);
     return r;
   }

   (v: %) + (w: %): % == {
     r := new();
     for i in 1..vectorSize repeat r(i) := v(i) + w(i);
     return r;
   }

   (v: %) * (w: %): DoubleFloat == {
     ip : DoubleFloat := 0;
     for i in 1..vectorSize repeat ip := ip + (v(i) * w(i));
     return ip;
   }
}
```

Figure 2.2: Aldor source code for a domain of vectors that satisfies (implements) the category in Figure 2.1. The variable `vectorSize` is a lexically scoped variable that resides in the scope of the domain itself and is referred to by the functions for vector operations (negation, addition and inner product).

ments to a parameterised domain or category can be dependently typed (see section 2.1.2). Both domains and parameterised domains are also first class objects, in the same manner as normal objects/functions (again see section 2.1.2). The mechanism of domains and categories can be thought of as a *module language*, as it is used to define new modules (i.e. domains).

Domains, and the categories used to type them, can be defined recursively and mutually recursively. This can be used in conjunction with parameterisation and dependent types.

2.1.1.2 Relation to Example

Figure 2.1 gives an example of a category with six exports (all function signatures) intended to denote negation of a vector, the addition/subtraction of two vectors to form a third, the inner product of two vectors to give a member of the element type, and reading/writing of an individual vector element. Functions matching the signatures must be provided by any domain that is typed with the category. The category is parameterised by the domain of the vector elements, which itself must be typed by the `Field` category.

The domain in Figure 2.2 is typed with an anonymous category that inherits from the category in Figure 2.1 and extends it with a constructor and a destructor function. The domain representation (defined by `Rep == ...`) is a domain of double precision floating point number arrays, created using a built-in array functor (see section 2.1.3) and a domain of double floats. All the functions in the domain are implemented using the exports provided by the domain representation and the double float domain. Only the exports defined by the typing category are visible externally, so the type of the domain representation and the `vectorSize` variable are hidden.

2.1.2 General features

Aldor is *strictly evaluated* (or *eager*) and *imperative*. It is a *functional* language with *lexical scoping*, where variables from an outer lexical scope can be imperatively updated. The combination of imperative update, lexical scoping and first class functions

allows the creation of closures that carry state by manipulating values in their lexical environment.

As well as functions, domains are also first class values in Aldor, and thus functors can be instantiated with varying domain parameters decided at run-time (although this rarely occurs in practice). Aldor is *statically typed* and *type inferred*, and supports a limited form of *dependently typed* arguments to functions. The dependent typing is mostly used in this thesis to be able to type domain arguments to a function or parameterised domain using a parameterised category, but in general it is necessary for typing recursive domains etc.

Dependently typed functions allow the specification of an algorithm that acts on elements of a domain at the level of exports provided by a particular category, independently of the domain from which the elements are taken. *Category defaults* can be used as a shorthand for the same mechanism. They provide a default implementation of a given export in terms of other operations provided by the same category. The domain that provides the other exports upon which the default relies is an implicit parameter.

2.1.2.1 Relation to Example

Figure 2.1 gives an example of a default function (subtraction) defined in terms of other category exports. In Figure 2.2, the variable `vectorSize` used in the functions is lexically scoped (it resides in the scope of the domain itself).

2.1.3 The core language and the abstract machine

The core of the language consists of a small number of pre-defined (or built-in) domains [93]. The most important of these are the `PrimitiveArray`, `Record`, `Machine` and `Generator` types. The machine domain (i.e. `Machine`) is a package of simple types and operations, such as a single word integer with addition/multiplication etc, and single and double word floating point types. The primitive array and record domains are both functors that allow the definition of aggregate data types in the expected fashion, with functions to access and update elements and create aggregates from collections of elements.

Although the machine domain supports operations to create and manipulate arrays of arbitrary type, the primitive array domain is more than just a type system wrapper for this for several reasons, the most important being the *packed array* mechanism (see section 2.3.1.2).

The Aldor compiler compiles the source language to an intermediate representation consisting of operations defined on an abstract machine (see section 2.2). The operations and data types provided to Aldor by the machine domain and the record and primitive array types correspond closely to a large subset of the operations and data types provided by the abstract machine.

2.1.3.1 Generators

The `Generator` type is a core language functor that can take an arbitrary type as its argument to define a domain of *generator* objects that "generate" elements from the parameter type. Objects in the domain are constructed using special syntax and almost arbitrary control flow (including general recursive functions), and are first class (they can be assigned to variables etc). Generators are used with the Aldor `for` loop construct to define the values which the `for` loop variable takes. At the beginning of each iteration, the generator is prompted for its next value which is then bound to the loop variable for the duration of the body. This usually continues until the generator is empty, although other exits from within the loop can occur.

All `for` loop constructs in the language rely on generators, including those which equate to the simpler constructs with the same name from less developed languages, such as simple iteration over a closed integer segment. Note that generator objects have state that can be carried across different `for` loops. To get the expected behaviour for a sequence of loops that iterate over the same set of values (such as an integer segment), a new generator must be created for each loop. This is made the default behaviour by a syntactic mechanism. If the object supplied as an argument to a `for` loop is not a generator, a function to create a generator from the object is implicitly invoked, which usually results in a new generator for each `for` loop even if the same (non generator) object is reused.

A generator is implemented as a small set of functions that manipulate the same

environment (see 2.2.5). Thus, the invoking of the function to get back a generator, and its subsequent use in the `for` loop is a simple, but important and pervasive use of the functional features of the language.

2.1.3.2 Relation to Example

The domain in Figure 2.2 uses the core language functor `PrimitiveArray` to create its domain representation. The `for` loops to iterate over vector elements are controlled by generators of one word integers created from the integer segments `1..vectorSize`, where each successive integer value is bound to variable `i`.

2.1.4 Basic libraries

Because the built-in elements of the language form such a small group with very minimal functionality, it is usual practice to write programs in terms of a *basic library* that provides a much richer set of abstract data types with far more available operations on them. This basic library is likely to include what would otherwise be considered fairly fundamental types for other languages, such as list, tree, integer and floating point types. Nonetheless, any given basic library has been written as user defined domains and categories, and as such is just a convention – indeed, there is more than one Aldor basic library [93, 18].

The basic library used in this thesis is called `axllib`, and most of the domains of interest from it are fairly thin wrappers around a machine domain type that corresponds exactly to an abstract machine type. The wrapper exists to supply many extra functions, and also to separate the relevant part of the machine domain package into a stand-alone type. The library also contains a hierarchy of categories that form the basis of the work in Chapter 5.

2.1.4.1 Relation to Example

As well as the core language array functor, the underlying type of the domain in Figure 2.2 relies on the `DoubleFloat` domain from the `axllib` basic library. The `SingleInteger` and `Segment` domains are also used to control `for` loops etc.

2.1.5 Storage model

Aldor does not have arbitrary pointers, but it does have references through which updatable structures such as records and arrays are handled. This allows the creation of aliases via shallow copying (but not partial aliases as references must point to the head of an object, and objects may not overlap), and also introduces undefined behaviour if a reference with no object attached to it is read or written through. Aggregates are therefore fundamentally different from simple variables such as the integer or floating point types provided by the machine domain.

Updatable aggregates must be allocated and can either be explicitly deallocated or left to a garbage collector. Allocation/deallocation functions for arbitrary aggregate types can be written using those from the core language domains, but deallocation routines, usually called `dispose!` functions, are not strictly necessary (because of garbage collection). Most allocated objects at the source level corresponds to equivalent objects in the heap of the underlying abstract machine, so the ability to garbage collect objects and the strategy employed is inherited from the abstract machine's memory model. The correspondence is not strict however, as certain objects with a restricted lifetime may be turned into a collection of abstract machine stack frame variables by compiler optimisations (see section 2.3.1.3).

2.1.5.1 Relation to Example

The function to create a vector in Figure 2.2 explicitly allocates a new array by using the functionality provided by the domain representation, and similarly provides a `dispose!` function.

2.1.6 Purity and overloading

Syntactic operators are a pre-defined part of the language, and it is possible to overload most of them with user defined functions, including the syntax for function application. The overloading of syntax for fetching or updating elements of records or arrays (known as the `apply` and `set!` mechanisms respectively) is of particular interest and allows the variables of an arbitrary domain to be treated syntactically as aggregates.

Overloaded infix operators such as + and * are particularly common exports. Heavy use of overloading, especially in the basic libraries, leads to programs typically composed of layers of very fine-grained functions.

Although Aldor is an imperative language, the functions attached to the infix operators exported by a domain of updatable objects tend to be pure to preserve the natural semantics of the operator – that is, they do not destructively update either of the arguments, and allocate a new object to hold the result of the operation. Given that abstraction and overloading are used so heavily in typical Aldor programs, the net result is a heavy bias toward object allocation rather than programmer directed explicit overwriting of existing aggregates element-by-element using loops, as tends to occur in more traditional imperative languages. Instead, `dispose!` functions are used to discard an object if the data it holds is no longer needed.

2.1.6.1 Relation to Example

All the exports from the domain in Figure 2.2 are overloaded operators, excluding the constructor/destructor functions. The reading and writing of array elements within `for` loops demonstrates the use of `apply` and `set!` functions exported by the underlying array domain (as well as infix operators such as + etc from the double float domain). The syntactic convention gets resolved to the relevant export from the domain – i.e. the form `v(i) := 4` is equivalent to `set!(v, i, 4)`, and `v(i)` with no assignment operator is equivalent to `apply(v, i)`. Their use for the vector domain itself would be similar.

The addition of two vectors is an example of a function attached to an overloaded infix operator that uses destructive updates internally, but is outwardly pure in that it affects neither argument and always returns the same values in a vector for a given argument pair.

2.2 Abstract Machine and Compilation Model

The current design of Aldor incorporates an *abstract machine*, some of whose functionality matches closely the built-in machine domain. The code for the abstract ma-

chine, also known as the intermediate representation (IR), is called First Order Abstract Machine (FOAM) code.

FOAM is a fairly high level procedural language, however, quite a lot of information that was implicit at the Aldor source level is made explicit at the FOAM level. This includes:

- closures, their environments, and the instructions for manipulating them, including lexical references
- almost all compiler generated temporary variables
- most of the code to initialise domains at run-time
- the implementation of generators as collections of closures
- control flow within functions, which is lowered to the level of labels and branches

FOAM itself is structured as a fairly simple tree, the details of which are given in [94] (although it has since developed somewhat).

2.2.1 Libraries and whole program optimisation

The compilation model that Aldor uses is *whole program optimisation* (WPO) , and this is achieved by splitting compilation into two phases. The first phase is the compilation of source files to abstract machine code. The second phase is when the native executable is actually created, and at this step all the abstract machine code for any domains or functions that are used by the executable must be available.

To enable this, Aldor libraries (including basic libraries) consist of files written in a machine-readable version of FOAM code, and are intended to take the role of collections of machine independent object files. The WPO strategy is a key enabler for optimisation of the language, as it enables cross-component optimisations in the presence of multiple compilation units. For example, in the current compiler it makes sure that there's enough grist for the inliner to work on by employing cross-file inlining. Similarly, it enables other cross-component optimisations, such as those introduced in Chapter 7.

2.2.2 FOAM types and variables

FOAM has a number of built-in types that fall into two main groups. The first of these is the updatable *compound variables*, including closures, environments that represent scope levels in the source language, arrays and records, all of which which are allocated on the heap and handled via references. Records and arrays in FOAM correspond directly to the core domains in the source language. Records and environments are typed by means of globally visible *formats*, which describe the number and type of their elements. The type of an array is determined by the type of its elements, which must be one of the simple variables (see below). Arrays of compound types are constructed as arrays of pointers.

The second group is the *simple variables*, including a generic word type which is used as a catch-all (any heap allocated variable is pointed to by its reference which is stored as a word), and various integers and floating point values usually represented by one or two words. The simple types and the operations on them correspond to those available in the machine domain of the core language. These variables are stack allocated (i.e. they are specified as part of a FOAM function stack frame) unless they are part of a record/environment.

2.2.3 Uniform representation rule

The *uniform representation rule* states that any type used as the representation of a domain must fit into a single word. This is necessary in the context of dynamically handling domains – i.e. when the domain parameter to a dependently typed function is not statically determinable. In this case, the size of the representation for the data type cannot be known, so all data types must have the same size.

This is achieved by converting compound types into the generic word type i.e. handling them through uniform size pointers. Note that this calling convention means that any simple data type larger than one word must be *boxed* inside one of the compound data types if it is to be used as a domain representation.

2.2.4 Memory model

The lifetime of stack variables is determined by the use of stack frames for function calls. Heap allocated variables can either be explicitly deallocated with instructions resulting from `dispose!` functions, or left to a garbage collector. An implementation of the abstract machine is free to ignore deallocation instructions, so the use is more a programmer optimisation hint than an intrinsic part of a program's semantics.

2.2.5 Aldor domains and generators in FOAM

The implementation of Aldor domains in FOAM code relies on a mixture of FOAM instructions generated by the compiler and support provided by the run-time system. Domains exist at run-time as lazily instantiated objects containing references to any constants exported, and an environment for any internal variables contained within their scope. The use of laziness allows support for language features such as recursive domains.

Domain objects are implemented by allocating an empty shell that contains a pointer to a compiler generated initialisation function. When certain information is needed from the domain, the initialisation function is used to fill-in parts of the object. This can happen in several stages, or all at once, depending on the event that triggers the initialisation. Code to trigger the initialisation of the domain must also be inserted by the compiler in the appropriate places to ensure program correctness, using abstract machine instructions specific to this purpose. The most common example is the need to ensure that a domain has been created before calling an exported function that refers to variables in its lexical environment (i.e. the scope of the domain itself, the environment for which is part of the run-time domain object). This is done via the FOAM instruction `envEnsure`.

As domains may rely upon other domains, a single `envEnsure` instruction may trigger a long chain of domain initialisations. Instructions to trigger domain initialisation are usually littered throughout any piece of FOAM code, and so can be control dependent on dynamically determined branches etc.

Generators are first created at the FOAM level as a small collection of closures

(step!, value, empty? etc.) that manipulate state in a common lexical environment. There is no special functionality required to support them. The action of compiler optimisations at the FOAM level may reduce a generator to stack variables and simple control flow using labels and branches, which can dramatically decrease the overhead otherwise involved (see section 2.3.1.3).

2.3 Compiler Implementation

The Aldor compiler can be roughly split into a front-end and a back-end that match the phases of the compilation model. The front end of the compiler is responsible for lexing, parsing, type inference, semantic analysis etc., and finally the generation of the IR. Other than the fact that it generates FOAM code, the detailed implementation of the front-end is irrelevant to this thesis.

The back-end of the compiler has two main phases. The first consists of several compiler optimisation passes that operate on the IR, and the second consists of a more-or-less direct translation of the final version of the FOAM code into another language to be compiled by a host compiler. The two languages currently used for this are C and Lisp, but only the C back-end is considered in this thesis. Note that the generated code is not completely free-standing, in that it relies on a small run-time environment that must be a linked against the native executable to provide support for domains and garbage collection etc.

The memory allocation in the run-time library for the C back-end makes use of a conservative tracing garbage collection scheme, with the ability to explicitly deallocate objects on demand. Only the compound FOAM types are heap allocated, with simple types translated into stack variables at the C level. It is quite often the case that a huge number of stack variables get defined in this way, but the assumption implicit in the design is that it is better to leave this mess to the register allocator of the C compiler than it is to generate a large amount of garbage for the collector to have to deal with. Indeed, the emerger pass of the compiler is specifically designed to turn heap allocated records and environments into simple FOAM variables, and frequently gives quite dramatic improvements in performance.

2.3.1 Pre-existing optimisations

2.3.1.1 The inliner

The inliner is an important part of the Aldor compiler. The current implementation is fairly simple, and works top-down on FOAM code. It operates on a per function basis, and given a function it gathers a list of the functions that are called from it. This list is prioritised based on a number of factors including:

1. *Generators* – if the functions result from a generator in the top level source code, then the inliner prioritises them. The inlining of generators is crucial to the performance of general code as their use is pervasive.

2. *Function size* – smaller functions are prioritised over larger functions.

3. *Leaf Functions* – leaf functions have a higher priority than non-leaf functions.

Once the queue is assembled, the inliner begins to expand functions into the current function in order of priority. If a non-leaf function is inlined, the child functions are added to the priority queue, it is sorted, and the process resumes. This continues until the limit for the growth of the function has been reached. Functions that are not called directly by name, i.e. that are called through a closure, may be impossible to inline as the function at the call-site will not be known until run-time.

2.3.1.2 Packed arrays

The FOAM abstract machine has instructions for allocating flat arrays of any of the simple data types. Arrays of heap allocated objects are arrays of single words which are pointers to the heap allocated objects themselves.

The core language array functor `PrimitiveArray` takes the array element type as a parameter, as per the example in Figure 2.2. By the uniform representation rule the FOAM representation of the parameter domain must fit into a single word. In the example, the parameter domain is `DoubleFloat` from `axllib`, whose representation is a double word floating point number boxed inside a record. To enable flat arrays of data types such as this, a type may optionally export functions to be used by the array

domain that describe how to allocate and manipulate flat arrays of itself, including the word size of the flattened type etc. These *packed array* functions may be written by the user, but, if they are not, the compiler contains a mechanism that attempts to provide them. If it fails, arrays of pointers to heap objects are used instead.

2.3.1.3 The environment emerger

This pass of the compiler reads sections of FOAM code, and unpacks heap allocated records and environments into their constituent parts, which then become separate stack allocated variables. This saves allocating the record (or environment) itself, but also means that any simple FOAM types that were contained in the record will now get translated into C stack variables and hence become visible to C compiler optimisations such as instruction scheduling, register allocation etc.

The use of `DoubleFloat` in Figure 2.2 demonstrates the purpose of the environment emerger. To fit with the standard Aldor calling convention (see Section 2.2.3) and give the domain a pure semantics, its elements are represented by heap allocated records that contain a double precision floating point variable (i.e. it is boxed), and all the binary operations exported by the domain allocate a new record to hold the result that they produce. This can be painful in tight loops, even if the functions exported by the domain are successfully inlined.

The function for an inner product of two vectors (in Figure 2.2) is an example of this. In its original form, assuming that the packed array functions, addition and multiplication operations are successfully inlined, the body of the loop still requires several new records to be allocated on each iteration to calculate the running sum:

1. Fetch `v(i)` from flat array `v` and place in a new record A
(inlined packed array function)
2. Fetch `w(i)` from flat array `w` and place in a new record B
(inlined packed array function)
3. Multiply contents of records A and B and place result in new record C
(inlined multiplication function)
4. Add the contents of records C and `ip` and place result in new record D
(inlined addition function)
5. Set `ip` to point at D

After the environment emerger has done its job, the records involved have been replaced with simple FOAM types and the loop no longer makes any allocations at all. If the running sum is passed out of the function however, then the emerger has to reinsert instructions to allocate a record and box the final result before returning it. This would be done for the last line of the example, but note that this is one single allocation, outside of any loop. This explanation has ignored the allocation/emerging of function environments, but similar reasoning applies.

The usage of packed arrays frequently interacts positively with the environment emerger. In isolation, the function that accesses elements of a packed array must return a heap allocated object corresponding to the domain representation, usually by allocating a new object and copying values from the array into it. Conversely, a write to a packed array copies information from an object into the array itself. If a packed array element is accessed, operated on, and then stored, the environment emerger can frequently avoid the allocation and use stack temporaries instead as the lifetime of the object is short and it is never aliased.

2.3.1.4 Sundry optimisations

Various other optimisations are also implemented in the compiler as FOAM to FOAM transformations. These include textbook standards such as copy propagation, constant folding, various peephole optimisations, strength reduction, and more Aldor specific transformations such as the flow converter that cleans up control flow after inlining functions from generators, and several optimisations specific to the run-time handling

of domains.

2.4 Summary

This chapter has introduced the computer language Aldor used in this thesis. Specific points of interest include:

- The module language, consisting of domains and categories.
- The core language types, including the `Machine` package that provides most of the interface to the abstract machine, and the `PrimitiveArray`, `Record` and `Generator` functors.
- The use of overloading, and how this encourages fine-grained function composition and object allocation.
- The smallness of the language, how this encourages the use of basic libraries, and how the compilation strategy of whole program optimisation permits the tackling of the potential inefficiencies of this modularity with cross-component optimisations.
- The definition of the abstract machine, including its memory model and how this is visible at the source level, and how code is generated to deal with domain instantiation and generators.
- The structure of the compiler and its pre-existing optimisations, including the inliner and the environment emerger that help to tackle the problems of fine-grained modularity by removing function calls and unnecessary heap allocation.

Chapter 3

The Iterative Solvers

This chapter gives a brief overview of a subset of the iterative solver algorithms, with an emphasis on the algorithmic structure. This background is the foundation of the design of the framework discussed in Chapter 5. See [73, 41, 40] for more detailed material.

3.1 Notation

Standard notation is used throughout, except for the use of square brackets to indicate a scalar element of a vector or a matrix corresponding to the indices within the brackets. For example:

$$
\begin{aligned}
U &= \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \\
U[1,2] &= 1
\end{aligned}
$$

3.2 Overview

3.2.1 The form of the problem

Square nonsingular linear systems of equations can be simply stated in matrix algebra as follows:

$$Ax = b$$

where A is the $m \times m$ matrix of coefficients of the unknowns, b is the m-vector of constraints, and x is the m-vector of unknowns for which the system has to be solved. Usually, an approximation to the solution is sought with some kind of bounds on the error. Throughout the thesis A is usually referred to as the operator, firstly because it is a linear operator (square matrix), and secondly because the name helps to distinguish it from other matrices that enter the discussion.

3.2.2 Krylov subspaces

Krylov spaces are the foundation upon which the non-stationary iterative solvers are built. A Krylov space is a space spanned by a set of basis vectors produced from an operator and a given starting vector, usually presented as a sequence. The n-th vector in the sequence is the $n-1$-th power of the operator A applied to the initial vector v, with n running from 1 to infinity :

$$\{v,\ Av,\ A^2v,\ A^3v,\ \ldots,\ A^{n-1}v,\ \ldots\}$$

where each vector can be generated by applying the operator to its immediate predecessor (in exact arithmetic). The Krylov subspace of the first n vectors in the sequence generated from operator A and vector v is written $\mathcal{K}_n(A,v)$. If the operator is a (nonsingular) finitely dimensioned linear transformation, then at some point in the infinite sequence the space will stop growing – either the basis vectors will completely span the range of the operator, or they will span some smaller finite dimensional invariant subspace.

The basic idea behind the iterative solvers is to generate up to some number of basis vectors for the Krylov space in question, and choose the candidate solution to the

linear equations x_n as a linear combination of these basis vectors, or those from some very closely related space. The candidate solution is then tested to see if it is a good enough approximation, and if it is, the process terminates. If it is not, then more basis vectors are added to the Krylov subspace and a new, hopefully better approximation is constructed. This cycle continues until an acceptable approximation has been found. This must happen eventually in infinite precision arithmetic as the system is nonsingular – finite precision arithmetic complicates the issue, but it is assumed here that some acceptable approximation can always be generated. It is usual to grow the space by one basis vector each time, then construct a solution and test to see if it is acceptable or not. Hence the algorithm takes the form of a test-repeat loop.

3.2.3 Halting condition

There are various different strategies for deciding when to halt the algorithm, such as attempting to approximate some of the eigenvalues of the operator, or watching the rate of change of certain scalars associated with the algorithm, but we will not go into much detail here. We note that most schemes usually involve the *residual norm* (i.e. the 2-norm of the residual vector) or some approximation to it. The *residual vector* r_n (or just residual) is the projected error on the approximate solution x_n:

$$\begin{aligned} r_n &= A(x_{solution} - x_n) \\ &= b - Ax_n \end{aligned}$$

3.2.4 Orthogonality conditions

When selecting a candidate solution from a solution space of size n, one has n degrees of freedom, being the scalars that determine the linear combination. To fully specify the solution, one therefore needs n independent constraints. The constraints are normally specified in terms of an orthogonality condition on the residual vector – that is, r_n has to be orthogonal to a space of dimension n. Hence, the orthogonality condition can be expressed as follows:

$$C_n^H r_n = 0$$

That is, the residual is orthogonal to the space spanned by the columns of C_n. The different orthogonality conditions have different properties, and deciding on the appropriate orthogonality condition for a given problem is a complex task indeed that is not covered here.

3.2.5 Reduced (projected) system as interface

Taking a basis of the solution space to be S_n gives:

$$x_n \in span\{S_n\} \Rightarrow x_n = S_n y_n \tag{3.1}$$

that is, y_n is an n-vector that specifies the linear combination of the basis vectors that gives the approximate solution at step n. By imposing the orthogonality condition on the residual vector, an equation that the candidate solution vector has to satisfy is derived:

$$\begin{aligned} C_n^H r_n = C_n^H(b - Ax_n) &= 0 \\ C_n^H A S_n y_n &= C_n^H b \\ F_{n,n} y_n &= g_n \end{aligned}$$

This *reduced* or *projected* system has size n at step n – the last line above serves to indicate this, where vectors y and g are of size n and the matrix F is of size $n \times n$ (note that F and g are not referred to again). This will be smaller than the original system and can be solved by conventional means, after which 3.1 can be used to reconstruct the approximate solution. In practice the solution space and orthogonality condition are carefully chosen to make the the projected system as easy to generate and solve as possible, as well as giving certain numerical properties to the approximate solution. In addition, the update of the projected system from size n to $n+1$ needs to be as efficient as possible.

The combinations considered in this thesis give rise to projected systems with a common structure that makes it possible to separate the generation of the reduced system and its solution. The projected systems therefore define an interface that allow the same generating components to be re-used with different projected systems and hence different orthogonality conditions. The interface can be exploited from the other side as well – it is possible to define the components that solve the projected systems in such a way that they can be combined with different ways of generating the reduced system. However, this re-use amongst the solver components has not been fully captured in the implementation discussed in this thesis.

3.2.6 Operator structure

The iterative solver algorithms only require the ability to perform matrix-vector products (an operator *application* to a vector) and vector operations to solve $Ax = b$. Certain aspects of the derivation of the algorithms require that the operator have certain mathematical properties (e.g. Hermiticity), but the algorithms are entirely independent of the structure of the operator, taking structure to mean sparsity patterns or possible decomposition into factors.

This is important for two reasons. Firstly, although the algorithms are independent of operator structure, they are frequently used because of it, as they have no need to alter the operator itself in any way. In the simple case of an operator stored in some sparse matrix format, in may be cheaper to use an iterative algorithm rather than a sparse direct one due to storage considerations (i.e. possible "fill in" during the use of a sparse direct method). Where an operator is stored as factors, for example as the Kronecker product of two matrices, it is not concretely represented at all and its entries are not available to be directly acted on (see Section 6.2.2). Secondly, when the linear system arises from the discrete approximation of a differential operator on some continuous function, the approach is conceptually much neater as it is not clear how the atomic manipulations of the discretized operator correspond to any continuous counterpart. Operator application and scalar/inner products of vectors do usually have a continuous equivalent however.

The independence of the algorithms with respect to operator structure means that

further discussion of these issues can be delayed until Chapter 6.

3.3 Generating Krylov Subspaces

3.3.1 The Arnoldi relation

The methods of generating Krylov subspaces considered in this section orthogonalise the newest vector in the sequence against all the previous vectors, using for example some variant of Gram-Schmidt orthogonalisation, and then normalise it. This approach is called the Arnoldi method, and is neatly captured by the Arnoldi relation:

$$\begin{aligned} AV_n &= V_n H_{n,n} + \beta_n v_{n+1} u_n^H \\ &= V_{n+1} \underline{H}_{n+1,n} \end{aligned} \tag{3.2}$$

where A is the operator in question, the columns of V_n are the orthonormal basis vectors of the Krylov subspace $\mathcal{K}_n(A, v_1)$, vector u_n is the n-th canonical unit basis vector, $H_{n,n}$ is the $n \times n$ upper Hessenberg matrix of orthogonalisation (upper triangle) and normalisation (sub diagonal) coefficients, and $\underline{H}_{n+1,n} = H_{n,n} + \beta_n u_{n+1} u_n^H$ (with an extra row of zeros implicitly appended to the bottom of $H_{n,n}$). Each new (raw) vector, before it is orthonormalised, is created by applying the operator to the previous basis vector in the sequence. This information is contained in the left-hand side of the equation – that is, multiplying V_n by A gives a matrix whose columns are the sequence of raw vectors. The right hand side shows that each raw vector can be expressed as a linear combination of the orthonormal basis vectors of the Krylov space, and more specifically that the n-th raw vector is a linear combination of the basis vectors from 1 to $n+1$ (as $\underline{H}_{n+1,n}$ is upper Hessenberg). Hence, if we have the n-th raw vector, and the basis vectors up to n, we can construct the $n+1$-th basis vector.

Assume A is nonsingular, and $n = m$ such that V_n is $m \times m$ and spans the range of A. If we pre-multiply both sides of the Arnoldi relation by the Hermitian transpose of the matrix of basis vectors, then by the fact that its columns are orthonormal we have the following:

$$V_n^H A V_n = H_{n,n} \tag{3.3}$$

which shows that the matrix of basis vectors is a unitary similarity transformation, and that the matrix of coefficients is the operator projected onto this basis.

To reduce clutter in the following sections, the subscripts on the matrices derived from the Arnoldi relation are shortened as follows:

$$\begin{aligned} H_{n,n} &\equiv H_n \\ \underline{H}_{n+1,n} &\equiv \underline{H}_n \end{aligned}$$

and likewise for their tridiagonal counterparts.

3.3.2 Long and short recurrences

The Arnoldi relation is used more-or-less directly to generate an orthonormal basis of the Krylov space for the *long recurrence* solvers such as *FOM*, *GMRES* etc., when the operator is non-Hermitian. The term "long recurrence" is used to indicate the fact that the number of basis vectors against which a raw vector has to be orthogonalised grows as the sequence goes on.

If the operator is Hermitian and the orthonormal matrix of basis vectors is considered as a similarity transformation (equation 3.3), then it must preserve Hermiticity, and so:

$$V_n^H A V_n = T_n$$

with tridiagonal T_n because a Hermitian upper Hessenberg matrix can only have one super-diagonal. Having T tridiagonal means that the n-th raw vector only has to be orthogonalised against the n-th and $n-1$-th basis vectors as all the other orthogonalisation coefficients are zero. This means that there is now a fixed length *short recurrence* for generating the basis vectors. This is important for two reasons – firstly, the amount of effort required to produce a new basis vector, and secondly the amount of storage

required as the algorithm progresses, are now both fixed. This variation on the Arnoldi method is called the Hermitian Lanczos method[1] (or simply the Lanczos method).

There are two ways of getting short recurrences for the generation of the basis vectors if the operator is not Hermitian. The first is simply to truncate the orthogonalisation process after some fixed number of steps. This then gives us an upper banded upper Hessenberg matrix, where the band width is determined by the number of orthogonalisation steps. While this is indeed a short recurrence, the basis vectors are no longer orthogonal – hence it is called the incomplete orthogonalisation method (*IOM*) by some authors. The second is to use the two-sided Lanczos method.

3.3.3 The two-sided Lanczos method

The two-sided Lanczos method builds a more general non-unitary transformation which projects the operator again into a tridiagonal matrix. The idea behind the method is to build a basis for the Krylov space of the operator $V = \mathcal{K}(A, v_1)$, and a basis for the dual Krylov space of the Hermitian transpose of the operator $W = \mathcal{K}(A^H, w_1)$, and arrange it so that the two bases are biorthogonal — i.e $W^H V = D$ where $D = diag(\delta_1, \delta_2, \ldots, \delta_n)$ is a diagonal matrix. It can be seen that the resulting matrix of coefficients must then be tridiagonal by considering the two complementary versions of the Arnoldi relation:

$$AV_n = V_{n+1}\underline{H}_n \tag{3.4}$$

$$A^H W_n = W_{n+1}\underline{\hat{H}}_n \tag{3.5}$$

and exploiting biorthogonality:

$$\begin{aligned} W_n^H A V_n &= (V_n^H A^H W_n)^H \\ D_n H_n &= (D_n^H \hat{H}_n)^H \end{aligned} \tag{3.6}$$

From equation 3.6, H_n must be lower as well as upper Hessenberg as it is structurally equivalent to a transposed upper Hessenberg matrix, and the same principle applies to $\hat{H}_n^H$. Hence they are both tridiagonal, and:

[1] A similar approach can be employed for complex symmetric (rather than Hermitian) matrices, although the generation of the orthonormal basis may fail as the inner product is no longer definite [34]

$$W_n^H AV_n = D_n T_n = \hat{T}_n^H D_n \tag{3.7}$$

where the biorthogonality of W and V comes inductively from the definition of the two complementary recurrences 3.4 and 3.5 (omitted – see [31] for a concise derivation of this fact and the algorithmic variations described below).

Equation 3.7 provides some constraints on the recurrences, but does not define them completely. Taking γ and $\hat{\gamma}$ as the sub diagonal entries of T_n and $\hat{T}_n$ respectively, then they, along with $D_n = diag(\delta_1, \delta_2, \ldots, \delta_n)$, can be freely chosen provided $\gamma_n \hat{\gamma}_n \delta_{n+1} = \delta_n$, giving two degrees of freedom. Some popular further constraints include $\overline{T} = \hat{T}$ which also implies both T and $\hat{T}$ are symmetric (but not Hermitian), or $\hat{T} = T^H$ with D arranged to equal the identity, and various strategies for normalising basis vectors. Certain choices can simplify the algorithm, but may also affect the stability of the method.

Note that two start vectors rather than one are now needed, one for each recurrence. Although the first start vector is normally determined by the Krylov basis we are trying to generate, the choice of the second is arbitrary, as long as the two start vectors are not orthogonal. In fact, the algorithm can terminate prematurely at any step if it generates a pair of non-zero vectors from the two spaces that are orthogonal before it manages to generate an acceptable solution. This is known as a *serious breakdown*, or in the context of an iterative solver, a breakdown in the underlying Lanczos process. There are various strategies to deal with this situation, such as look-ahead techniques, but apart from noting their existence we will not go into them here.

3.3.3.1 Functional parallelism in the two-sided Lanczos process

There are normally a couple of "join points" in any given implementation of the two-sided Lanczos algorithm, being inner products with one vector from each basis set, but between these points the two sets of basis vectors evolve separately. Because of this, work could be done in parallel, and importantly this includes the application of the operator which is usually the single biggest cost in an implementation of an algorithm. See Section A.3.

3.4 Orthogonality Conditions and Projected Systems

3.4.1 Orthogonality conditions and orthogonal Krylov bases

In this section we introduce the three orthogonality conditions considered in this thesis, and show how they are usually combined with a choice of orthonormal Krylov space basis and solution space to get a reduced system. This section is developed as if for the long recurrence (Arnoldi) methods, but can be made equally valid for the short recurrence (Lanczos) methods by replacing any upper Hessenberg matrices H with their tridiagonal equivalent T.

It ought to be noted that, in exact arithmetic, the three orthogonality conditions for a Hermitian positive definite operator all produce approximate solutions at any given step that are very closely related. The relationship between them is less clear-cut for finite precision arithmetic and non-Hermitian positive definite operators, but the generated solutions may still be closely related to one another.

3.4.1.1 The Galerkin condition

The Galerkin condition requires that the residual of the candidate solution be orthogonal to the space from which it is taken, so $C_n = S_n$ giving:

$$S_n^H A S_n y_n = S_n^H b$$

The standard solution space is $S_n \in \mathcal{K}_n(A,b)$, the Krylov space generated by b, and V_n is calculated as an orthonormal basis of this space by using a recurrence based on the Arnoldi relation. By exploiting the orthonormality of the basis (see 3.3), this results in the following projected system:

$$\begin{aligned} V_n^H A V_n y_n &= V_n^H b \\ H_n y_n &= \beta_1 u_1 \end{aligned}$$

where $\beta_1 = ||b||$, and hence the candidate solution at step n is:

$$x_n = V_n H_n^{-1} \beta_1 u_1$$

If the residual is orthogonal to the solution space, then for A Hermitian positive-definite the error must be A-orthogonal to the solution space:

$$
\begin{aligned}
r_n = Ae_n \quad &\perp \quad V_n \\
e_n \quad &\perp_A \quad V_n
\end{aligned}
$$

Hence x_n is the result of an A-orthogonal projection onto the solution space, and is thus the vector from the solution space which gives the smallest possible A-norm on its associated error.

3.4.1.2 The minimum residual condition

The minimum residual condition requires that the residual be A-orthogonal to the space from which the candidate solution is taken, so $C_n = AS_n$ giving:

$$S_n^H A^H A S_n y_n = S_n^H A^H b$$

The standard solution space and basis are the same as for the Galerkin condition, resulting in the following reduced system which again relies on the orthonormality of the basis:

$$
\begin{aligned}
V_n^H A^H A V_n y_n &= V_n^H A^H b \\
\underline{H}_n^H V_{n+1}^H V_{n+1} \underline{H}_n y_n &= \underline{H}_n^H V_{n+1}^H b \\
\underline{H}_n^H \underline{H}_n y_n &= \underline{H}_n^H \beta_1 u_1
\end{aligned}
$$

and so the candidate solution is :

$$x_n = V_n (\underline{H}_n^H \underline{H}_n)^{-1} \underline{H}_n^H \beta_1 u_1$$

This immediately gives us that the residual norm is minimised (which is where the method gets its name from), and for A Hermitian positive-definite the A^2-norm of the error for the candidate solution is minimised.

3.4.1.3 The minimum error condition

The minimum error method directly minimises the 2-norm of the error vector, and thus the error is orthogonal to the solution space S_n. In order to achieve this, the candidate solution $x_n \in span(S_n)$ must be chosen from a slightly different space than the two other methods, and this is most easily shown by starting from the orthogonality condition on the error:

$$\begin{aligned} S_n^H e_n &= 0 \\ S_n^H A^{-1} A e_n &= 0 \\ r_n &\perp A^{-H} S_n \end{aligned}$$

The above can be satisfied by generating V_n as an orthonormal basis of the Krylov space $\mathcal{K}_n(A^H, b)$ and choosing the solution space as $S_n = A^H V_n$, giving:

$$\begin{aligned} r_n &\perp A^{-H} A^H V_n \\ &\perp V_n \end{aligned}$$

The residual must be orthogonal to a Krylov space generated by applying the *adjoint* of the operator to b – i.e. $C_n = V_n \in \mathcal{K}_n(A^H, b)$. Combining this choice of Krylov space, solution space and orthogonality condition gives us the following expression which can be solved to find the projected solution y_n:

$$\begin{aligned} V_n^H A A^H V_n y_n &= V_n^H b \\ \underline{H}_n^H \underline{H}_n y_n &= \beta_1 u_1 \end{aligned}$$

and hence the candidate solution:

$$\begin{aligned} x_n &= A^H V_n (\underline{H}_n^H \underline{H}_n)^{-1} \beta_1 u_1 \\ &= V_{n+1} \underline{H}_n (\underline{H}_n^H \underline{H}_n)^{-1} \beta_1 u_1 \end{aligned}$$

3.4.2 Orthogonality conditions and non-orthogonal Krylov bases

The development of the reduced systems in the previous section relies on the orthonormality of the basis for the Krylov space. This does not hold though for the incomplete orthogonalisation or two-sided Lanczos methods.

In the first case, the usual approach is simply to ignore the non-orthonormality of the basis and use the same reduced systems for calculating the projected solution as before. This means that the projected system still constitutes an interface.

The second case is somewhat more ad hoc. For the Galerkin condition, it is possible to simply change the orthogonality condition so that the residual is now orthogonal to the space spanned by the basis of the dual Krylov space of the adjoint of the operator, $C_n = W_n$, which gives us the following:

$$\begin{aligned} W_n^H A V_n y_n &= W_n^H b \\ T_n y_n &= \beta_1 u_1 \end{aligned}$$

The possibilities for adapting the minimum residual condition are less clear cut. The most popular approach is to take the candidate solution that satisfies the same reduced system as before[2], and ignore the fact that it is no longer derived directly from an orthogonality condition and has no direct relation to the norm of the residual – hence it is a *quasi-minimum residual* method. There does not appear to be a well-known (quasi) minimum error method based on the two-sided Lanczos algorithm.

For both the two-sided methods considered here, the projected systems still fit the same mould as those for the orthogonal Krylov basis methods, so the interface is preserved across all variations of the algorithm.

3.4.3 Orthogonality conditions and breakdowns

It may be possible that for a nonsingular indefinite operator the Galerkin condition cannot be satisfied for a given step, in exact arithmetic, even though the Krylov basis vectors are well-defined and the condition is satisfiable at some later step ([64] gives an

[2] That is, it finds y_n such that $||\beta_1 u_1 - T_n y_n||$ is minimised.

explanation based on formally orthogonal polynomials). The unsatisfiability equates to a singular projected matrix in the reduced system. This type of breakdown is not possible for positive definite operators with the Galerkin condition, and cannot happen for the other orthogonality conditions at all, although there is the related concept of stagnation. In finite precision arithmetic, complete breakdown is rare, and the problem tends to manifest itself as numerical instability.

This phenomenon is not critical for the long recurrence algorithms, because steps where a projected solution is not defined can simply be skipped. The problem arises in the short recurrence methods if the algorithm defines the solution at step n in terms of the solution at step $n-1$, and therefore requires the projected operator to be invertible at every step. This is the case for the classic Hermitian–Galerkin method, CG, and the original motivation for using the alternative orthogonality conditions to derive MINRES (which constitutes an improvement when the Galerkin condition cannot be satisfied for an iteration) and SYMMLQ (which does not breakdown) [67].

3.5 Solving the Projected System and Recovering the Solution

The projected linear systems are expressed in terms of the projected operator (either upper Hessenberg or tridiagonal) and the norm of the initial right hand side b. The three orthogonality conditions correspond to three types of reduced system to be solved to obtain y_n, the projected solution. They are summarised in Table 3.1.

Table 3.1: Orthogonality conditions and their reduced systems

Condition	Reduced system	Type
Galerkin	$H_n y_n \doteq \beta_1 u_1$	matrix inversion
Minimum residual	$\underline{H}_n^H \underline{H}_n y_n = \underline{H}_n^H \beta_1 u_1$	least squares
Minimum error	$\underline{H}_n^H \underline{H}_n y_n = \beta_1 u_1$	matrix inversion (on $\underline{H}_n^H \underline{H}_n$)

An orthogonality condition is usually implicitly tied to a particular matrix decomposition. Solving for the Galerkin condition is normally accomplished using the *LU*

decomposition, and solving for the minimum residual and minimum error conditions relies on the QR decomposition (which for the minimum error condition is called the LQ decomposition as the QR factors are transposed). This gives us the set of equations in Table 3.2.

Table 3.2: Orthogonality conditions and the standard decompositions

Condition	Decomposition	Solution
Galerkin	$H_n = L_n U_n$	$y_n = U_n^{-1} L_n^{-1} \beta_1 u_1$
Minimum residual	$\underline{H}_n = Q_n R_n$	$y_n = R_n^{-1} Q_n^H \beta_1 u_1$
Minimum error	$\underline{H}_n = Q_n R_n$	$y_n = R_n^{-1} R_n^{-H} \beta_1 u_1$

For techniques based on the long recurrence solvers we have now assembled all the necessary pieces for a working algorithm. If y_n can be calculated, then the solution x_n can be reconstructed by combining it with the solution space basis vectors derived from the Krylov space process.

There is a problem for the short recurrence solvers though. The vector y_n changes fully at each step of the algorithm, and thus to reconstruct the solution at any step all the basis vectors generated so far must be stored. This means we have lost most of the advantage of having a short recurrence in the first place, which was a small, fixed storage requirement. To remedy this problem, another set of vectors is usually introduced, called the search vectors.

3.5.1 Search vectors

The search vectors are just a grouping of the basis vectors with one of the factors of the projected operator, and the general idea can be demonstrated using the Galerkin condition and the LU decomposition:

$$
\begin{aligned}
x_n &= V_n y_n \\
&= \underbrace{V_n U_n^{-1}}\,\underbrace{L_n^{-1} \beta_1 u_1} \\
&= \quad P_n \qquad z_n
\end{aligned}
$$

where P is the matrix whose columns constitute the search vectors, and z_n is an n-vector that only changes from step to step by adding the next entry. The upper triangular factor U only has two entries per column, the diagonal and the super diagonal, because it is derived from a tridiagonal matrix. Hence, the n-th column of V_n is a linear combination of the last two columns of P_n, and n-th search vector can be constructed from the factor U_n, the n-th basis vector, and the $n-1$-th search vector. This results in another short recurrence:

$$\begin{aligned} V_n &= P_n U_n \\ p_n &= (v_n - U_n[n-1,n]p_{n-1})/U_n[n,n] \end{aligned}$$

The short recurrence for the solution vector is simple:

$$\begin{aligned} x_n &= P_{n-1}z_{n-1} + z_n[n]p_n \\ &= x_{n-1} + z_n[n]p_n \end{aligned}$$

Now that everything is calculable by short recurrences, the whole algorithm has a small fixed storage requirement. The only further complication is that short recurrences mandate that the matrix decomposition is *pivotless* and, for the Galerkin condition, that L^{-1} and U^{-1} exist at every step (see Section 3.4.3). Both facts can affect the stability of the algorithm. Pivoting would still lead to short recurrences [34], but this is nonstandard, and a breakdown due to singular T is unavoidable.

The other short recurrence methods are summarised below. The minimum residual condition with a QR decomposition results in:

$$\begin{aligned} V_n &= P_n R_n \\ p_n &= (v_n - R_n[n-1,n]p_{n-1} - R_n[n-2,n]p_{n-2})/R_n[n,n] \end{aligned}$$

where the upper triangular factor R has two super diagonals, and thus we need to store one extra search vector for the recurrence. In this situation $z_n = Q_n^H \beta_1 u_1$, and note that

Q_n is actually $n+1 \times n+1$, but the $n+1$-th entry of z_n is never used because even if R_n (which is $n \times n$) is extended to have an extra row they would all be zeros.

The standard matrix factorisation for the minimum error condition gives us the following:

$$
\begin{aligned}
x_n &= AV_n y_n \\
&= V_{n+1} Q_n R_n R_n^{-1} R_n^{-H} \beta_1 u_1 \\
&= V_{n+1} Q_n R_n^{-H} \beta_1 u_1 \\
&= P_n z_n
\end{aligned}
$$

where the update of the search vectors is based on the unitary part of the factorisation. Taking $\hat{Q}_n$ as the n-th individual Givens rotation:

$$
\hat{Q}_n = \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & & c & s \\ & & & -\bar{s} & c \end{pmatrix} \in (n+1) \times (n+1)
$$

the search vectors can be derived as follows:

$$
\begin{aligned}
P_n &= V_{n+1} Q_n \\
&= (V_n, v_{n+1}) \, Q_{n-1} \hat{Q}_n \\
&= (V_n Q_{n-1}, v_{n+1} Q_{n-1}) \, \hat{Q}_n \\
&= \left(\hat{P}_{n-1}, v_{n+1}\right) \hat{Q}_n
\end{aligned}
$$

$\hat{P}$ is used above to show that P_n (which has $n+1$ columns) differs from P_{n-1} in its n-th column as well as having an extra column, giving the following relations:

$$
p_{n+1} = s\hat{p}_n + cv_{n+1}
$$

$$p_n = c\hat{p}_n - \bar{s}v_{n+1}$$

where p_{n+1} (and p_n) denotes the $(n+1)$-th (and n-th) column of P_n. Although P_n at step n has $n+1$ columns, z_n only has n non-zero entries, so $x_n = x_{n-1} + z_n[n]p_n$.

3.6 The Common Algorithms

3.6.1 Initial guess

It is possible to exploit prior knowledge as to the likely solution of the system of equations to be solved [73] i.e. some x_1 that is a better approximation to the actual solution than $x_1 = y_1[1]v_1$. The technique alters the problem to a new set of equations for which there is no information, so all cases can be treated by considering how to solve the case where no prior knowledge is available.

3.6.2 Calculating the recurrence residual

As mentioned in the introduction to this chapter, there are various schemes for deciding when to halt an iterative method, and most of them rely one way or another on the 2-norm of the residual vector. The vanilla method that is built into most recipes is to assume that there is some available a priori bound on this value. For both the Galerkin and minimum residual conditions coupled with orthonormal Krylov bases, the 2-norm of the residual is available from the reduced system. For the Galerkin condition:

$$\begin{aligned} r_n &= b - AV_n y_n \\ &= \beta_1 v_1 - V_n T_n y_n - \beta_n v_{n+1} u_n^H y_n \\ &= \beta_1 v_1 - \beta_1 v_1 - \beta_n v_{n+1} y_n[n] \\ &= \beta_n y_n[n] v_{n+1} \\ ||r_n|| &= |\beta_n y_n[n]| \end{aligned}$$

and the scalar $y_n[n]$ can be recovered in the following way:

$$\begin{aligned} y_n &= U_n^{-1} L_n^{-1} \beta_1 u_1 \\ y_n[n] &= z_n[n] / U_n[n,n] \end{aligned}$$

For the minimum residual condition, any method of solving the least-squares problem that calculates the scalar least-squares value itself (such as a QR decomposition) automatically gives us the norm of the residual due to the orthonormality of V_{n+1}:

$$\begin{aligned} r_n &= b - AV_n y_n \\ &= \beta_1 v_1 - V_{n+1} \underline{T}_n y_n \\ &= V_{n+1}(\beta_1 u_1 - \underline{T}_n y_n) \\ ||r_n|| &= ||\beta_1 u_1 - \underline{T}_n y_n|| \end{aligned}$$

Both of these methods carry-over to the non-orthogonal Krylov basis algorithms. For the Galerkin condition the residual now depends on the norm of v_{n+1}, and for the minimum residual condition the scalar value calculated is related to the norm by the condition number of V_{n+1}. It is possible to construct V to have columns with unit norm, or calculate $||v_{n+1}||$ directly, so the residual norm is available for the Galerkin methods, but $\kappa(V_{n+1})$ is not directly calculable so only a *quasi-residual* can be recovered for the minimum residual methods.

The situation is messier for the minimum error condition as the solution space is not the same as the Krylov space. Calculating the residual now requires incorporating the extra application of the operator to the Krylov basis vectors:

$$\begin{aligned} r_n &= b - Ax_n \\ &= b - AA^H V_n y_n \end{aligned} \tag{3.8}$$

For a Hermitian operator, it is possible to exploit the Hermiticity of T and calculate a residual from components of the reduced system. From:

$$\underline{T}_{n+1} \underline{T}_n = \left(\begin{array}{cccc} \multicolumn{4}{c}{\underline{T}_n^H \underline{T}_n} \\ \hline 0 & \dots & \eta_1 & \eta_2 \\ 0 & \dots & 0 & \eta_3 \end{array} \right) \in (n+2) \times n$$

it is possible to derive a convoluted expression for the residual in terms of the last two entries of y_n which themselves can be recovered from z_n and R_n. For a long recurrence method using a non-Hermitian operator, the product AV_{n+1} in 3.8 is unknown, meaning that the residual cannot be calculated from the projected system.

3.6.2.1 Recurrence residual vs. Real residual

While the methods above theoretically give the residual, in practice they suffer from certain numerical problems. Hence, the value calculated is known as the *recurrence residual*, as opposed to the *true residual*, and the difference between these two values is known as the *residual gap*.

Although it is not perfect, the recurrence residual is essentially free, whereas calculating the true residual at each step would require another operator application. Also, calculating the recurrence residual does not require any extra information outside that provided by the reduced system (with the exception of the two-sided Galerkin method which may require $||v_{n+1}||$).

3.6.3 Putting it all together

With two classes of exception (the standard two-term recurrence Galerkin methods (CG, BiCG) and the product methods, discussed in Sections A.1 and A.2 respectively), we can now construct the popular unpreconditioned iterative methods from algorithms that generate and manipulate the pieces that we have presented in this chapter, summarised below:

- Scalars (α, β etc), vectors (b, v_1 etc), dual vectors (w_1 etc) and operators (A).
- The matrices H (or T) and V, generated by an algorithm that satisfies the Arnoldi relation for operator A and initial vector v_1 (and possibly w_1).
- u_1, the (unbounded) first unit canonical basis vector.
- U or R, and z, the products of an LU or QR solve of a reduced system.
- P, generated from a search vector recurrence.

Table 3.3: Constructed iterative methods

Name	Basis Generation	Orthogonality Condition	Decomposition
FOM	Arnoldi	Galerkin	LU
GMRES	Arnoldi	Minimum residual	QR
GMERR	Arnoldi	Minimum error	LQ
DLanczos	Hermitian Lanczos	Galerkin	LU
MINRES	Hermitian Lanczos	Minimum residual	QR
SYMMLQ	Hermitian Lanczos	Minimum error	LQ
BiDLanczos	Two-sided Lanczos	Galerkin	LU
QMR	Two-sided Lanczos	(Quasi) Minimum residual	QR

Note that most of the pieces capture a recurrence relation, and hence are unbounded (i.e. ∞ entries in a vector or columns in a matrix) – it is possible to give a bound in infinite precision arithmetic, but this does not carry over to numerical calculation. The combinations of algorithmic choice that determine the common methods are presented in Table 3.3. The version of GMERR that would result from assembling the pieces presented in this chapter is actually a variation on the original, as mentioned in [72].

3.6.4 Preconditioning

Similarly to having an initial guess for x_1, it is possible to alter the system of equations being solved to improve their numerical properties in some way, e.g. to give faster convergence to a solution. This is known as preconditioning, and can be added to any method. It comes in three varieties:

- Left: $MAx = Mb$
- Right: $AMM^{-1}x = b$
- Symmetric: $MAMM^{-1}x = Mb$

Assuming the termination condition can be suitably adapted, the first two techniques reduce to solving a different set of equations, and, for right preconditioning, a final step

to recover the solution to the original system. Because the generation of the Krylov space only requires operator applications, the new operator can always at the very least be created by function composition.

Symmetric preconditioning can be treated in the same way, but if there exists some $N = M^2$ the method can be simplified to use only one application of N per iteration rather than two applications of M. This trick can be freely captured in the generation of the Krylov space [41].

3.7 Summary

This chapter has given a brief introduction to the iterative solvers, and discussed how common components of the algorithms can be factored out by using the structures of the projected systems as an interface. The rest of the thesis will show how some of these components can be implemented to join together the different algorithmic pieces whilst explicitly representing as much of the structure as possible, and issues in the optimisation of the program arising from the direct representation of that structure in the programming language.

The discussion given above covers what is important for the thesis, but leaves out the relationship to other iterative solvers such as conjugate gradients, and the Lanczos type product methods. For completeness, this is discussed briefly in Appendix A.

Chapter 4

Functional and Algebraic Language Optimisation

This chapter provides some general background on previous work dealing with the compilation of functional languages and MATLAB, and an evaluation of its relevance to Aldor and the iterative solvers. These areas are of interest given that Aldor is both a functional language and has its roots in computer algebra. Even in these fairly restricted domains, the literature on optimisation techniques (and how they relate to general design) is large. Consequently, the work covered here is only a small selection of examples from different topics and projects. Appendix E contain some pointers to material on broader questions, such as choice of appropriate language etc.

Background specific to the optimisations used in this thesis (with an emphasis on imperative work) is supplied in Chapters 7 and 8. It is presented along with a discussion of the relevance of the technique to the framework (as well as the language), and thus comes after the description of the framework design and component implementations in Chapters 5 and 6.

4.1 Compilation of Functional Languages

The majority of the work on functional languages has concentrated on features that are not present in more traditional languages. This is likely due to two reasons. Firstly,

the main applications for these languages are symbolic programs where these features play an obvious role, and secondly it is often assumed that the novel features of these languages are also directly the cause of the main problems for performance. The subjects touched on in this chapter reflect this trend – they are polymorphism, fine-grained function composition, pure languages, higher order functions and the use of recursion, with a discussion of how they affect the related topics of arrays and compiler implementation.

4.1.1 Fine-grained function composition and recursion

As its name suggests, functional programming leans toward the pervasive use of fine-grained functions to structure a program. In addition, some schools of thought regard recursion as the most natural means of phrasing repetitive control flow. This has led researchers interested in optimisation to concentrate on making function calls cheap, and optimising tail recursion under the assumption that they are both common. One example of this being taken to its logical conclusion is the continuation passing style (CPS) intermediate representation [78], in which all control flow is converted to function calls and all functions are tail recursive. In some sense this makes the optimisation of space usage for tail recursive functions from the original source automatic. Another example of the emphasis on recursion is the recasting of traditional scalar optimisations for "loops" arising from simple tail recursive functions [84].

Aldor is at the imperative end of the functional programming language spectrum, so recursive functions are less of an issue. Indeed, generators are an abstraction of control flow and may be safely implemented by either recursion or loops. Fine-grained function composition is still very much an issue however, but the approach in the current compiler is to rely on aggressive inlining rather than lowering the cost of function calls.

4.1.2 Higher order control flow analysis

Functional programming encourages the use of higher order functions. The use of closures can severely complicate the recovery of the function call tree for a program, re-

quiring the use of higher order control flow analysis (HOcfa) [78]. HOcfa has received much attention in the functional programming literature, and some of the techniques have been borrowed by other communities such as the object-oriented languages community (for one example see [86]). Unfortunately, HOcfa is both complex and relatively expensive even for monovariant analyses, with the original 0-cfa of Shivers having $O(n^3)$ complexity in the number of call sites. None the less, various incarnations of the technique have been implemented in various projects, to support various optimisations, including some ambitious frameworks [11].

Two recurring uses for HOcfa are the recovery of types for weakly typed languages such as Lisp or Scheme [78, 76, 100], and the optimisation of closure representations [76, 19], to which it seems reasonably suited. Type recovery is much less relevant to Aldor though, due to static typing and the very infrequent use of true first-class domains, and it is not obvious how much benefit could be accrued from closure analysis. Another mooted application is inlining that can cope with nonlocal (i.e. interprocedural) flow of functions[10, 100], but this is somewhat less convincing than the previous applications. Nonlocal HOcfa is only necessary when direct inlining is cannot be done, but flow directed inlining is only legal in the case where a single function flows to a given call site. This combination appears to reduce its applicability to rather obscure cases, and makes the usefulness of the analysis heavily dependent on particular programming styles and/or the use of an intermediate representation such as CPS, which can affect the availability and ease of recovery of flow information [74].

Several authors from the functional programming community have pointed out that using a direct style compiler (i.e. based on a traditional call stack rather than CPS) and simple inlining tends to get rid of the large majority of higher order functions, thereby removing the need for HOcfa itself, and generally producing better performance (for one example see [84]). This is the approach taken in the Aldor compiler, again implemented by aggressive inlining. With respect to the language, HOcfa may not be suited to the current implementation due to the way the type system is implemented (especially the parts instantiated at run-time – see the explanation of domains in Section 2.2.5), which could make any analysis either overly conservative or computationally very difficult.

Specifically with respect to this work, the information provided by HOcfa has no obvious application with the exception of inlining, and local inlining on its own covers the vast majority of cases, so HOcfa would appear to be overkill even if it is feasible. After suitably aggressive inlining, the use of closures for the iterative solvers is reduced to a handful of cases to combine different recurrences. Within an individual recurrence, which is where the vast majority of computation takes place, all the higher order functions are removed, including those arising from the use of generators (see Chapter 7).

4.1.3 Polymorphism, boxing and modules

In this section, "parametric polymorphism" is used to mean ML-style polymorphism, a simple form of type abstraction that assumes the minimum of information about the objects it deals with – i.e. that objects can be moved, shallow copied, or discarded. The term "module", is used to denote a mechanism to define an abstract data type using a signature of some sort (e.g. an Aldor category).

The simplest way of dealing with polymorphic functions is to require all program objects to be one size, enabling a static calling convention. This is just the uniform representation rule from Section 2.2.3, usually achieved by boxing. A naive approach results in large amounts of heap allocation and prevents the passing of function arguments in registers (especially floatingpoint data), both of which have an associated performance impact. Consequently several authors have looked at the art of unboxing, usually in the context of floatingpoint data types and occasionally including small aggregate types such as pairs or records with some small number of entries.

Approaches to unboxing include:

- **Type passing**, where information about the size and nature of the type is passed as an argument to the function allowing unboxed representations everywhere [84].

- Static insertion of **coercions** (i.e. boxing and unboxing steps) that allow monomorphic functions to take unboxed arguments and ensure that objects are always boxed if necessary before being passed to polymorphic functions [54].

- Leaving all arguments to all functions boxed but using **local unboxing** within functions [55] when the type is statically known.

Of these options, Aldor employs the latter, which is effective provided enough inlining has been done. This approach is acutely sensitive to the effectiveness of the inliner though. Failing to inline functions that take and return boxed scalar arguments within loops with many iterations can easily create enough garbage to seriously degrade performance. For an example, see Section 2.3.1.3, which discusses the case where the emerger doesn't remove all boxing steps – failing to inline the function in the first place is at least as bad. See also the problem discussed in Section 9.3.3.

One alternative to implementing polymorphism is (complete or partial) monomorphisation, for example see [87, 45]. In complete monomorphisation, every polymorphic function is cloned once for each type with which it is used, and calls to polymorphic functions are replaced with calls to their monomorphic counterparts. In common with the other unboxing methods, note that this technique permits unboxing, but does not define how much unboxing ought to be done. While it is difficult to imagine scenarios where unboxing a simple floatingpoint type would be a bad idea, the technique can be extended to flattening larger and/or nested data structures for which the trade-offs are much less clear.

Module language constructs can be treated as an extension of parametric polymorphism, where the functions to manipulate a member of a parameter type as well its unboxed size are unknown without further analysis. Applying the same simple solution again requires uniform size types and thus boxing, and this is the method proposed by some ML compiler authors, mainly in order to allow completely separate compilation of modules. However, full monomorphisation has been used by at least one group to deal with the module language (as outlined in [19]) in the context of a whole program optimisation strategy. Monomorphisation, whether complete or partial, could be an interesting approach for Aldor, but again it is not clear that it is compatible with the type system. Unlike the ML module language, Aldor domains are not completely static (e.g. polymorphic recursion is possible), although in practice most are. Monomorphisation could be used as an enabling step to allow bottom-up inlining in the presence of heavily type parameterised code.

In the context of this work however, unboxing or monomorphisation without inlining is not really of interest as the transformations must always create custom copies of the loops that they manipulate, which equates to inlining the code rather than leaving separate functions intact. Also, a result similar to monomorphisation can be achieved based on Aldor's WPO approach if enough inlining is done for code containing parameterised domains.

4.1.4 Arrays

The representation of arrays in functional languages is directly affected by the implementation of polymorphism. The efficient use of arrays and floatingpoint types is usually considered a lower priority for functional languages, as symbolic code is their bread-and-butter[1]. Some groups however (e.g. the ML community), have considered techniques for arrays in polymorphic languages such as those discussed below.

Naively allowing polymorphic functions to access array elements requires that the elements of the array obey the uniform size rule (i.e. the array consists of pointers to heap allocated objects). This can introduce a large overhead for array operations including pointer chasing and cache effects caused by the scattering of objects throughout the heap, and flat arrays are generally considered much preferable. With regard to the techniques for implementing polymorphism described in Section 4.1.3, static coercions are very expensive (wrapping or unwrapping each element of an array in turn), and local unboxing does not apply as the type of the array elements will not be known. This leaves type passing, which has high overhead for simple function calls, or disallowing the use of arrays with polymorphic functions (e.g. [55]). Aldor is not restricted to these options as it can use its dependent types mechanism to allow local boxing of values derived from flat arrays (and unboxing of values to be stored into an array), thereby allowing them to be used with dependently typed functions. Again though, the efficiency of this scheme is very sensitive to how well the inliner works – executing functions to perform box-unbox steps within loops is typically a disaster –

[1]There are a handful of notable exceptions to this, including SISAL [38], which is a strict pure language that incorporates loop constructs, but does not have higher order functions, polymorphism or a module system (although these were proposed in the literature), and SAC [75], a successor to SISAL.

so in practice the function needs to be inlined and have the boxing steps removed by optimisation.

4.1.5 Pure languages and the management of state

Languages without side-effects (i.e. pure languages) are an interesting subclass of functional languages. The lack of side-effects means that a compiler that hopes to produce efficient executables ought to do some kind of analysis of the lifetime of objects to be able to perform destructive updates where possible and limit the pressure on the garbage collector, and a run-time system with a garbage collection mechanism that is as efficient as possible [97, 98]. Analysing the lifetime of objects with dynamic extent is far from easy however (and may need a HOcfa analysis [77]), so pure languages tend to rely on a garbage collector and potentially suffer from weak performance as a result. An interesting refinement of garbage collection that uses some static analysis is the concept of region inference [46]. One alternative to analysing programs is to introduce exotic type systems that allow destructive updates (e.g. [92, 101]).

The relationship of pure languages to Aldor exists through its use of techniques such as garbage collection, the environment emerger optimisation (which analyses and limits the dynamic extent of objects), and what are usually pure functions attached to overloaded infix operators (see Section 2.1.6). The more extreme problems of pure languages are avoided however, as Aldor permits the destructive updating of single elements of an aggregate, and a programmer can limit the lifetime of objects by using the `dispose!` command. As such, the more sophisticated techniques developed for pure languages are not necessary, with the assumption that the programmer can intervene when efficiency is important.

A further specialisation of pure languages is the class of lazy (or normal order) languages[2] such as [1]. Aldor is related to these by the generator construct, which constitutes programmer specified laziness, and the type system, where the run-time representations of types are lazily instantiated. It may be possible that techniques from the lazy community, such as strictness analysis, are applicable to these aspects of the language, but this was not directly pursued.

[2]It may be possible to have non-pure lazy languages, but this would be an unorthodox combination.

4.1.6 Compiler implementation

Functional language compiler writers have employed many methods over the years. The most prevalent are direct compilation to machine code [4, 3] and compilation to some low-level language, typically C [85, 25]. Going direct to machine code allows the implementation of techniques that do not sit well with an intermediate language. Examples of these include precise tracing garbage collection, exceptions, custom function invocation methods, intermediate representations such as CPS, and optimisations such as avoiding stack frame allocation for tail calls. Also, the issue of precise tracing garbage collection can interact with code generation strategies to support polymorphism and unboxing. The run-time system must be able to identify exactly the live root set in the call stack, and when the stack frames may contain unboxed objects this implies providing some kind of precise descriptor for each frame. This in turn favours direct machine code generation, as the compiler of an intermediate language (such as C) is frequently free to arrange stack frame layout as it sees fit, and information on the final layout is very difficult to get at. However, compiling directly to machine code requires a lot of effort, especially if multiple back-ends are to be supported.

Conversely, going via C gives portability and offers some degree of optimisation for free. This comes at the cost of conservative garbage collection, and a poor mapping of certain techniques on to the C function call model. These two disadvantages have spawned projects that aim to avoid them whilst providing the labour saving advantages of targeting C. For compilation straight to machine code there exist compiler kits that support techniques popular from functional programming and machine specific low level optimisations (e.g. [2]), and the `C--` project [49] aims to provide a C-like target language to give portability and ultimately a common set of optimisations, alongside such features as standardised exceptions and formalised stack frame layout rules to allow precise garbage collection.

As described in Chapter 2, Aldor compiles to C and uses a conservative mark-and-sweep collector supplemented with optional deallocation hints. While the other routes to machine code mentioned above may be useful in future, currently it is not clear what advantages they will bring for the extra implementation cost. In addition, the issues they tackle are low level and quite general, and therefore not particularly relevant to

the application at hand (i.e. the iterative solvers).

4.1.7 Fusion and loop restructuring in functional languages

Most previous work on compilation and intermediate representations for functional languages has not borrowed the loop restructuring techniques from traditional imperative scientific computing. One interesting exception to this sought to join the OCaml compiler with SUIF [24]. However, it appears to be a very small project, and only considers pre-existing loops in the code which are translated in isolation. This ignores the impact of modules and fine-grained function composition, and does not deal with interloop locality which is the main thrust of the optimisations in this thesis.

Fusion has received some attention in the functional language community. Two examples are deforestation [91], a form of fusion for lists, and fusion of array combinators [20]. The authors of the latter example have also previously worked on optimisations for an intermediate representation of a parallel functional language. The original scheme (from a different group) compiled to an intermediate language with primitives for operating on arrays that was implemented with individual native function calls for each primitive [14]. Problems with the performance of code using this approach prompted the authors of [20] to attempt to break the abstraction barriers introduced by the intermediate language and combine primitives together, including using fusion [50], although the exact mechanism is not specified. The optimisations in [50] were performed by hand, and those in [20] are implemented for a lazy functional language using static rewrite rules on the source.

These techniques perform fairly limited fusion, with the authors claiming in [20] that deforestation does not work well for fusing functions that consume more than one list, and although their work covers multiple arguments there is no mention of how they would approach collective loop fusion. As such, it appears that there is some way to go before these techniques reach the level of sophistication found in the imperative community.

4.2 Compilation of Numerical Computer Algebra Systems

This section concentrates on the compilation of MATLAB, which can be viewed as an interactive array language or a computer algebra system with a bias toward numerical linear algebra.

The FALCON project has covered various techniques. The first batch [28] were mainly concerned with static analyses to try and reduce the interpretation overhead of the dynamic features of the language, and a subsequent translation to Fortran 90. These features include dynamic typing, where type information includes number type (logical, integer, real, complex), rank type (scalar, vector, matrix), extent (i.e. the dimensions of non-scalar variables), and structure type for matrices (square, triangular, diagonal, Hermitian etc.). The language also uses dynamic resizing of arrays and array bounds checking. These problems are not directly relevant to Aldor as it is a statically typed language that does not mandate array bounds checks.

The MATLAB environment includes a large number of built-in routines that are called by the interpreter for operations on vectors and matrices, and some of these routines are implemented as optimised native binaries. In the first set of optimisations no attempt was made to break this abstraction and fuse together components. The second batch [58] included restructuring, but the system was based on pattern matching and was at least partly interactive. The main idea was to allow a developer to explore algebraic transformations at the level of the source code rather than performing traditional optimisations such as fusion, although they get a brief mention. Again, no real attempt is made to break the abstractions of the library routines, although support is added to match expressions in the abstract syntax tree and replace them with different library routines such as native BLAS. This can be likened to other work that manipulates library routines as language primitives based on some programmer supplied rules (for example [44]). A follow-up piece of work [63] suggests adding more source-to-source transformations to the FALCON framework, but, as before, the library routines are treated as primitives. As such, they ignore the cost of modularity and sidestep the optimisation issues that we wish to consider.

The Menhir project [22] added directives to the language and targeted alternative standard libraries, including parallel implementations such as ScaLapack. Again, they do not appear interested in restructuring.

4.3 Summary

The principal result of the survey in this chapter is that many of the problems traditionally tackled by functional language optimisations are either adequately handled by techniques already implemented in the compiler, or are not directly relevant to Aldor. In addition, authors from the functional language and computer algebra language communities have only fleetingly considered the impact of modularity for work intensive loop based numerical codes. This can be contrasted with the amount of research done in the imperative language community on tackling the costs of fine-grained structuring (such as collections of loops).

Chapter 5

Algorithm Framework

This chapter describes the design of the algorithm framework written in Aldor, based on the approach discussed in Chapter 3. The framework can be used to add together the various algorithmic pieces to form any of the unpreconditioned solvers listed in Section 3.6.3. The chapter begins by discussing the hierarchy of categories. An individual category describes the interface to a class of objects, and the relationships of inheritance from and parameterisation by other categories capture the structure of the framework. This is followed by a description of some example domains that implement the categories to give an instantiation of the framework, often using abstractions provided by other parts of the framework that will in turn be implemented by other domains etc.

This chapter is supplemented by appendices C and D, which contain more detailed code extracts.

5.1 Category Hierarchy

The first task in the design of the framework was the construction of categories to capture as much of the structure discussed in Chapter 3 as possible. Categories are related by inheritance and by taking members of other categories as parameters. In terms of the inheritance relation, there are three main groups – the pre-existing categories from `axllib` (see Section 2.1.4) that capture basic algebraic features [93], the *linear alge-*

bra categories that build on these to provide a richer structure, and the *problem specific* categories that are less purely mathematical. The first two groups are used to type the pieces presented in Section 3.6.3, and the last is used to capture mappings between them.

Each category in a chain of inheritance is intended to capture some additional structure not present in its ancestors. This usually means defining some extra operations on the category, or some extra structure on the parameters to the category, but occasionally it is used to capture information that is more abstract. An example of this is the category for Hermitian operators, which is intended to convey some extra information about the type that cannot fully be captured by the type system (i.e. the Hermiticity of the operators).

The categories and their relations are summarised using diagrams. Figure 5.1 represents the inheritance relationship between some root members of `axllib` and the linear algebra categories; Figure 5.3 shows the handful of problem specific categories that are related by inheritance; and Figure 5.2 shows the parameterisation relationship (i.e. which categories are parameterised by domains belonging to other categories). In the latter diagram trivial arcs have been removed – when a category is parameterised by domains that are typed by parameterised categories, dependent typing requires that it take all the parameters from its parameterised arguments as additional arguments, and representing all of these in the graph would make it unreadable. However, when a parameter that would be required by dependent typing is actually a subtype of the most general one allowable for the other parameterised arguments, or when multiple domains of a certain type are required, only one of which is necessary for dependent typing, then the arc remains as it carries nontrivial information. Examples of these two cases are the arc from `NormedLinearSpace` to `KrylovSpace` as `LinearOperator` only requires a parameter of type `LinearSpace`, and that from `IndexedVector` to `DirectQRSolve`.

Sections with descriptions of some of the more interesting categories follow, and details of the category exports can be found in Appendix C.

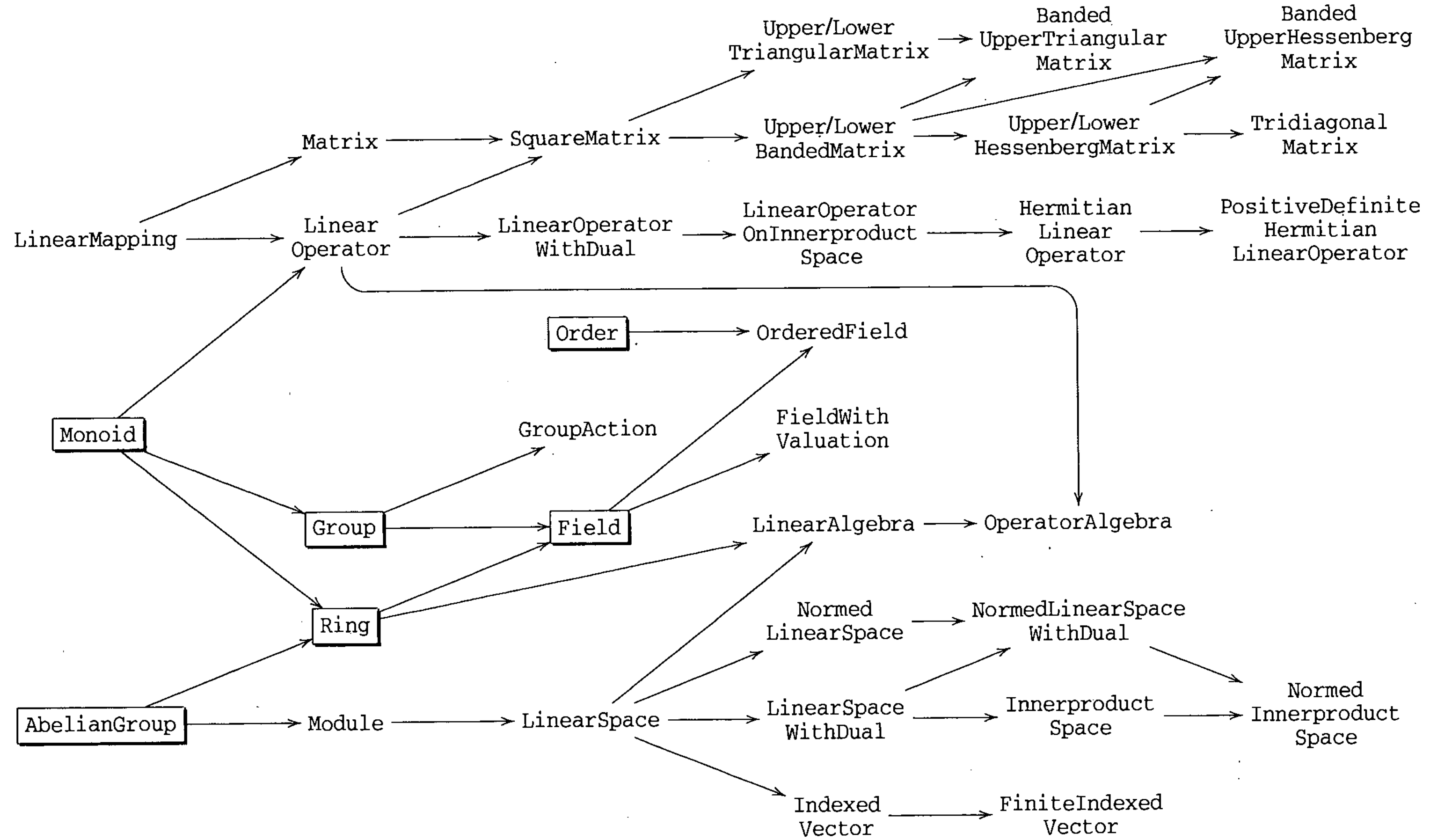


Figure 5.1: Category inheritance diagram for the linear algebra categories. Pre-existing members from `axllib` have borders and shadows.

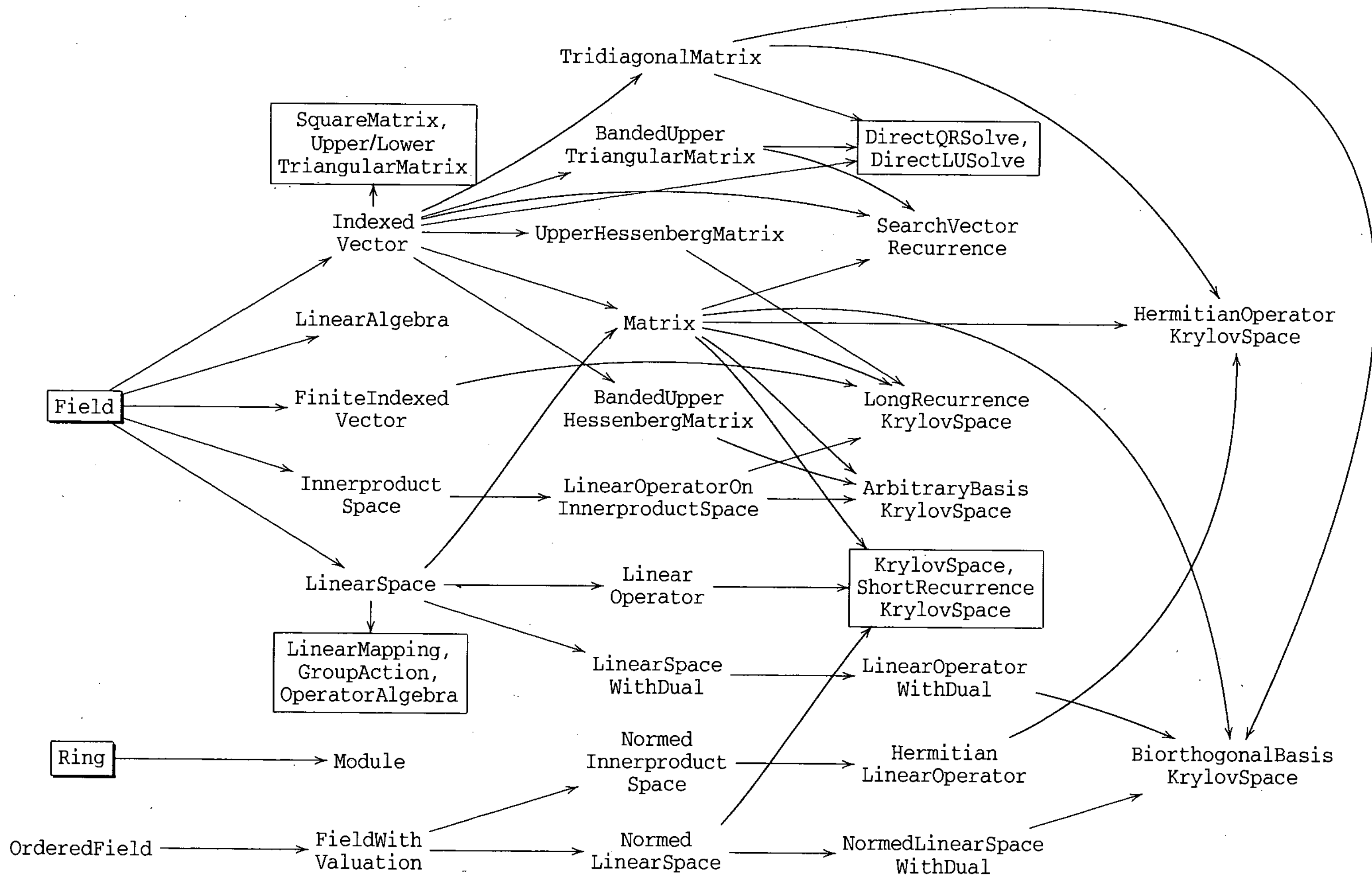


Figure 5.2: Category parameterisation diagram. Pre-existing members from axllib have borders and shadows, elements representing multiple entries have borders.

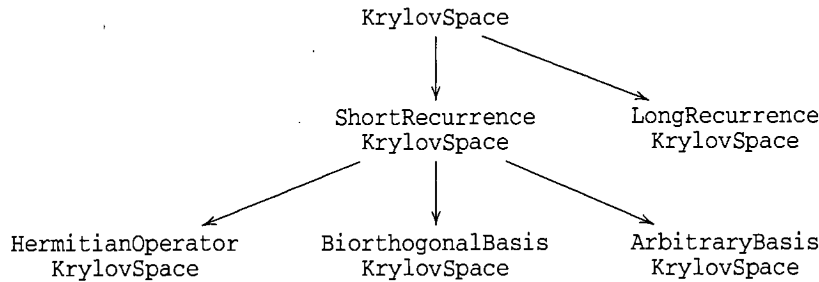


Figure 5.3: Category inheritance diagram for the few problem specific categories that are related by inheritance.

5.1.1 Linear algebra categories

These categories embody various constructs from linear algebra and their interrelationships.

5.1.1.1 `FieldWithValuation`

This is a (possibly unordered) field with an Archimedean valuation operation that maps elements to members of an ordered field. This category is used to provide a norm for both real numbers and complex numbers in this thesis, but it may generalise to other fields and valuation operations. The valuation domain is not restricted to being positive in order to be able to capture some extra structure in the algorithms at the cost of a slight abuse of terms. This is explained in the description of the `HermitianLinearOperator` category.

Having the valuation as a separate type means that we can specify different operations when using the result of a valuation in contrast to some arbitrary member of the ground field. This allows certain operations to be more efficient, for instance, multiplying a vector of complex numbers by an arbitrary member of the ground field (i.e. a complex scalar) requires more arithmetic than multiplying the vector by a member of the valuation, which only has a real component. Because the information is encoded statically, there is no need for dynamic tests at run-time.

5.1.1.2 `LinearSpaceWithDual`

Although all linear spaces have a mathematical dual, in terms of a concrete program the dual may not always have been implemented. This category captures the relationship with the dual if it has been defined.

5.1.1.3 `NormedLinearSpace`

The ground field parameter to this domain must be a field with a valuation. This allows us to capture the relationship between a linear space over a possibly unordered scalar field and its metric. This category exists separately from that for inner product spaces, which automatically define a norm, as the converse is not necessarily true – this is analogous to the distinction between Banach spaces and Hilbert spaces.

5.1.1.4 `InnerProductSpace`

The name of this category is an abuse of terminology, as a true inner product automatically defines a norm. The definition of norms here requires a ground field with a valuation (see below), so this category can be used to provide some kind of inner product for a ground field without an explicitly supplied valuation.

The category denotes a vector space that is dual to itself, and is recursively defined using the `LinearSpaceWithDual` category (see Section 5.3.1).

5.1.1.5 `NormedInnerProductSpace`

Here the norm is implicitly provided by the inner product, so the ground field must be a field with valuation in order for the norm to provide a metric.

5.1.1.6 `GroupAction`

A group action is a group whose members can act as operators on a linear space. Note that this category does not inherit from the linear operator category, as not all operator groups are closed under the operations that that category provides e.g. scalar multiplication.

5.1.1.7 `OperatorAlgebra`

A linear algebra is a linear space whose elements also form a monoid with respect to some operation. This category can be used to type certain domains of operators (such as a domain to represent any general linear transformation) that form a linear algebra, but not all useful domains of operators automatically form a linear space as they are not all closed under all the operations. For example, a domain of nonsingular matrices does not contain the 0 matrix, and hence is not closed under addition (and nor is it a proper linear space). As such, the category of linear operators does not automatically inherit from the linear algebra category, and this category joins the two chains.

5.1.1.8 `LinearOperatorWithDual`

A linear operator on a finite vector space with a dual automatically defines an operator on the dual space by the definition of linear functionals, and this is captured in a category default.

`NormedLinearSpaceWithDual` is used in conjunction with this category to structure the non-Hermitian short recurrence solvers, as they capture the separation between the basis of the original Krylov space and its dual.

5.1.1.9 `LinearOperatorOnInnerProductSpace`

Given that an inner product space is self-dual, a linear operator on the space now defines two actions. These are the usual action and the action on a vector when it is interpreted as a member of the dual space. This is captured in the category by defining left multiplication by the operator, and also by defining an adjoint operation.

This category is used with `NormedInnerProductSpace` to structure the non-Hermitian long recurrence solvers, because the inner product of the vector space introduces the extra necessary structure for the orthogonalisation step.

5.1.1.10 `HermitianLinearOperator`

This category is a specialisation of a general linear operator on an inner product space, and is used to structure the Hermitian short recurrence solvers. Its ground field must

have a valuation to capture the type issues with the quadratic forms resulting from the Hermitian forms associated with operators. A quadratic form derived from an operator A is taken as defining a possibly indefinite norm-like function (resulting from a possibly indefinite inner product $(.,.)_A$), which has the same type as the valuation of the ground field, but, due to indefiniteness, may also be negative. This potential negative value is an abuse of the definition of the valuation as the domain of values resulting from the norm of the ground field, as mentioned earlier, but is useful to capture information about the projected system (see Section 5.1.2.1).

Given that the operators are self-adjoint, the adjoint operation is defined here as being empty by means of a category default.

5.1.1.11 `PositiveDefiniteHermitianLinearOperator`

This is a simple extension of `HermitianLinearOperator`, with a curried function to define norms using the quadratic form of the operator.

5.1.1.12 `Matrix, IndexedVector, SquareMatrix`

The `Matrix` category is used to type linear mappings that can be decomposed into column vectors. It is used in conjunction with the `IndexedVector` category, which types domains that provide a function to access individual elements. Neither category is meant to imply that elements of an adhering domain have finite extent – i.e. matrices can have an infinite number of columns, and vectors may have an infinite number of entries. To define a way of computing the effect of a linear mapping by a linear combination of its column vectors, a further category of `FiniteIndexedVector` must be used. This avoids problematic termination issues.

`Matrix` is extended by `SquareMatrix`, a category that decomposes a linear operator into its individual scalar elements. "Square" in this case is intended to denote that it is an explicit linear operator (i.e. it concerns the type of the vectors involved), but elements from an adhering domain can be treated as rectangular matrices by placing limits on which entries are used. Hence $\underline{H}$ and H from Chapter 3 are really the same unbounded matrix, and the difference lies in how they are manipulated by the solves of the projected system.

`SquareMatrix` is the root of subtypes such as upper/lower triangular, upper/lower Hessenberg and tridiagonal matrices. The `Banded` specialisations of Hessenberg and triangular matrices are used to type the upper Hessenberg matrix resulting from an Arnoldi-type relation for an incomplete orthogonalisation method, and the U or R factor resulting from the LU or QR decomposition for a short recurrence.

5.1.2 Problem specific categories

The modular structure of the iterative solvers is captured by defining mappings between the separate pieces described in Section 3.6.3. These mappings are defined in categories that are more problem specific and less mathematical in nature than those previously mentioned. The long recurrence solvers are split into two steps – firstly the generation of the Krylov basis vectors and projection of the operator on to that basis, and secondly the factorisation of the projected matrix. The short recurrence solvers are split into three steps. The first two are the same as those from the long recurrence solvers, with the additional step being the generation of the search vectors and the updating of the solution vector based on them.

5.1.2.1 Interface to Krylov space object

The most significant subsection of the algorithms is the generation of the Krylov space, which defines a mapping from an operator and a vector to an object consisting of a linear mapping and a matrix of scalars. The linear mapping is conceptually the matrix whose column vectors are the sequence of Krylov space basis vectors, and the matrix of scalars is the upper Hessenberg matrix of coefficients that represents the operator projected onto the Krylov basis vectors and orthogonally to some other set of vectors. These are the matrices V and H (or T) from one of the Arnoldi relations, so the Krylov space objects define the mapping $(A, v_1) \mapsto (V, H)$, with possibly some extra arguments (such as the dual start vector w_1 for a biorthogonal Krylov space).

Derived from the general `KrylovSpace` category are two sub-categories, one for the long recurrence Arnoldi-type algorithms, and an abstract parent for the short recurrence Krylov spaces. This latter category is not meant to be directly used, but is further specialised into three sub-categories, one each for incomplete orthogonalisation-type

algorithms, Hermitian Lanczos-type algorithms, and non-Hermitian Lanczos-type algorithms (see Figure 5.3). These categories differ in the types of their parameters, and the types of the matrices that the Krylov object generates. The short recurrence category exists as a separate entity to give somewhere to put the template algorithm default, which provides structure common to all the short recurrence methods, and to provide a means of specifying different types for the ground field of the linear spaces in question and the entries of the matrices used in the reduced system. Usually both are the same, but a Hermitian operator over $\mathbb{C}$ gives rise to a tridiagonal T whose entries are elements of $\mathbb{R}$, and this can be captured here.

The sequence of Krylov basis vectors is infinitely long, and as such the implementation of the linear mapping and matrix of scalars require some nonstandard techniques. However, the only part of this that is visible at the category level is the fact that one domain parameter for the long recurrence Krylov space must be finite length indexed vectors.

5.1.2.2 The template functions

A template for an iterative solver algorithm is provided for each immediate subcategory of `KrylovSpace` as a category default, with the intention of providing a useful amount of structure common to the vanilla algorithms. Missing parts of the algorithm are provided as function arguments to a template with the result being a complete iterative method. The long recurrence method template uses the Krylov space object constructor provided by its category. Because the short recurrence category is the ancestor of several categories whose constructor functions have different names and different signatures, its template function takes the constructor as another function parameter.

5.1.2.2.1 Long recurrence template The template for the long recurrence methods (fig. 5.4) requires a function that maps the projected operator and the scalar factor in the right hand side of the reduced system to the projected solution – that is $(H, \beta) \mapsto y$. This function incorporates a solve by some decomposition of the relevant projected system of equations for the different orthogonality conditions given in Section 3.4, along with some halting criteria to determine when the tentative projected solution is

```
iterativeSolve(correction : (HDom, Valuation) -> yDom)
               (A : Operator,
                x : Vector,
                b : Vector) : Vector == {

   if x = 0
      then r := b;
      else r := b - A x;
   rNorm : Valuation := norm(r);
   K := orthonormalKrylovBasis(A, r/rNorm);
   H := coefficients(K);
   V := basis(K);
   y := correction(H, rNorm);
   x := x + V y;
   return x;
}
```

Figure 5.4: Aldor code for the long recurrence method template. Note that the domains in the function signature (i.e. `HDom`, `Valuation` etc) are parameters to the category itself. For further details see Appendix C.

good enough. Once the projected solution has been found with this function, the actual solution is reconstructed using the matrix of basis vectors and returned as the result.

Halting conditions based on estimates of the residual derived from the reduced system exist for the standard *LU*–Galerkin and *QR*–minimum residual algorithms (see Section 3.6.2). However, the template is not such a good match for the minimum error algorithm for two reasons. Firstly, it is not clear if it is possible to calculate a recurrence residual from the projected system, and secondly the result returned by the assembled template is no longer the actual solution – it must be recovered by multiplying the final vector once more by the operator A.

5.1.2.2.2 Short recurrence template The template for the short recurrence methods (fig. 5.5) requires (in addition to the Krylov space constructor) a function that takes the projected operator, matrix of Krylov basis vectors and projected right hand side and

```
iterativeSolve(krylovBasis : (Operator, Vector) -> %,
               correction : (HDom, VDom, Valuation)
                               -> (VDom, yDom, SI -> Boolean))
              (A : Operator,
               x : Vector,
               b : Vector) : Vector == {

    if x = 0
       then r := b;
       else r := b - A x;
    rNorm : Valuation := norm(r);
    K := krylovBasis(A, r/rNorm);
    H := coefficients(K);
    V := basis(K);
    (z, P, lastIteration?) := correction(H, V, rNorm);
    for i in 1.. repeat {
       xNew := x + z(i) * P(i);
       dispose!(x); x := xNew;
       if lastIteration?(i) then break;
    }
    return x;
}
```

Figure 5.5: Aldor code for the short recurrence method template. Note that the domains in the function signature (i.e. `Operator`, `Vector` etc) are parameters to the category itself. For further details see Appendix C.

maps them to the z and P factors that combine to give corrections to the solution, and a function that signals termination – i.e. $(H,V,\beta) \mapsto (z,P,$`lastIteration?`$)$ where `lastIteration?` is a function from a step n to a Boolean : $n \mapsto (true|false)$. In this template, the body of a loop updates the solution vector and then tests the function to see if the algorithm has converged. If it has, the loop ends and the solution is returned.

The functions passed to the templates are composed of further pieces such as matrix decomposition and search vector recurrences, along with some glue code to assemble them. These pieces have their own interfaces, described below.

5.1.2.3 *LU* and *QR* decompositions

The interface to the pivotless *LU* decomposition, `DirectLUSolve`, provides a function that maps an upper Hessenberg matrix and an initial right hand side to the right hand factor, and a vector that is the result of the other factor being inverted and applied to the initial right hand side – i.e. $(H,\beta) \mapsto (U,z)$.

The *QR* decomposition, `DirectQRSolve`, generates in addition a vector of running residual values, which is used as the recurrence residual in the minimum residual methods – i.e. $(H,\beta) \mapsto (R,z,res_n)$. Note that there is no notion either of the size of the matrix to be decomposed or of the size of the resulting factors. Again, this is done deliberately to be able to deal with an iterative process that generates arbitrary length vector sequences and thus an arbitrary size projected matrix.

Both categories use the most general type for their possible parameters, in order to be applicable to all types of decomposition. Further specialisations, e.g. to banded matrices, can be specified in the type requirements for parameters to domains.

5.1.2.4 Search vector recurrence

The search vector recurrences for the short recurrence methods associated with the *LU* and *QR* decompositions are packaged in a similar way to the matrix of basis vectors from the Krylov space. They are simply a mapping from a upper triangular upper banded matrix and a matrix of basis vectors to a matrix of search vectors, where the size of all the matrices involved is unbounded – i.e. $(U|R,V) \mapsto P$. Although the triangular factor is required to be upper banded, the size of the band is not specified

and so the category applies to all search recurrences of this form, regardless of length.

5.1.2.5 LQ decomposition and search vectors

The solution of the projected system for the minimum error condition and the associated search vector update are less easily separated. The search vector update is based on the individual rotations calculated from the decomposition rather than a composition of all the factors. Consequently, no categories are provided to structure this part of the algorithm. Also, there is not the same degree of similarity between the decompositions used for the short and long recurrence methods, making the separation less important, as implementations cannot be reused.

5.2 Domain Implementation

After specifying the categories to define the important interfaces within the family of applications as a whole, suitable domains still have to be written to implement them. The following section discusses important points in the implementation of the algorithms at the domain level. A much more detailed (but still slightly abridged) listing is given in Appendix D.

The running examples in this section are the domains that are used to construct a version of the QMR algorithm. They include a two-sided, short recurrence Lanczos process based on [42], a pivotless QR decomposition for a tridiagonal matrix based on Givens rotations, and a recurrence for updating the QMR search vectors. Note that the domains themselves are parameterised over the objects that they manipulate (scalars, vectors etc) and thus make full use of the generality provided by the category hierarchy for maximum flexibility (for further details see the solver domains in Appendix D).

5.2.1 Index functions, recurrences, and infinite sequences

As discussed in Section 2.1.6, the standard array indexing functions can be overloaded using the `apply` mechanism. The indexing function to retrieve column vectors from a matrix is packaged using this, and hence these operations have the same syntax. This

```
delta := (v1 * w1);

u := A v1;
alpha := (u * w1)/delta;
beta := 0;
v2 := u - alpha * v1;
gamma := norm v2;

state := 1;
```

Figure 5.6: Aldor code for the first step of the Lanczos recurrence. Phonetic Greek letters (e.g. `delta`, `alpha` etc) indicate elements from a scalar domain, a single lowercase letter and a number indicate elements from a vector domain (e.g. `w1`, `v2` etc) and uppercase letters indicate an element of the operator domain (e.g. `A`). The types of these objects are inferred by the compiler. The integer state variable is called `state`. For further details see Appendix D.

enables a column index function for an "infinite" matrix – the index function, which maps an integer to a column vector, is actually linked to a recurrence and the integer argument is the number of steps to take to generate the correct column. In other words, infinite matrices are implemented lazily.

5.2.2 Krylov space recurrence

The algorithm for generating the Krylov space basis vectors and projected operator is written in a textbook style, with some wrapping around it to be able to package its results as lazy matrices, and some directives to manage storage. The domain containing it is typed with the `BiorthogonalKrylovSpace` category. The code for stepping the recurrence itself is split into two functions. The first of these is used as the initial step of the sequence, and the second is used for all subsequent steps. They are presented (with some less important lines deleted for clarity) in Figures 5.6 and 5.7 respectively.

The objects that the recurrence functions manipulate are lexically scoped variables from their environment. These variables hold the current state of the recurrence. To

```
t1 := AH w1 - conjugate(alpha) * w1 - conjugate(beta) * w2;
dispose!(w2); w2 := t1;
(deltaOld, deltaTemp) := (delta, (v2 * w2));
delta := deltaTemp / (gamma * gamma);

(v1, v2) := (v2/gamma, v1);
(w1, w2) := (w2/conjugate(gamma), w1);

u := A v1;
alpha := (u * w1)/delta;
beta := gamma * delta / deltaOld;
t2 := u - alpha * v1 - beta * v2;
dispose!(v2); v2 := t2;
gamma := norm v2;

state := state + 1;
```

Figure 5.7: Aldor code for the general step of the Lanczos recurrence. Phonetic Greek letters (e.g. `delta`, `alpha` etc) indicate elements from a scalar domain, a single lowercase letter and a number indicate elements from a vector domain (e.g. `w1`, `v2` etc) and uppercase letters indicate an element of the operator domain (e.g. `A`). The types of these objects are inferred by the compiler. The integer state variable is called `state`. For further details see Appendix D.

get the n-th step of the recurrence, the function for the initial step is called once, followed by the function for a general step $n-1$ times. After this the required state is read off from the lexical variables. By introducing a small wrapper that performs precisely this procedure, we now have a function from an integer denoting the step to the values produced by that step. There are two of these recurrence functions, one for producing the triplets of scalars that constitute the nonzero entries of the columns of the tridiagonal projected operator T, and one for the basis vectors V, and they both call the same stepping functions.

5.2.2.1 Use of state for the Lanczos process

It is inefficient to start the recurrence from scratch every time a vector from the sequence is required, especially if the indices $(i_1, i_2, \ldots i_n)$ of the requested vectors form an increasing sequence – that is $i_{n-1} \leq i_n \leq i_{n+1} \forall n$. To avoid doing unnecessary work an improvement is to leave the state as it is after a vector has been requested, and record the step at which the recurrence stopped.

Upon receiving a request for another vector, some surrounding code first checks the index of the request against the current step value. If the index of the request is greater than (or equal to) the current step, then the recurrence can be cycled the necessary number of times starting from the current state. If the index is lower than the current step, then the code re-initialises the state and cycles the recurrence up to the necessary step.

In addition, given that one recurrence produces both the vectors and the scalars, the two functions that produce either the vectors or the scalars can share the same state to reduce the work further. This particular caching and sharing strategy is algorithmically tuned to an expected sequence of requests through both functions, being the likely sequence of requests during a linear system solve. Hence, it is a domain implementation issue. Other domains could be written to incorporate alternative caching policies, either from scratch or as a wrapper around this domain.

5.2.2.2 Domain representation

The domain representation is a record of the two recurrence functions wrapped as matrices (see below), the operator A, and copies of the initial vectors v_1 and w_1. Having a concrete representation means that the domain can provide functions that return the operator and the initial state used to start the Krylov space recurrence. This is not provided for the other recurrences as it is deemed less necessary – their constructors directly return elements from different domains rather than an object which can yield these elements as is done here.

5.2.3 Matrix of basis vectors

The domain implementing the matrix of basis vectors V is used as a thin wrapper around the recurrence function from the Krylov space domain, and typed by the `Matrix` category. The domain representation is an arbitrary function from an integer to the required vector type. The constructor takes the recurrence function argument and simply obscures its type. The function to retrieve a column of a matrix takes its integer argument and passes it to the underlying representation. Matrix-vector multiplication with a finite vector (i.e. of bounded size) is computed as a simple linear combination of the vectors as they are produced by the recurrence, but this is only used for the long recurrence solvers.

Although this domain is currently very simple, it could usefully be extended with its own caching policy. For example, if the matrix were required as part of a Lanczos process for eigenvector approximation [40], the domain implementation could keep copies of all the vectors as they were produced. This is certainly not desirable for short recurrence linear systems solvers however, as they are specifically designed so that only a small fixed number of Lanczos vectors have to be kept.

5.2.4 Tridiagonal matrix of recurrence coefficients

The tridiagonal matrix domain for T (typed by the `TridiagonalMatrix` category) is similar to the domain for the matrix of basis vectors. This "matrix" is explicitly constructed from scalars though, and supports a two dimensional indexing function – i.e. it maps a pair of integers to an entry in the matrix. Given that the Krylov space recurrence produces triplets of scalars, some extra work is necessary to produce the indexing function. It first checks that the pair of integers indexes an element within the central band – if they do, then the function uses the recurrence to produce the necessary scalar, and if not it returns a zero. Functional versions of columns or rows can be produced by currying this index function over one of its arguments, provided there is a domain that will take this function as an argument to a constructor.

Although this domain currently supports no caching, in some sense it is the most obvious candidate. It is very cheap to store the scalars as they are produced, and the

```
state := state + 1;

R.u2 := sOld * T(state - 1, state);
u1Temp := cOld * T(state - 1, state);

R.u1 := (c * u1Temp) + (s * T(state, state));
dTemp := (c * T(state, state)) - (conjugate(s) * u1Temp);
cOld := c;
sOld := s;

(c, s, r) := givensRotation(dTemp, T(state + 1, state));
R.d := r;
z := (c * zTemp) + (s * y(state+1));
zTemp := (c * y(state+1)) - (conjugate(s) * zTemp);
```

Figure 5.8: Aldor code for the general step of the QR solve on T. Lowercase letters indicate elements from either a scalar or a vector domain (e.g. `c` and `s` are scalars, and `y` is a vector etc) and uppercase letters indicate an elements of a matrix domain (e.g. `T` is a tridiagonal matrix, and `R` is a record that is used to construct a banded upper triangular matrix). The result of indexing into a matrix or a vector (e.g. `T(state + 1, state)` and `y(state + 1)`) is a scalar. The types of these objects are inferred by the compiler. The integer state variable is called `state`. For further details see Appendix D.

tridiagonal matrix is all that is needed to produce approximations to the eigenvalues of the original operator [40].

5.2.5 QR decomposition

The pivotless QR decomposition, typed by `DirectQRSolve` is used to produce a lazy banded upper triangular factor (i.e. R) and two lazy vectors. The first is the result of the unitary factor being applied to the original right hand side $z = Q\beta_1 u_1$, and the second is a lazy vector whose n-th element constitutes the 2-norm of the residual resulting from the least-squares solution at step n by limiting the problem to be of size $(n+1) \times n$.

The decomposition essentially defines a recurrence, and it is handled in much the same way as the Krylov space recurrence. The code for the general step is given in Figure 5.8 (code for the first two steps is omitted as it is much the same). The decomposition only has to maintain a small number of scalars in its environment, unlike the Lanczos recurrence which requires the storing of a small number of vectors. The three objects produced by the recurrence are also wrapped in much the same way as the matrix of basis vectors and the tridiagonal matrix of coefficients.

The current implementation is specialised to the solve for QMR, but a more general procedure could easily be used instead. For example, it may be beneficial to use the same QR solve component for all solvers that need one.

5.2.6 Banded upper triangular factor

This domain is very similar to the tridiagonal matrix domain, being doubly indexed. It is typed with `BandedUpperTriangularMatrix`. The `upperBandwidth` function always returns the constant 2. It is generated by the QR solve, and used by the search vector recurrence.

5.2.7 Lazy vector domain

This domain adheres to `IndexedVector`, and is essentially the same as the matrix of basis vectors except that it wraps a recurrence that maps integers to scalars rather than basis vectors. The domain has no caching policy, but it would be cheap to implement. Two elements of this domain are generated by the QR solve.

5.2.8 Search vector recurrence

The search vector recurrence maps the matrix of basis vectors and the upper banded upper triangular factor to a matrix of search vectors: i.e. $(V,R) \mapsto P$ from $PR = V$. The domain that contains the recurrence is typed with `SearchVectorRecurrence`. Its implementation is similar to the Krylov space domain, and is presented in Figure 5.9 (again, code for the first two stages is omitted). It maintains a small number of vectors in its state, but only produces one object, being the matrix of search vectors. The

recurrence is wrapped to produce the matrix of search vectors in much the same way that the Krylov space recurrence is wrapped to produce the matrix of basis vectors, and contains similar hints to manage storage.

```
state := state + 1

t1 := 1/R(state,state) * ( V(state) - R(state-1, state) * p1
                                    - R(state-2, state) * p2 );

dispose!(p2); p2 := t1;
(p1, p2) := (p2, p1);
```

Figure 5.9: Aldor code for the general step of the search vector recurrence. A single lowercase letter and a number indicate elements from a vector domain (e.g. `p1` etc) and uppercase letters indicate elements of a matrix domain (e.g. `R` is a banded upper triangular matrix and `V` is a matrix of column vectors). The result of indexing into a matrix is a scalar or a vector depending on the type of matrix (e.g. `R(state, state)` and `V(state)`). The types of these objects are inferred by the compiler. The integer state variable is called `state`. For further details see Appendix D.

The recurrence uses both the matrix of basis vectors and the upper banded upper triangular factor from the QR decomposition. It is specialised to an upper triangular factor with a band width of two, which is what the QR decomposition for a tridiagonal matrix produces.

5.2.9 Matrix of search vectors

The domain used for this purpose is identical to the domain used for the basis vectors (see Section 5.2.3).

5.2.10 Glue code

The glue code, presented in Figure 5.10, constructs two new objects and combines them with the short recurrence template to produce the iterative solver algorithm QMR. The first object is itself a function that takes the matrix of Krylov basis vectors V and

```
QMR(A : Operator, x : Vector, b : Vector, t : GroundField)
  : Vector == {

      tolerance := valuation(t);

      minimumResidualCorrection(T : TDom, V : VDom, beta : Valuation)
          : (zDom, VDom, SI -> Boolean) == {

        (R, z, res) := directQR(T, canonicalBasisVector(1, beta));
        P := recurrence(V, R);

        lastIteration?(i : SI) : Boolean == {
          residual : GroundField := res(i);
          if valuation(residual) < tolerance then true else false;
        }

        return(z, P, lastIteration?);
      }

      solveFunction := iterativeSolve(biorthogonalKrylovBasis(b),
                                      minimumResidualCorrection);
      return solveFunction(A, x, b);
    }
}
```

Figure 5.10: Aldor glue code for QMR. Note that the domains in function signatures (i.e. `Operator`, `TDom` etc) are parameters to the domain wrapper for the glue code. A single lowercase letter indicates elements from a vector domain (e.g. `z`, `res` etc) and uppercase letters indicate elements of the operator domain or a matrix domain (e.g. `A` and `R`, `P` etc). For further details see Appendix D.

the knowns of the projected system (T and $\beta_1 u_1$), and produces the halting test, and the (lazy) matrix P and (lazy) vector z that go to make up corrections to the solution. For QMR the halting test is just based on res_n, but for an LU decomposition it may involve $||v_n||$ depending on how the Krylov space recurrence is written; and so must be constructed here where the basis vectors are available. The second object is the function from an operator and the first basis vector to a Krylov space object. This is done by supplying the first argument (the start vector for the dual Krylov basis) to the curried function `biorthogonalKrylovBasis`. The glue code is wrapped in a simple domain that is typed with an anonymous category.

5.3 Evaluation of Framework Design

The original motivation for pursuing the design of the algorithm framework and its supporting domain implementations came from colleagues in Particle Physics Theory at the University of Edinburgh (especially Professor Kennedy and Dr Joó). Their initial work, which dealt mostly with the linear algebra categories, was developed into the modular approach to the iterative solver algorithms presented here. The design introduces modularity and structure using the type system in order to:

- Enable easy assembly and reuse of multiple domain components.
- Encourage clarity and conciseness in the implementation of any given component.
- Provide flexibility without entailing a proliferation of different versions through the ability to customise individual parts.
- Highlight how the algorithms relate to each other by explicitly showing the parts they have in common.

The first point is demonstrated by Figure 5.10. The glue code assembles together many different objects to construct the QMR method. Reuse can be shown by considering the construction of the function value assigned to `solveFunction` near the bottom of figure. The template function for a two sided method, `iterativeSolve`, takes two

arguments. The first is a component that generates a two sided Krylov space and the second is a local function that uses other components to calculate the search vectors and check for termination based on the recurrence residual. A two sided method based on an *LU* decomposition could be constructed in a similar manner using the same template function and Krylov space component, but with a different function to calculate the search vectors. Conversely, a Hermitian *QR* method would use a different component to generate the basis vectors but otherwise would be essentially unchanged.

The clarity and conciseness of algorithm components themselves can be seen in Figure 5.7. Except for the `dispose!` functions and a handful of type annotations, the code is almost a direct copy of the original algorithm from [42].

Flexibility with respect to adaptation can be shown by considering the short recurrence template function in Figure 5.5. The termination condition in the loop that updates the approximation `x` is based directly on the `lastIteration?` function, which is based on the recurrence residual. The template function could easily be modified to take another argument, being the required tolerance on the *actual* residual, with the loop running until the recurrence residual is satisfied (i.e. `lastIteration?` returns true), after which the actual residual is calculated on each iteration. When the actual residual tolerance is satisfied the loop is terminated. This scheme is cheaper than having to calculate the actual residual on each iteration, which requires an extra matrix–vector multiplication, and more accurate than purely relying on the recurrence residual, which can drift substantially from the value of the true residual. Making the change in this part of the structure means that the new approach to termination conditions is automatically propagated to all the short recurrence methods as opposed to having to substantially change multiple individual recipes for each method[1].

The final goal of the design is essentially automatic from introducing modularity into the framework. The reuse of components for different algorithms by definition shows what parts the algorithms have in common.

[1]Note that the changes are not completely isolated however – the glue code for each method would require minor changes to accept the error tolerance on the actual residual and pass it to the template function in an appropriate manner.

5.3.1 Remaining issues

5.3.1.1 Recursive category definition

In a couple of places there exist problems with the type system, or possibly its current implementation. They stem from recursive definitions, with the two exemplars being the definition of an inner product space using the `LinearSpaceWithDual` category, and the first argument to `HermitianOperatorKrylovSpace` (which ought to be a field with valuation of itself so that it can be used as the coefficient field of the reduced system – see Section 5.1.2.1). In the first instance, the compiler accepts the original inner product space category definition without problem, but cannot subsequently deduce that something belonging to it is a linear space whose dual is itself. In the second instance, the compiler appears not to be able to accept any function parameter that is recursively typed. Both these problems are resolved by circumventing the type system using the `pretend` keyword.

5.3.1.2 Mutability and aliasing

Because Aldor is an imperative language that also supports garbage collection, the management of storage is a thorny issue. The subjects outlined here relate to aliasing of arguments to and outputs from the various recurrences, using the Krylov space domain as the example.

The operator and the two initial vectors provided as arguments to construct a Krylov space object may be aliased. If the object stores only a pointer to them, then if they are altered through their aliases the object is no longer correct – if the starting vectors are altered the object will produce a different sequence of vectors if a restart occurs, and if the operator is altered any further vectors in the same sequence will be wrong. Similarly, the recurrence stores pointers to the vectors that represent its current state, and if pointers to this internal state are yielded to a client so that the state may be read, then it may become corrupted if the client alters the vectors through those pointers. Also, if a client holds on to the aliases and the state is destructively updated then from the client's perspective the vectors may become corrupted.

There is no ideal solution to this. Adding explicit copy operations would remove

the possibility of corruption at the expense of ugliness, and possibly severe inefficiency as a result of garbage collection overhead. Consequently, a compromise is implemented. The operator is assumed never to be altered, and the client of the recurrence is assumed to be well behaved in that it never destructively updates a vector through a provided pointer, and never mistakenly re-reads the state after the Krylov object has been updated. Vectors provided as arguments to the recurrence are copied before being cached to provide some degree of security.

Similar reasoning underlies the use of the `dispose!` functions in the code examples given. It is reasonable to expect the compiler to deal with chunks of memory that are allocated within a routine and never escape it rather than leaving all heap variables to the garbage collector, and at the same time unacceptable to expect a programmer to explicitly manage all unnamed temporaries. A simple strategy for unnamed temporaries is the pre-allocation of space (see Section 9.2.2), which can be viewed as an extension to the action of the emerger. However, automatically dealing with explicitly referenced nonlocal heap allocated variables (such as named vectors) would require some kind of interprocedural alias analysis. This is likely to be exceedingly difficult in the presence of higher order functions (such as the closures manipulated by the various recurrences discussed in this chapter) or the leaking of pointers to clients to enable them to read internal state. The fully automatic management of storage is a subject in its own right, spanning garbage collection techniques, user annotations/exotic types systems, and static analyses of various sorts [97, 98, 92, 77] (not to mention combinations of these). However, the problem is very general and the form of the linear solvers does not raise any new issues to be addressed. Because of this, and the fact that Aldor allows the use of destruction hints, these issues were not pursued any further.

The cost of not explicitly destroying these variables is extra work for the garbage collector, and the severity of the penalty depends on the size and number of the objects, and the type of garbage collector. Early experiences with the programs discussed in this thesis suggested that garbage collecting heap allocated scalar variables is cheap enough to be inconsequential, but garbage collecting large objects such as vectors was simply too expensive. The problem may be pinning (the collector is conservative), or

lack of cache/page locality[2], but the problem is easily solved with the use of destruction hints.

5.4 Summary

This chapter has described how the structure of a family of iterative solvers has been captured using the rich type system of Aldor, and given an example of how the framework can be instantiated with implementations of various pieces to give QMR (an iterative solver algorithm). The structure of the solvers is embodied in the interfaces provided by categories and their interrelationships, by means of inheritance and parameterisation. The implementation is constructed from several recurrences that are packaged using the advanced functional features of the language.

While the relationships between the algorithms discussed in Chapter 3 are often mentioned in the literature, it is nonetheless still normal practice to present and code any one of the algorithms as a simple recipe with all the choices already made, and all the separate pieces unpacked and merged together. That chapter contained outlines of four families of Krylov subspace generation (Arnoldi, IOM, Hermitian Lanczos and two-sided Lanczos), three orthogonality conditions (Galerkin, minimum residual and minimum error), and two methods of matrix factorisation (*LU* and *QR*), along with the algorithms that arise by combining them together. Splitting up the algorithms into those sub-components and implementing them as shown in this chapter brings significant software engineering benefits in terms of flexibility, code re-use and comprehensibility.

Numerical programs of this nature are not common in functional languages, so there is no obvious body of work with which to compare the design. An isolated example for iterative solvers (which contains a handful of further references) is [101], but the use of the language is at the standard recipe level rather than attempting to represent the full structure of the algorithms.

[2]Due to relying on a tracing scheme rather than a collection strategy with better locality characteristics such as reference counting.

Chapter 6

Linear Systems

This chapter describes the implementation of the sparse linear systems of equations that are used with the iterative solver framework to conduct program optimisation experiments. As the vectors in these linear systems have little or no special structure, the focus is on the operators. A rough sketch is given of how these sorts of systems can arise from the discretisation of partial differential equations, in order to provide motivation and highlight the important differences from dense matrix problems. The chapter begins with this outline, followed by a high level description of the operators in question, with the remainder of the chapter devoted to an account of their actual implementation in Aldor. More details on the domain implementations can be found in Appendix D.

6.1 Partial Differential Equations and Their Discretisation

The discretisation that we consider is a regular finite difference approximation. The space on which the function is defined is approximated with a regular grid of points separated by a uniform distance a, and the value at each point on this grid defines the function. An approximate first or second-order derivative is usually calculated using some low order Taylor expansion in the grid spacing a, and so computing the required discretized approximation to the derivative of the function at a given point involves the

value at the grid point itself, and any immediate neighbours to which it is linked. If the point is at the edge of the space considered then the derivative will involve a boundary condition of some sort[1], but we will ignore that here.

A function over the grid is represented by a vector in the algebraic formulation, with each element of the vector corresponding to the value of the function at some grid point, and the differential operator is represented by a matrix. Applying the matrix to the vector to give a new vector[2] equates to calculating the required approximate derivative for each point on the original vector, and so the nonzero entries of the matrix correspond to a link with a neighbour on the grid for the purpose of approximating the derivative. The value of the entries in the matrix itself is determined by the differential operator in the PDE (for instance, possibly by scale constants). Thus, discretisation of a PDE gives a system of linear equations that can be solved with an iterative algorithm.

6.1.1 Grid numbering, matrix layout and stencils

Each grid point (also called a *site*) in the space corresponds to the index of one element of a vector, and so the correspondence is a numbering scheme for the sites. The numbering scheme used directly affects the form of the matrix for the differential operator. For instance, the *natural ordering* of grid points for a three-dimensional discretized Laplacian operator ∇^2 gives a multi-diagonal matrix, an example of which is shown in Figure 6.1. The exact structure of the matrix for a regular grid with this ordering varies depending on the boundary conditions, (and for some boundary conditions) the number of grid dimensions and whether each dimension has even or odd extent. Other orderings can be used, for example as part of read-black preconditioning, but they will not be considered here.

Most of the entries in the matrices under consideration are zero, as each site is only connected to a small fixed number of neighbours. Therefore, it is unnecessary to deal with the "full" matrix by storing all the zero elements, and more efficient to

[1](Anti-) periodic boundary conditions actually define a manifold with no edge (e.g. torus, twisted fibre bundle etc) but they can be thought of as a space that has a special procedure for wrapping around at the edges, and this is how they are typically implemented.

[2]NB: we are only considering operators that map a vector to another vector of the same type (e.g. scalar field to scalar field) rather than a different one (e.g. scalar field to vector field by calculating the gradient).

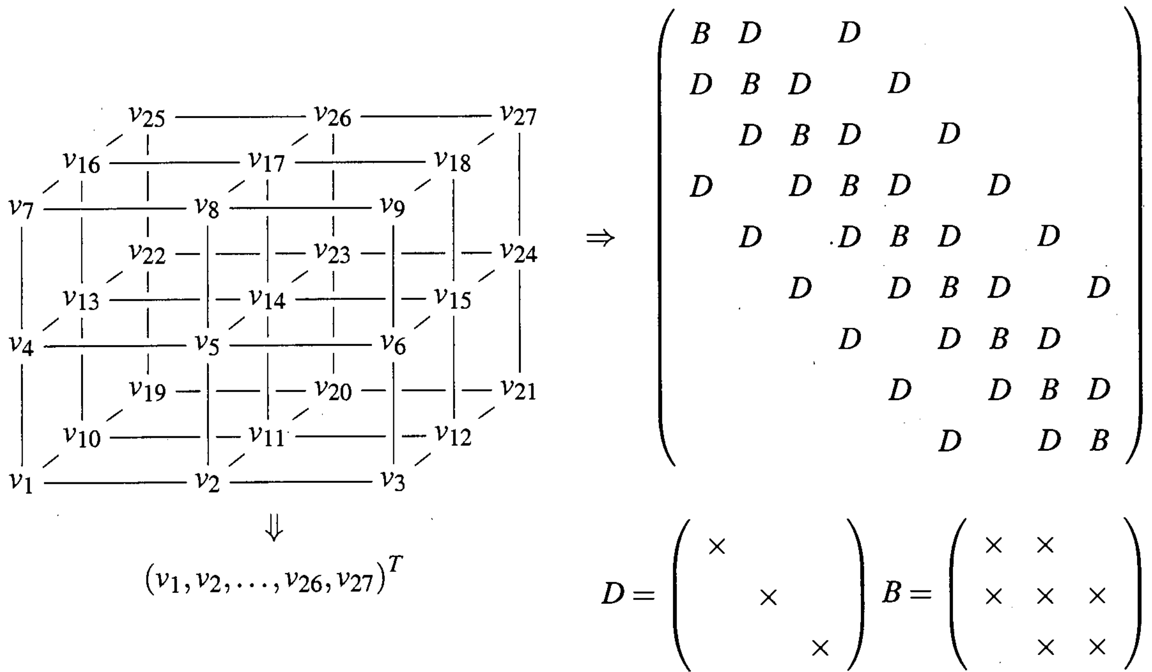


Figure 6.1: A naturally ordered labelling of sites on a 3D grid (top left), the correspondence to entries in a vector representing a function over the space (immediately below), and the 27×27 matrix that results from approximating ∇^2 using this labelling (to the right). The matrix is presented as being composed of 3×3 sub-blocks (D and B) whose structure is given below it. Blank entries denote zeros. Boundary conditions have been left out to aid clarity.

encode how to apply it. This requires knowing the neighbours of each site (which gives us the nonzero entries), and possibly some other information embodied by the operator (which gives us the value of those nonzero entries). Matrices with this fixed regular structure are called *stencils* in this thesis, and the special case where the value of nonzero entries can be factored out into, for example, a single scale constant are called *pure stencils*. An encoding of a stencil can save on space by not representing zero elements of the operator, and save on calculation by omitting operations on zero entries.

6.2 Example Operators

There are three different stencils used with the linear solvers in this thesis. In order of increasing complexity, they are:

1. A three dimensional, naturally ordered, simple Laplacian-like operator with a fixed scale constant, on a complex scalar valued function.
2. A four dimensional, naturally ordered, simple Laplacian-like operator with a fixed scale constant, on a complex scalar valued function.
3. An unpreconditioned Wilson-Dirac operator from QCD (four dimensional).

6.2.1 The Laplacian-like simple operators

The first two operators have more-or-less been described in the preceding section. They differ from a true Laplacian in that a complex scale constant κ is used. When explicitly represented as matrices they would have seven and nine diagonals respectively, but they are represented in the code as pure stencil operations. For example, to apply the simple 3D operator to produce a new vector, each element of the result is calculated using the following recipe:

$$\begin{aligned} u_{i,j,k} \; := \kappa \; & (v_{i+1,j,k} + v_{i-1,j,k} + v_{i,j+1,k} \\ & + v_{i,j-1,k} + v_{i,j,k+1} + v_{i,j,k-1} - 6v_{i,j,k}) \end{aligned} \tag{6.1}$$

where subscripts denote grid indices, which wrap around to give periodic boundary conditions (the 4D version is a simple generalisation of this to four dimensions). The form of a stencil can be easily related to the discretized grid – the new value at each point relies only on the values of neighbouring points. The zero values of the associated matrix are neither stored nor manipulated. Operators of a given type are applied using the same scheme. The only difference between them is the value of the single scale constant κ, which is all that has to be stored. Operations on an operator manipulate the scale constant - for example, taking the adjoint of an operator conjugates it.

Note that there exist specialised logarithmic time solvers for problems of this sort. Their use in this thesis is merely as a simple example of a purely functional operator –

a more difficult problem could be posed by extending the operator with a scale value that depends on some function of the site index.

6.2.2 The Wilson-Dirac operator

The unpreconditioned Wilson-Dirac operator is taken from applications for the numerical modelling of QCD. The following description of it is purely at the "recipe" level – that is, a simple description of how it is calculated in one particular instance, rather than any of the rich theory behind it. This summary is based on various sources including [83, 35, 59, 82].

The operator acts on a vector ψ representing a four-dimensional grid, but the "value" at each site on the grid is a $\mathbb{C}^{12}$ vector (a *colour-spin* vector) rather than, say, a single scalar in the case of the simple stencils. The operator can be written down as a short expression, containing a *delta term* as the most significant component.

6.2.2.1 The delta term

The delta term Δ can be thought of most simply as a complicated cousin of a simple four-dimensional Laplacian. The new value at a given site is a sum of the eight nearest neighbours (one in each direction for each dimension), after they have been acted on by certain matrices. The standard notation is to write the term as a sum over the four grid dimensions:

$$\Delta\psi(x) = \sum_{\mu=1}^{4} D(x,\mu)\psi(x+\mu) + D(x,-\mu)\psi(x-\mu)$$

The expressions in parenthesis denote indexing operations, so $\psi(x)$ denotes the value of vector ψ at site x (where x is a 4-tuple of integers), with $\psi(x+\mu)$ and $\psi(x-\mu)$ being its immediate neighbours in the μ direction. Similarly, $D(x,\mu)$ denotes the matrix specific to that site and that direction. Note that unlike the simple stencils, the previous value at a given site plays no part in the delta term – this is taken into account by a different part of the parent expression.

6.2.2.1.1 Decomposing D Conceptually, the values at each site are complex 12-vectors, and the matrices that act on them are 12×12 complex matrices. However, these $D \in \mathbb{C}^{12\times 12}$ matrices can be decomposed into a Kronecker product of two matrices, $\mathbb{C}^{3\times 3} \otimes \mathbb{C}^{4\times 4}$. To exploit this structure, the complex 12-vector is arranged as a 4-vector each of whose elements is a 3-vector, somewhat akin to a 4×3 complex matrix. Representing the values at the sites in this way allows the action of $D = U \otimes P$ to be computed by applying the two factors $P \in \mathbb{C}^{4\times 4}$ and $U \in \mathbb{C}^{3\times 3}$ one after the other along their appropriate dimensions.

6.2.2.1.2 Projectors The eight $P \in \mathbb{C}^{4\times 4}$ matrices used at each site are each the result of an expression with the following form:

$$I \pm \gamma_n \qquad (n \in 1..4) \tag{6.2}$$

where I is the identity, and γ_n one of four γ-matrices. The γ-matrix in expression 6.2 is determined by the grid dimension of the link (each matrix is associated with one of the four dimensions), and the sign is determined by the direction within that dimension, so the expression is independent of the site index.

All eight matrices that result from these expressions have many zeros and so are themselves sparse. The form of the matrices means that their product with any $\mathbb{C}^4$ vector will only have two linearly independent components, and so can be represented by a $\mathbb{C}^2$ vector and some implicit information (the linear factors). Hence these matrices are called *projectors* in this thesis[3].

This fact is customarily exploited to reduce the overall work in applying the $\mathbb{C}^{3\times 3}$ matrix. First the projector P is applied which reduces the 4×3 matrix representing the value at a site to a 2×3 matrix, after which the 3×3 matrix can be applied for half the cost, and finally the resulting 2×3 matrix is reconstructed using the implicit information into a 4×3 matrix again so that it can be added to the running sum. This gives the following:

$$\Delta\psi(x) = \sum_{\mu=1}^{4} U(x,\mu)P(\mu)\psi(x+\mu) + U(x,-\mu)P(-\mu)\psi(x-\mu)$$

[3]This is a slight abuse of terminology, as to be true projectors the matrices must be idempotent (that is $P^2 = P$) for which they need an extra scalar factor of a half.

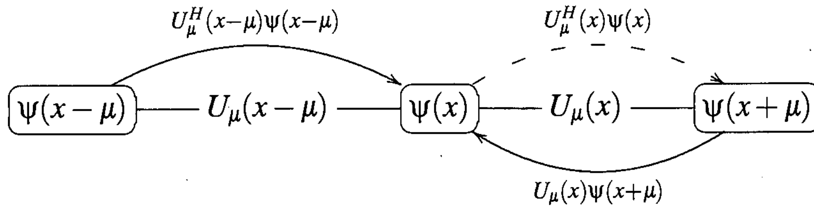


Figure 6.2: One dimension of the grid for the delta term, showing sites (boxed), links (where the link matrices are grouped with the site on their left), the contributions to $\Delta\psi(x)$ (in bold arrows) and part of the contribution to $\Delta\psi(x+\mu)$ (dashed)

6.2.2.1.3 $SU(3)$ **matrices** The eight $U \in \mathbb{C}^{3\times3}$ matrices used at each site (two per direction) are elements of the fundamental representation of the matrix group $SU(3)$, and as such the inverse of each matrix is its adjoint. Each matrix is associated with a link between sites, with one per link. To give them an index, the link matrices are grouped with one of the sites that they link, with four matrices per site. Hence the $U_\mu(x)$, where $\mu \in 1\ldots4$, are grouped with site $\psi(x)$. This is what gives rise to the indices used in equation 6.3 — the $U_\mu(x)$ are conceptually grouped with the current site and the $U_\mu(x-\mu)$ are all grouped with different neighbouring sites. When calculating the contribution to a site $\psi(x)$ from its neighbours, the value $\psi(x\pm\mu)$ will be multiplied by the link matrix or its inverse (which is just its Hermitian transpose), depending on which site the matrix is grouped with. This is illustrated in Figure 6.2 – the contribution of site $x+\mu$ to the new value at site x will be multiplied by $U_\mu(x)$, whereas the contribution of site $x-\mu$ will be multiplied by $U_\mu^{-1}(x-\mu) = U^H(x-\mu)$. The entire collection of link matrices is called a *gauge field.*

Bringing all this information together, the delta term for a given site at index x can be written as follows:

$$\Delta\psi(x) = \sum_{\mu=1}^{4} U_\mu(x)\left(I-\gamma_\mu\right)\psi(x+\mu) + U_\mu^H(x-\mu)\left(I+\gamma_\mu\right)\psi(x-\mu) \qquad (6.3)$$

where the projectors have been written in terms of their expressions and enclosed in parentheses (note that this is not supposed to indicate indexing). The Hermitian transpose of the delta term can be calculated simply by reversing the sign of both expressions involving the γ-matrices, which amounts to changing the projector.

6.2.2.2 The unpreconditioned Wilson-Dirac operator

With the description of the delta term, the unpreconditioned Wilson-Dirac operator can be written as follows:

$$(I - \kappa\Delta) \tag{6.4}$$

where $\kappa \in \mathbb{C}$ is some scalar parameter, and the sites on the grid are arranged in the natural ordering. Applying the stencil at each site $\psi(x)$ involves calculating the new value $\psi_{new}(x) = \psi(x) - \kappa\Delta\psi(x)$.

6.3 Domain Implementation

The interface between the implementation of the iterative solver algorithms and the implementation of the systems of linear equations that they deal with is captured by a handful of categories – the valuation, ground field, vector and operator categories. The Laplacian-like systems require scalar domains, described below, and vector and operator domains, described in the following section. The Wilson-Dirac system uses the same scalar domains as the simple operators, but has an extra layer between them and the vector/operator domains, being the subdomains representing projectors, link matrices and the objects at the sites upon which they act. These are discussed in Sections 6.3.3 and 6.3.5 respectively.

The descriptions of the domains include an outline of how they and operations on them are ultimately represented in FOAM code after the standard Aldor compiler optimisations. This prepares the way for a discussion of optimisation issues in subsequent chapters.

6.3.1 The scalar domains

The scalar domains are simple number types that are close to the abstract machine – that is, they are usually represented in FOAM by a small number of abstract machine words, and the operations on them are either directly FOAM instructions or a small

sequence thereof. The scalar types that are larger than a single word have to be wrapped in records (or "boxed") to satisfy the uniform representation rule (see Section 2.2.3).

It is not possible to destructively update elements of these domains – i.e. they are pure. For the boxed domains, each operation if taken in isolation must allocate a new box to hold its result.

6.3.1.1 `SingleInteger`

This is the domain of signed single word integers, taken directly from `axllib`. The domain representation is directly the built-in single word integer from the definition of the language. Most of the mathematical operations on this type can be implemented with single abstract machine instructions. The domain belongs to the `axllib` category `Ring`, and both the following two domains are typed as a `Module` over this one.

6.3.1.2 `DoubleFloat`

This domain is originally from `axllib`, and has been extended with several categories and their corresponding operations. For instance, it now satisfies `OrderedField`, the specific named category that is used for valuations, and also `FieldWithValuation` where the valuation domain is simply itself. The representation of the domain is a record with a single member, a double precision float type from the definition of the language that is two words large. The majority of the operations on elements of the domain involve abstract machine instructions for handling the boxing and unboxing of the actual values, with a single instruction to perform the mathematical operation.

6.3.1.3 `ComplexDoubleFloat`

The `axllib` library has a parameterised domain for complex numbers, where the parameter is the type of the component elements. This has been used as the basis for an extended non-parameterised version based on the double float domain described above, and typed as a `FieldWithValuation` over it. The domain representation is a record of two further records that each contain a double precision float. It should be noted that there is no complex number type built into the definition of the language,

and consequently no complex number type supported by the abstract machine. As such, in addition to the boxing/unboxing steps, most operations on this type involve some small number of the elementary abstract machine operations on double precision floats.

6.3.2 The simple stencil operator and vector domains

The simple vector (and associated operator) domains are built from the scalar domains and a small number of core Aldor domains, which correspond more-or-less directly with FOAM counterparts. Operations on the members are either fairly simple straight-line programs or loops (after generator inlining, emerging and control flow restructuring, see Section 2.3.1) to act on the elements of an array.

6.3.2.1 `Vector3D`

This domain is typed in the most general way possible as a `NormedInnerproductSpace` so that it can be used with any of the Krylov spaces. Its representation in Aldor is a simple packed array of complex double floats using `PrimitiveArray`. Operations on the members of the domain are defined by iterating operations on individual vector elements in a simple one-dimensional loop. At the source level, the loop is specified as a `for` loop controlled by a generator over a closed integer segment that gives the successive values of a loop variable used to index into the array (see Figure 2.2 in Chapter 2 for a simple example of these loops over a vector of double floats). This translates directly to flat arrays and simple loops in FOAM after the standard optimisations. Each complex element of the vector is represented by two double precision values, so the FOAM array has twice as many elements as the vector it represents.

Operations that produce a vector are pure, in that they do not destructively update their arguments or alias them in any way. The vector to hold the result of the operation is allocated and then written to element-by-element using the destructive update, and so strictly speaking the internals of an operation are not pure. The Hermitian inner product is implemented by conjugating its second argument, and the norm is implemented by calculating the valuation of the scalar produced by taking the inner product of a vector with itself.

6.3.2.2 `SimpleOperator3D`

The function to calculate the application of an operator to a vector (using equation 6.1) is associated with the domain, and the only variation between elements of the type is the value of the scale constant κ. Hence, the domain representation is just a record containing said constant. The domain is typed with `LinearOperatorOnInnerProduct-Space` as its associated vectors are treated as an inner product space, and hence it can be used with any Krylov space except `HermitianOperatorKrylovSpace`.

The choice of how to code the mapping of a flat array to the three-dimensional grid, the calculation of the stencil, and especially what happens at the boundaries of the domain is not trivial. Section 6.4.1 discusses some of the trade-offs. The method employed here is to calculate the relevant flat indices for the neighbours of each point, the number of which is determined by the form of stencil, and store them in an offset table. This is done once at beginning of the program as soon as the grid dimensions are available, with the offset table being used for each application thereafter. The relevant mapping from three-dimensional index to flat index is hard coded within the function that calculates offsets; boundary conditions are periodic. To calculate the application of the operator, a one-dimensional `for` loop over a generator created from a closed integer segment is used to visit each point of the result vector in turn. The value for the point is calculated using the stencil expression 6.1 using elements from the source vector as directed by the offset table.

6.3.2.3 `SimpleOperator4D`

This domain (and its associated vector) is a trivial modification of the three-dimensional version described above, using a four dimensional rather than three-dimensional stencil and associated offset table.

6.3.3 Subdomains for the Wilson-Dirac problem

This section introduces the domains from which the Wilson-Dirac operator and vector are constructed. They are the colour vector, 4-spinor and SU(3) domains, the projector package, and the complex double float domain detailed earlier. The `ColourVector`

and `Spinor4` domains in combination define the $\mathbb{C}^{12}$ colour-spin vector values that constitute the grid sites. The `SU3` domain represents elements of the gauge field (link matrices), and the `Projector` package represents the eight static parts of the delta term given by expression 6.2.

6.3.4 Aggregate structures of subdomains

Packed array operations supplied by the programmer for some element type can deal equally well with objects taken from domains that use arrays or records as their representation, but records are of a known fixed size for the compiler and therefore can be acted on by the environment emerger. Failure to remove the heap allocation otherwise needed when accessing objects from a packed array almost always has a large performance impact.

The colour vector, 4-spinor and SU(3) matrix domains are all homogenous aggregates of a simpler domain (complex scalars for the colour vector and SU(3) matrices, and colour vectors for the 4-spinor), and so the first choice of domain representation might be arrays. However, all of the subdomains, either directly or indirectly, are used at some point as the element type of an array.

Because of this, the domain representations are written as (nested) Aldor records, and in practice these are always replaced with some collection of simple FOAM variables due to the action of the environment emerger. Using records rather than arrays complicates access to individual elements. This may introduce overhead if the index of the element is not known statically by the compiler, such as accessing elements using an induction variable in a loop. By writing operations on the subdomains as straight line programs with fixed static offsets for element access, this problem is avoided through a combination of inlining, constant folding, and dead code elimination. The objects are small enough for this style to be natural, and consequently they can be thought of as "larger" cousins of the scalar domains.

6.3.4.1 `ColourVector`

The domain of colour vectors defines an `InnerProductSpace` of three-element vectors where the elements are members of the complex double float domain. The domain

representation is a record, and the linear space operations are implemented element-wise in the expected way.

6.3.4.2 `Spinor4`

The 4-spinor domain defines a `InnerProductSpace` over the complex double float domain. The domain representation is a record of four elements of the colour vector domain, whose operations are used to implement the linear space operations of the 4-spinor.

The 4-spinor is structured in this way so that the Kronecker product decomposition of the operator can be exploited (see Section 6.2.2.1.1) , and the domain is not intended to represent a linear operator in its own right. The order of the composition of the domains has been chosen due to the order in which parts of the Kronecker product of the operator are applied, which in turn is determined by techniques to reduce the amount of computation (see Section 6.2.2.1.2) .

6.3.4.3 `SU3`

The SU(3) matrix domain is a `GroupAction` on the colour vectors, with the group operation being matrix-matrix multiplication and the action being matrix-vector multiplication. The domain is conceptually made up of $\mathbb{C}^{3\times 3}$ matrices with elements taken from the complex double float domain, and uses a nine element record for its domain representation. The exported operations (e.g. matrix-vector multiplication) are implemented in a simple element wise fashion.

The domain is intended to represent the $SU(3)$ matrix group, and as such any given element of the domain is implicitly unitary and has determinant equal to one, but these properties are not checked for statically or dynamically. As the matrices are unitary, the inverse of any given matrix is simply its adjoint. For greater efficiency, the default inverse multiply export \ is overridden with a function that calculates the result directly without creating the adjoint of the matrix.

```
gamma2pos(U : SU3, v : Spinor4) : Spinor4 == {
   u0 := U * (s(0) + i * s(3));
   u1 := U * (s(1) + i *  s(2));
   return [u0, u1, (-i) * u1, (-i) * u0];
}

gamma2neg(U : SU3, v : Spinor4) : Spinor4 == {
   u0 := U \ (s(0) - i * s(3));
   u1 := U \ (s(1) - i *  s(2));
   return [u0, u1, i * u1, i * u0];
}
```

Figure 6.3: Two of the eight functions from the `Projector` package, representing $U(I-\gamma_2)v$ and $U^H(I+\gamma_2)v$ respectively.

6.3.4.4 `Projector`

The projector domain is a package of eight functions used to capture the part of the delta term involving the gamma matrices and the associated tricks. The functions do not represent their gamma matrices concretely, but encode how to apply them in order to take advantage of their sparse nature. The package is typed with its own special purpose anonymous category.

The functions are defined on the second linear space, that is they map 4-spinor objects to 4-spinor objects. However, the action of a projector function actually represents applying the Kronecker product of the relevant projector and an SU(3) matrix, and this is reflected in the functions taking as arguments an element of the SU(3) domain as well as the 4-spinor being acted on. See Figure 6.3 for some example code.

This arrangement allows us to reduce the amount of work in applying the SU(3) matrix (see Section 6.2.2.1.2) without having to have an awkward explicit type for a "projected" 4-spinor consisting of the two linearly independent components and the linear factors.

6.3.5 The Wilson-Dirac Operator and Vector Domains

6.3.5.1 `SpinorField`

The spinor field domain represents vectors of 4-spinor objects, and is very similar to the simple 4D vector. The domain representation in Aldor is a packed array of 4-spinor objects, which translates to a flat array of double floats in FOAM, where each 4-spinor object corresponds to 24 elements of the array. The linear space operations for the spinor field are again implemented as simple one-dimensional loops, using the linear space operations exported by the 4-spinor domain.

6.3.5.2 `NaturallyOrderedWilsonDiracOperator`

Elements of the Wilson-Dirac operator domain are represented by a record of a scale constant (i.e. κ), similarly to the simple operator domains, and a packed array of SU(3) matrix objects with four matrices per site (i.e. for every spinor field element), which translates to a flat array of double floats in FOAM with 72 elements per site. The domain is typed using `LinearOperatorWithDual`. Consequently, no dynamic tests are needed to know which of the normal or adjoint applications to use for a given matrix-vector multiplication, but the domain can only be used with `BiorthogonalKrylovSpace`.

A member of the Wilson-Dirac operator domain is applied in a similar manner to the simple operators. In addition to fetching elements of the source vector to use with the delta term, the relevant SU(3) matrices must be fetched from the gauge field. The index of the set of gauge matrices for the positive directions is just that of the current site, but the offset table must be used to get the relevant indices of the four matrices in the negative directions.

The delta term is implemented in a straightforward manner using the functions from the projector domain, and the stencil term is very simply constructed using it. The elements of the operator domain all use the same function to calculate their application to a vector, and differ only in the values of the scale constant and the gauge field. A specialised function to directly apply the adjoint of an operator follows the same approach but uses a slightly different delta term.

6.4 Design Issues

6.4.1 Boundary conditions and indexing

Currently, elements for the stencil are fetched based on entries in an offset table holding the linearised addresses of the neighbours of a given site. Ultimately, the code was written in this way to allow a more equal comparison against codes written in other languages using the same mechanism to do the same job (see the assembly and C controls in Chapter 10). An alternative might be to write the stencil as a three/four dimensional loop over a vector, making use of a three/four dimensional index function that incorporates the boundary conditions of the grid. A brief discussion of some of the issues is given below.

The direct advantage of the alternative method is that no extra storage is needed for the offset table, with the extra cache/register pressure that it brings. The disadvantage is that the offsets must be calculated for each element, in each iteration of the loop, for every application. In addition, boundary conditions mean that the indexing functions must contain conditional branches to cope with accesses at the edges of the grid (and thus possible branch penalties), and, in the case of periodic boundary conditions, calculating the offsets for boundary points involves expensive modulo arithmetic. A less obvious cost associated with the use of these complicated multi-dimensional indexing functions is the greatly increased code size after inlining multiple instances of them (and the associated instruction cache/TLB misses) as compared to using the offset table. Experiments with a prototype of the Wilson-Dirac operator using four dimensional loops suggest that it has significantly worse performance than the version using offset tables, even if iteration over the internal points and boundary points is separated to avoid run-time tests and code blowup for the loop over the internal points (in fact, the separated version performed worse than the non-separated one).

Although the use of an offset table is probably more efficient in most cases, using multi-dimensional loops and index functions is possibly more elegant in terms of presentation. One issue related to this is that an offset table obscures what may otherwise be statically determinable data dependencies and data reference patterns by making the required information part of a dynamic data structure. This may prevent a compiler

from reordering the iterations of the loop (e.g. tiling the application of the stencil, discussed in Appendix B) or being able to fuse it with some other operations.

6.5 Summary

This chapter has given a description of three sparse linear systems (characterised by their operators – 3D, 4D simple operators and unpreconditioned Wilson Dirac operator) and how they have been implemented in Aldor to be used with the solver framework described in Chapter 5. Together they provide a means of encoding a linear system (of certain restricted types) to be solved and constructing a numerical algorithm from the framework that can be used to solve it. This chapter also discussed how the sparsity structure of the systems can be captured, and how this is reflected in the implementation. Overall, the emphasis in the description has been on describing the sections of code that are important to a discussion of the optimising transformations developed in Chapters 7 and 8.

The simple 3D and 4D systems are built from relatively simple domains of scalars, a vector domain and an operator domain. The vector domain is represented by an array of scalars, and its associated operations are calculated using loops over the arrays. The operator domain captures how to apply a stencil to a vector, and has a very simple representation as there is little difference between individual operators (a scale constant). The unpreconditioned Wilson-Dirac system is similar, but has an extra layer of complexity. A series of subdomains is constructed from the scalar domains (represented implicitly or explicitly as small matrices), and the vector and operator domains are built from and manipulate these subdomain objects.

Chapter 7

Optimisation across Components

This chapter describes the building blocks of the optimising compiler transformations developed in Chapter 8, and motivates the overall approach with reference to the language (Chapter 2) and the modular components structure of the application (Chapters 5 and 6). The chapter begins by giving some basic formalisms that are fundamentally necessary to further discussions. The objects under consideration are statements, basic blocks, loops, arrays and dependencies between statements.

After laying these foundations, the transformations called *loop fusion* and *array contraction* are introduced, followed by a description of their impact on the performance of programs on cache based computer architectures with reference to temporal locality. These code transformations are the basic constituents of collective loop fusion and array contraction. After introducing them, a motivating discussion is given that details why these cross-component optimisations rather than other transformations are applied in the context of the global loop structure under consideration. The main point is that each loop taken in isolation from the original program has little or no exploitable reuse (intraloop locality), and so loops must be considered collectively (for interloop locality) to improve cache performance. This discussion is separate to considerations of the specificity of optimisations to the combination of language and application mentioned in the introduction.

7.1 Basic Terminology and Formalisms

7.1.1 Loops

In this part of the thesis, a *loop* refers to an iterative program construct where a finite number of different *iterations* are described by distinct values of a single *induction* or *loop variable* (i.e. a restricted `for` loop). A loop variable has a lower bound, an upper bound and a stride, with the lower bound and the stride both equal to one unless otherwise stated, and strides always positive (to simplify the discussion). Induction variables are only ever updated by the loop construct to which they belong, and the set of values that a variable takes is referred to as its *range*. The section of code executed on each iteration is called the loop *body*. A loop of this type may be explicitly represented in the constructs of the language in question (e.g. Fortran), or implicitly built up from smaller operations such as conditional tests, branching, and arithmetic on the loop induction variable (e.g. FOAM). For some example pseudocode loops, see Figure 7.1.

Loops can be nested inside one another. For a loop nested inside another loop (an inner loop), an iteration can be described by the value of the induction variable for the loop itself and the value of the induction variables for any enclosing (outer) loops if the extra context is necessary. Each tuple of values that the describing induction variables can take for a given iteration is called a *loop index*, and the set of all loop index tuples for a given loop is known as its *iteration space*.

A *perfect loop nest* is one where only the innermost loop has a body that contains anything other than another loop. The innermost loop body (or simply the body) may only contain forward branches to targets within itself, meaning that the iteration space of the loop nest fully describes how the loop is executed as there can be no early exit by jumping out of the loop body and no implicit loops within it. Perfect loop nests of depth n are referred to as n dimensional loops, with an n dimensional iteration space. The n dimensions of the iteration space are ordered by the nesting of their associated loops, with the first dimension corresponding to the outermost loop, so a given iteration space implicitly gives a complete ordering on the execution of its iterations. If the lower bound, upper bound and stride are constant for all n induction variables in an n

dimensional loop, the iteration space is said to be rectangular.

7.1.2 Dependencies between loop iterations

Although operations in a program are normally presented with a strict ordering, there is usually only an implicit partial order constraint between them, called the *program dependencies*. Alterations to the program that do not violate these dependencies maintain the program's original semantics (provided we are willing to ignore problems with the expected order of exceptions, and possible problems arising from the reordering of semi-associative operations such as floatingpoint arithmetic). Program dependencies come in two types, *control dependencies* and *data dependencies*. Here we are concerned with the latter type.

Loop bodies usually contain some number of operations to read and write data (see Section 7.1.3). These operations can induce data dependencies and thus an ordering constraint between separate iterations of an individual loop (*loop-carried* dependencies) or between two iterations taken from different loops. A loop with no loop-carried dependencies is termed *fully parallel*, as it would be legal to execute all its iterations concurrently. A loop-carried dependence can be described by a *distance vector* formed by subtracting the index tuple of the source iteration from the dependent (target) iteration. Given the assumption of all strides being positive, a distance vector must be lexicographically non-negative (i.e. all its entries must be ≥ 0) to be legal[1]. The set of distance vectors for a given iteration space is usually summarised (e.g. as a *dependency vector* [65]), based on the idea that a loop should only be altered if all its distance vectors meet some criterion.

7.1.3 Statements and their dependencies

A loop body or basic block consists of a sequence of *statements* in some language. Dependencies between loop iterations arise due to the dependencies of statements in the loop body associated with each iteration. In the case of loop-carried dependencies the

[1] A lexicographically negative distance vector implies that the source of the dependence is executed after the target, which is clearly nonsense

same loop body is associated with both the source and target iterations of the dependencies, and, for iterations taken from separate loops, the dependencies link different loop bodies.

For the purposes of this discussion we assume that a statement may read an arbitrary number of source operands, but we restrict the ability to write values to a special type called *write statements*, with each write statement writing one and only one named destination (i.e. the generation of intermediate results from expressions does not count as a write). A destination (or location) being written to is either a scalar variable, or an indexed entry in an array of scalar variables. A source operand is similar but may also be a constant. The value used to index into an array is the result of an *index expression.* All variables, including arrays, are assumed to be non-overlapping.

Arrays can be accessed using multi-dimensional index functions. A scalar variable always refers to the same location, but the array element referred to by a statement in a loop body can rely on the values of the induction variables for that iteration. Because a single statement may do something different on each loop iteration, it makes sense to talk about the n-th statement of the m-th loop iteration (of loop i) even though syntactically the loop body is the same for each iteration.

Data dependencies can be further classified as *true*, *anti-*, or *output* dependencies. True dependencies flow from a write to a read from the same location, antidependencies flow from a read to a write to the same location, and output dependencies flow from a write to a subsequent write to the same location[2]. Dependencies between statements taken from different iterations of a loop can depend on the order in which the loop iterations will be executed as determined by the original source program.

Statement dependencies for statements in loops have associated distance vectors, defined analogously to distance vectors between loop iterations. For this purpose, basic blocks can be treated as loops with a single iteration. The union of statement dependencies comprises the dependencies for that iteration (of the loop body) as a whole. Dependencies from a statement in the loop body to a subsequent statement in the same body in the same loop iteration are said to have *zero dependence distance* (i.e. their

[2] A notional *input dependency* can be thought of as existing between two reads to the same location (with no intervening writes), but the name is misleading as it does not really constitute a dependency – i.e. it does not constrain the ordering of statements in the program, and therefore is undirected

distance vector equals the zero vector) and do not induce a dependency between loop iterations. To summarise, statement dependencies give rise to loop iteration (and basic block) dependencies. The caption of Figure 7.1 gives a summary of the distance vectors within and between loop iterations in the example, and points out the statements that give rise to them.

7.1.4 Dependence testing

The discovery of dependencies between statements is not covered in this thesis (see Section 9.1.3).

7.1.5 Temporal Locality

In the following discussion of performance, it is assumed that the programs are to be run on a cache based architecture, which provides lower latency and higher bandwidth for references to the same address with "good enough" *temporal locality* (provided a structural conflict has not occurred). Temporal locality is taken to be the number of distinct addresses referenced in between a pair of references to the same address, where good enough locality (i.e. a small enough number of addresses) for a given level of the cache hierarchy means that the second reference will be a hit there.

7.2 Loop Fusion and Array Contraction

7.2.1 Loop Fusion

Perfect loop nests with the same dimension and iteration space are said to be *conformable*. As long as their respective bodies obey certain legality constraints, two conformable loops can be *fused* into a single loop whose body consists of the two bodies of the original loops executed consecutively. See Figure 7.1.

The standard concept of distance vectors can be extended to describe dependencies between iterations from separate but conformable loops. The distance vector is that which would result from fusing the two loops and treating the dependence as if it were calculated from the new aggregate loop. Note that a lexicographically negative

for i **in** 1..10 **do**
    a[i] := ...
done
 	
for i **in** 1..10 **do**
    b[i] := alpha * a[i];
    r := r + b[i];
done
	for i **in** 1..10 **do**
	    a[i] := ...
	    b[i] := alpha * a[i];
	    r := r + b[i];
	done
		for i **in** 1..10 **do**
		    a := ...
		    b[i] := alpha * a;
		    r := r + b[i];
		done

a)	b)	c)

Figure 7.1: A pseudocode example of loop fusion and array contraction. a) Pseudocode for the original pair of loops – array *a* has no other use than in the second loop, but *b* is referred to after this section of code. The loops are conformable, and the distance vector for the use of *a* on each iteration in the second loop is 0 (and hence they can be legally fused). The second loop also has a constant loop-carried dependence of distance 1 for each iteration by way of example, but this does not affect the legality of fusion. b) The result of applying loop fusion. c) The result of subsequently applying array contraction – *a* is now a single scalar rather than an array. Array *b* cannot be contracted as it is still live.

distance vector is no longer nonsensical as all the iterations of the loop with the source dependency will be executed before any iterations of the loop containing the target. The existence of any negative distance vectors between the iterations of two loops indicates that the loops cannot be legally directly fused as the resulting aggregate will have illegal dependencies.

7.2.2 Array contraction

If the only references to a given array occur in a single loop body, each access is to the same element, the first access in the body to the array is a write, and the dependencies associated with all the statements that access the array have zero dependence distance, then the array itself can be replaced by a single element. This code transformation

is known as (complete[3]) *array contraction* (see Figure 7.1). It can be applied irrespective of loop and array dimension provided the necessary dependence information is available, and generalises easily to loop bodies that access multiple array elements provided all accesses obey the conditions given above (for consecutive elements this can be thought of as accessing one single object in an array of objects, where each object consists of multiple array elements). Loop fusion is an enabling transformation for array contraction, as fusing loops together may increase the opportunities to apply it. Indeed, the original motivation for studying collective loop fusion was to enable array contraction [37].

7.2.3 Effects of loop fusion and array contraction

Loop fusion and array contraction can be used to improve the memory subsystem performance of a program. Array contraction reduces the number of addresses touched within a loop (assuming nonoverlapping arrays). This improves the temporal locality of references either side of the loop, and can reduce the cost of saves to the temporary by replacing a sequence of isolated references with a sequence of references to the same address resulting in less memory traffic, lower latency for write hits etc. In certain cases it may be possible to keep the contracted scalar in the register file, thus completely eliminating read/write latency and the need to issue load/store instructions (and any associated bandwidth limitations). Array contraction on its own is highly unlikely to degrade the performance of a program.

Loop fusion can affect locality in a number of ways. A fusion step that does not enable array contraction but merges two loops that are connected by an input dependence will improve the locality of the pairs of reads to the common array. Any fusion of two loops that enables contraction must improve the locality of the creation and subsequent use of each element of the temporary array by bringing them into the same loop iteration. Loop fusion can also degrade the temporal locality of some pairs of references by changing the order and/or proximity of the remaining unfused loops.

Loop fusion also has secondary performance effects. Reducing the number of loops

[3]Note that the subject of partial array contraction for loop carried dependencies of a known fixed distance is not considered in this thesis.

in a program reduces the overhead for execution. Combining loop bodies together may have a negative impact, such as instruction cache misses if the loop body gets too large, and increased structural conflicts in the data cache – e.g. fusing together loops that manipulate distinct arrays will increase the amount of live data in the cache for a given loop iteration, which increases the likelihood of data being mapped to the same cache line and exceeding the associativity limits. Similarly, loop fusion can increase register pressure and require the introduction of the spill code. In theory, a large enough increase in live data could lead to capacity misses in the first level of cache, but this is unlikely and is certainly not the case for the programs and architectures considered in this thesis. Finally, the more complex data access patterns of fused loops may interfere with hardware data prefetching mechanisms where they exist, leading to a trade-off of higher latency for the loads in a fused loop versus the savings from more cache hits.

7.3 Temporal Locality, Aldor and Iterative Solvers

Given the general goal of improving memory subsystem performance by targeting temporal locality through high-level transformations, the following section summarises why loop fusion and array contraction were chosen for investigation in the context of the iterative solver programs developed in this thesis. Any discussion of temporal locality assumes some concrete program (rather than just an abstract algorithm) and a machine on which it is run. Hence, this section brings together the algorithms, the implementations of the domains that they manipulate, the way they are compiled together, the definition of the abstract machine and a mapping from code on the abstract machine to an executable on a real architecture.

The first task is to characterise the temporal locality of the original programs as specified and compiled, and the second task is to consider how to improve it.

7.3.1 Temporal locality of original programs

The source level programs derived from the algorithmic framework consist of high level algorithms composed of operations on the elements of a handful of lower domains (operators, vectors and scalars) expressed as separate functions. The operations

exported by these lower domains are implemented using simple operations and the higher-order `generator` construct to iterate over an integer segment and the associated array elements, which are ultimately optimised to simple scalar manipulations or loops over arrays in FOAM, as discussed in Chapter 6. Assuming "large" vectors, the vast majority of memory references occur during operations involving them, so parts of the program that do not involve loops over entire vectors (represented as arrays) in some way are ignored as being inconsequential to the temporal locality characteristics and performance of a program.

The compiler compiles source into FOAM code with far fewer function calls by means of aggressive inlining and emerging. Inlining and emerging on their own do not alter the overall loop structure of the program, so even after compilation to FOAM the loops in the program that manipulate vectors can be directly associated with a high level operation, although there is no longer a 1-1 correspondence as inlining creates multiple copies of the original functions. The native C compiler used for the experiments in Chapter 10 does nothing further by way of inlining or loop restructuring, and it is reasonable to suggest that this would be the case for most native compilers as the functions that it gets given are already large after aggressive inlining by the Aldor compiler, and it is unlikely to have the necessary alias information for restructuring given the nature of the generated C code. In brief, the compiler chain as it stands does not alter the temporal locality of data references as defined by the original source program.

7.3.2 Finding opportunities to improve temporal locality

To find opportunities to improve the performance of a program that manipulates large arrays, it is usual to start by examining each individual loop nests in isolation (an approach stated as being the norm in [61]; see [12] for a survey of various loop transformations), and considering how to reorder its iterations to improve locality.

7.3.2.1 Intraloop locality

The vector operations equate to three types of loop over arrays:

- *Simple loop* – this reads two arrays and writes a single array. There are no

dependencies between separate loop iterations.

- *Reduction loop* – this reads two arrays and writes a single scalar. Each iteration depends on the previous one (i.e. there is a constant true dependence of distance one).
- *Operator application loop* – this reads the offset table (an array), a source array, and optionally some representation of the operator (e.g. a gauge field array), and writes a single array. There are no dependencies between separate loop iterations, but potential dependence information is carried in the offset array, which is created at run-time.

The only operation with any intraloop reuse that can be targeted is the operator application (see Appendix B), where the stencil access pattern means that separate iterations may access the same part of the source vector. The maximum extra reuse available is only two references per site, comprised of two references per element of the source vector, and additionally for the Wilson problem some sort of access to elements of the operator representation. For the simple operator problem, exploiting this reuse is equivalent to saving approximately two loads of a complete vector, and for the Wilson problem a saving of approximately three (one quarter of a load of a gauge field is saved, which is slightly less than one complete load of a vector).

7.3.2.2 Interloop locality within a recurrence

To find larger amounts of reuse, it is necessary to consider temporal locality between loops. For the sake of argument, consider the situation where an individual vector is larger than the cache. Fusing together a producer and a consumer of an array gives one factor of reuse and subsequently applying array contraction gives a further factor of re-use for the writes the to the contracted temporary (see Section 7.2.3). Consequently, for both the simple operator and Wilson problems, fusing and contracting only two producer/consumer pairs gives more reuse than tiling the operator application. This leads naturally to a consideration of how best to fuse and contract the collection of loops (all of which are of the three types given above) within the update function for a given recurrence, which is where the majority of loops occur after inlining – for an

example of how a collection of loops arises, see the Krylov space recurrence in Section 5.2.2. In the experimental problem (see Chapter 10), it is possible to fuse many more than two producer/consumer pairs within recurrences using the approach to collective loop fusion outlined in Chapter 8.

7.3.2.3 Interloop locality between recurrences

Similarly to the existence of temporal locality of data across loops within a recurrence, there is also some degree of locality across loops from different recurrences, given that an iteration of a recurrence will produce data that is consumed by another. Again, this leads to a consideration of how this locality can be exploited. Unlike locality within a recurrence however, there is much less to be gained, and the separation of different recurrences using higher order language features that are not immediately removed by the compiler poses a significant barrier to analysis and transformation.

While this problem could theoretically be attacked with some kind of higher order control flow analysis, the problems of developing such a complex analysis framework and the significantly lower amount of locality to be mined means that the priority must be dealing with interloop locality within recurrences.

7.3.3 The impact of modularity

Ultimately, the lack of intraloop locality and the need for interloop optimisations is a result of the modular style of the programs, which is strongly encouraged by the language itself. Consequently, the general idea of *cross-component optimisation*, of which loop fusion is one exemplar, will be important for Aldor and languages like it.

This modularity also affects the dependence structure of the programs. Firstly, there is a simple dependence structure between pairs of loops. A dependence vector from a simple loop or operator application to any other loop (except operator applications) has distance zero; a dependence from a reduction to any other loop has distance n (where n is the dimension of the loops); and a dependence from a simple loop to an operator application has some fixed set of positive and negative distance vectors determined by the access pattern of the stencil for non-boundary iterations (periodic boundary conditions give some other fixed set of distance vectors depending on which

boundaries the point is located). Secondly, there is a direct correspondence between statement dependencies and aggregated loop iteration dependencies. This means that dealing with dependencies at the loop iteration level and using the simple test for the legality of loop fusion does not introduce unnecessary conservativeness.

7.3.4 Applicability of proposed method

It is much less usual to find programs with such simple loop structure when they have been written in standard third-generation languages, such as C or Fortran, as they encourage programmers to construct their own arbitrarily complex loops by hand on a per expression basis[4]. This may limit the direct applicability of collective loop fusion as presented in Chapter 8 by artificially imposing the dependence restrictions of one subsection of the loop body on the whole, and by restricting the choices of the compiler by forcibly combining some statements in loops. It also reduces the expected benefit as some degree of fusion is already incorporated into programs. It is interesting to note however, that although standard languages do not support modularity in the same way as Aldor, one important study shows that general interloop locality is nonetheless important in the context of a set of well-known imperative benchmarks and ought to be targeted by compiler optimisations [61]. This suggests that there is no intrinsic cost in encouraging modularity, as not having it (and by implication expecting the programmer to arrange loops by hand) is at best a partial solution to the optimisation problem, under the reasonable assumption that techniques to handle the interloop problem in traditional programming languages should easily extend to properly modular codes such as Aldor programs.

Most work on collective loop fusion has been done with these languages in mind, leading some authors to suggest that full loop distribution and/or scalar expansion ought to be used as a pre-processing pass before collective loop fusion to get around these problems [51]. The collective loop fusion/contraction problem derived from programs written in Aldor and other more traditional languages may consequently end up looking very similar, and so the work on loop fusion in this thesis is not restricted to this type of language.

[4]With the partial exception of languages with array statement constructs such as Fortran 90.

7.4 Summary

This chapter has introduced the basic transformations called loop fusion and array contraction, explained how they can be applied as cross-component optimisations to programs derived from the iterative solver framework from Chapter 5, and outlined why optimisation across components is crucial for these types of programs when considering temporal locality for cache based architectures. Fusing together two loops with a cross-loop dependence of distance 0 brings into the same iteration references to the same address that would otherwise be separated by many loop iterations (and associated accesses to different addresses). This makes the address far more likely to be cache resident for the second use. Subsequent array contraction changes a series of references to different addresses in different loop iterations into references to the same address thus reducing the number of addresses touched. This eliminates the possible compulsory misses for all but the first iteration, and improves temporal locality of addresses touched either side of the fused loop.

These optimisations are important for programs derived from the solver framework (and after standard optimisations) as there is virtually no locality of reference within any individual loop, but large amounts across different loops that can be exploited by fusion/contraction – thus we need to consider collective loop fusion (and array contraction). Although such simple loops are less common in traditional imperative languages, interloop locality is still considered to be crucial, and techniques such as loop distribution and scalar expansion applied to programs in those languages may result in a similar optimisation problem to the one considered here.

Chapter 8

Iterative Collective Loop Fusion

This chapter introduces the standard formalisms for a compiler approach to tackling the collective loop fusion (and array contraction) problem. This is followed by the theory and algorithms behind our novel approach to collective loop fusion, which, in Chapter 10, is applied to the programs derived from the algorithmic framework developed earlier in the thesis. The description builds on the formalisms and basic transformations introduced in Chapter 7.

8.1 Loop Dependence Graph

A *loop dependence graph* (LDG) describes a *program section* that consists of basic blocks and perfectly nested loops with no branching allowed (ignoring the branching implicit in the loop constructs themselves and that permitted in their bodies), for which a set of data dependence relationships is available that constitutes a safe (but possibly conservative) approximation of those possible in the actual program. Nodes in the graph represent the loops of the program section, and a directed edge exists between two nodes if the target is data dependent on the source in some way. The lack of branching in the program section ensures that its LDG is acyclic. Basic blocks are not explicitly represented in the LDG, but the dependencies connecting them to each other and to loops must be known, and the dependencies between loops that they induce must be present. Hence an edge exists in the graph if the target loop is directly

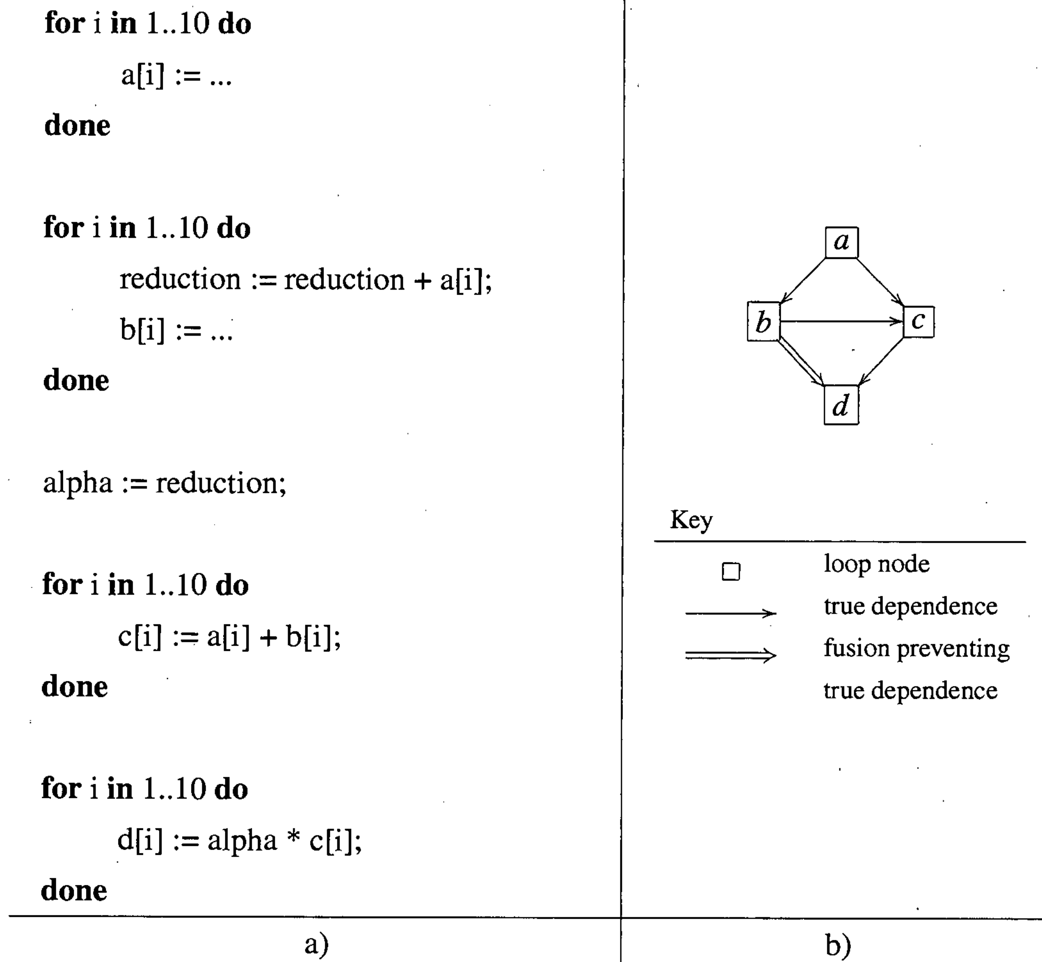


Figure 8.1: An example LDG. a) Pseudocode for the original program section, with four loops and one basic block. Only array *d* is live out of the program section (i.e. read at some later point), so all the other arrays can potentially be contracted. The loops are all conformable, and all distance vectors are 0, except for the loop-carried dependence in the second loop for a reduction variable, the dependence of the basic block on said reduction variable, and the dependence of the fourth loop on the basic block. b) The corresponding loop dependence graph. Nodes in the graph are labelled with the name of the array that they write to.

data dependent on the source loop, or if there exists some chain of data dependencies through basic blocks.

The LDG is used in this thesis to reason about loop fusion for the program section that it represents. The nodes representing two conformable loops are possible candidates to be fused (subject to further constraints detailed in the next section) if there is no dependency between them, or if they are directly dependent and all the distance vectors from the source to the target are non-negative. In the latter case an edge in the LDG representing such a dependency is labelled as *collapsible*, and concomitantly an edge representing a collection containing negative distance vectors, a dependency between non-conformable loops, or a dependency carried by a chain of basic blocks is *non-collapsible*. A *dependency path* (or just path) in the LDG is a set of edges describing a path from a source node to a destination node through the graph following the directed edges. A path is collapsible if all its edges are collapsible, and non-collapsible otherwise.

An example of a program section and associated LDG is given in Figure 8.1.

8.2 Collective Loop Fusion and Fusion Partitions

Loop fusion can be considered as a transformation on the LDG. Data dependencies are transitive, and so two nodes may be legally fused if there is no path between them in the LDG, or if there exist only collapsible paths of length one. When two loops are fused together, their corresponding nodes are removed from the graph and replaced with a node representing the aggregated loop. Any edges that were incident at either of the original two nodes are now incident at the new node, except those edges that linked the two original nodes which are removed from the graph entirely.

If there is a legal opportunity, loop fusion can be applied again to the new LDG, and the process can be continued arbitrarily until at some point we run out of opportunities to apply the transformation. Repeatedly applying loop fusion in this way is called *collective loop fusion*, and can be treated as finding a legal *fusion partition* for the LDG. A fusion partition is a partitioning of the nodes of the LDG into disjoint sets (*partitions* or *clusters*) where the nodes in each set will be fused together to produce

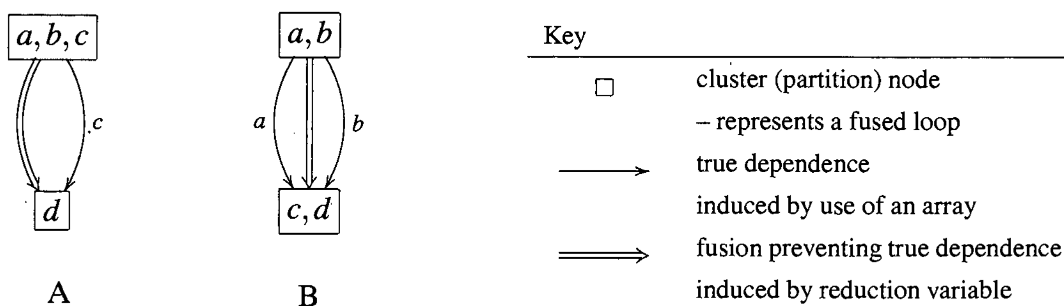


Figure 8.2: The graphs of two possible fusion partitions of the LDG from Figure 8.1. Nodes in the graph (clusters) are labelled with the letters representing the loop nodes within that cluster. Both fusion partitions are the same size (2), but permit different amounts of array contraction – partition A allows two arrays to be contracted (*a* and *b*), whereas B allows only one (*c*). This corresponds (inversely) to the inter-cluster array dependency edges in the graphs of the fusion partitions, which are labelled with the non-contracted array they correspond to – one for partition A and two for B.

the final transformed code. Note that the partitions themselves are not distinguished, so permuting any cluster labels (if they exist) for a fusion partition does not give a new fusion partition.

The *size* of a fusion partition is the number of non-empty partitions it has (empty partitions are not allowed). A fusion partition itself can be represented by a graph where nodes are clusters, and there is an edge between cluster nodes for every edge that exists between the loop nodes that belong to the respective partitions in the LDG. For a fusion partition to be legal, it must be possible to fuse together all the nodes within a given partition, and the graph of the fusion partition must be acyclic. The first condition is satisfied by the absence of non-collapsible edges within the cluster, as the method of fusion does not re-order the iterations of the loops involved and so fusion is associative. In the context of fusion partitions, non-collapsible edges are also known as *fusion preventing* edges. An example of two fusion partitions of the same size derived from the LDG in Figure 8.1 is given in Figure 8.2.

A given LDG has a lower bound on the size of its legal fusion partitions determined by the dependency path with the most fusion preventing edges in it, and trivially an upper bound determined by the number of nodes (loops are never split). The number

of legal fusion partitions of a given size for an LDG can be very large, usually reaching a maximum somewhere in the middle of the size range and becoming smaller at either end.

8.2.1 Array contraction

Finding a fusion partition equates to applying loop fusion to a program section. Subsequent (complete) contraction will be legal for an array in the transformed code if all the dependencies associated with it appear in the same partition, and they all have distance zero. This is equivalent to there being no edges corresponding to a dependence on that array existing between clusters in the graph of the fusion partition.

Applying array contraction to the two fusion partitions of the same size given in Figure 8.2 gives different contraction amounts. Conversely, different size partitions with the same amount of contraction are also possible. The simplest example is two separate nodes unconnected by any dependencies at all – fusing them together gives a partition of size one rather than two, but does not enable any contraction.

A fusion partition on an LDG can be labelled with a pair of numbers that denote the size of the fusion partition and the amount of array contraction that it permits. For a large enough LDG there will be multiple fusion partitions with the same (*contraction amount*, *partition size*) label, and these can be grouped together to give (*contraction amount*, *partition size*) sets (see Chapter 10).

8.3 The Motivation for Search

The previous sections introduced the necessary concepts for the legality of collective loop fusion and array contraction. A given LDG is likely to have many legal fusion partitions though[1], so there needs to be some method for choosing one from the space of possible options based on the intention of improving some characteristic of the program. The two characteristics most studied are total space requirements [32] and

[1] The exact number depends on the form of the loop dependence graph, and cannot be given in a formula. A loose upper bound (which may include many illegal configurations) is given in Section 8.4.1.

program performance. Our focus here is on the latter.

Section 7.2.3 introduced the effects of loop fusion and array contraction on performance. The choice of fusion partition on an LDG usually involves a trade-off in locality for different pairs of references, and so the best choice depends on how the locality characteristics of the program interact with the architecture on which it is being run. These include considerations such as cache size, miss penalty and bandwidth limits, for multiple levels of cache. Hence, choosing a good fusion partition with respect to temporal locality is architecture dependent and far from trivial. The following sections present previous work on choosing a fusion partition to try and maximise benefit, the technique of iterative optimisation, and the approach taken in this thesis.

8.3.1 Standard model based approach

Loop fusion has been used in many contexts to improve temporal locality or to enable other loop transformations (see [60] for one example). In this instance however we are interested in approaches that operate on some graph model, which represents a collection of loops in the program and the expected benefit of fusing any given subset. This collective loop fusion can be contrasted with a purely ad hoc case-by-case method of fusion with no consideration of global effects and choices. Associated with the model is a *cost function* that ranks the possible transformations that can be applied to it.

The simplest example of this is preferring more fusion over less (e.g. [51] in the context of typed loop fusion), with all fusion partitions of the same size being equal. A more sophisticated (and more common) approach is to add to the LDG a set of edges and associated weights that model the expected benefit of fusing the loops that they connect, with the aim of finding a fusion partition with the minimum total cost, calculated as the sum of the weights on the non-collapsed edges between partitions. There have been numerous minor variations on the second approach, depending on the intended purpose of loop fusion. Some examples include transformations specifically for array contraction [37, 56], and a technique which minimises memory usage and simultaneously improves locality whilst limiting the size of any fused loop that is produced (i.e. avoiding "over fusing") [80]. The limit exists to avoid introducing register

spills or associativity conflicts. One adaptation replaced edges in the cost graph with hyper edges to better capture re-use between array operands being read [29]. There have also been several composite approaches, such as a technique that prevents the creation of loops with parallelisation-preventing loop-carried dependencies [51], and a related approach that uses adjustable weights which can be altered to favour fusion for parallelism or fusion for locality [79].

The abstract formulation of various problems has been shown to be at least NP-hard [29, 26, 27]. Consequently, most work on loop fusion is based on heuristic algorithms to find some approximation to the optimum model answer. Approaches have included various greedy algorithms [56, 52], and algorithms based on max-flow min-cut heuristics [37, 29, 80].

As well as finding the best fusion partition, some authors have considered how to deal with less well-behaved loops using techniques such as preprocessing with peeling/shifting etc [80, 27]. This can be related to other approaches using much more general formalisms, such as affine transformations – for example see [90] for locality optimisations and [57] which combines array contraction and tiling. However, these last two works are more ad hoc in that they are not collective – they do not attempt to find the optimal fusion partition with respect to some model, and in addition neither of them attempts to apply search.

8.3.2 Iterative optimisation

Current implementations of computer architectures contain a wide variety of complex structures [47], with variations including cache architecture (with differences in miss policy, replacement policy, capacity, line size, and set-associativity at multiple levels of cache etc), register renaming, and speculation (including differences in re-order buffer size, load/store buffer size, speculative loads and hardware instruction/data prefetching mechanisms etc). Consequently, they are very difficult to model accurately, and any idealisation of the hardware is likely to miss a large number of subtle interactions that can affect how a program executes – for one example of this see [68]. To combat this problem, the approach of iterative optimisation [16] treats the goal of finding good transformations as a search problem, with the space of possible transformations as the

space to be searched through, and the cost function as the empirical cost of executing the program that results from a candidate transformation. Typically the target of optimisation is single processor performance, using either high-level loop transformations [53, 36] or standard scalar optimisations [6], but combinations of performance and code size [88], energy consumption for DSPs [39] and parallel performance [66] have also received attention. Here we are interested in single processor performance.

Because it employs empirical testing, iterative optimisation can only be completely accurately applied in situations where the choice of transformations is the only variable that affects program execution across different runs (although some degree of dynamic difference between search runs and the final use can be accommodated [36]). This rules out many programs that have strongly dynamic behaviour, but tends to be suited to numerical scientific and technical codes that manipulate large data structures (usually arrays) using simple, repetitive and entirely deterministic control flow. In addition, the process of iterative optimisation is very compute intensive, requiring each candidate to be compiled and executed. Consequently, conducting any significant amount of search will require large amounts of time and/or resources, and is only suitable when the cost can be recouped. Happily, this again fits with numerical scientific and technical codes that are typically very compute intensive and run for long periods of time. Another area where iterative optimisation has been profitably used is compiling for DSP applications, where again control flow is largely static [13] and the cost of search is amortised over many products.

8.3.3 Previous approaches and this work

There are two major weaknesses in previous model based fusion/contraction work. The first is the use in some approaches of overly simple search strategies to find some approximation to the solution of the idealised NP-hard problem (e.g. greedy search). As pointed out in [62], the majority of LDGs encountered in realistic programs will be small, and hence there is no real reason to emphasise the efficiency of the search so much at the cost of the quality of the approximation. Indeed, the authors generated a number of artificially large LDG problems and showed that, for their particular cost function, the problems could be exactly solved using a commercial integer linear pro-

gramming package in a small number of seconds. The second problem is that although all the approaches discussed in Section 8.3.1 target slightly different optimisations, it can be assumed that their ultimate goal is to get the best performance for a given LDG, but no authors have adequately explored the possible differences between their idealised problem and the implementation details of actual hardware, choosing instead to largely ignore the trade-offs outlined in Section 7.2.3.

One illustration of this is that, for a given LDG, there may be many fusion partitions all ranked equal according to some abstract cost function (e.g. all with the same amount of contraction). However, for any method in the literature there is not usually any indication of how any particular one is chosen, or any indication of how the actual quality of the equally ranked LDGs varies in practice. Another illustration is the lack of any indication as to how fusion for locality and fusion for contraction may conflict, how the trade-off should be managed to get the best performance, and crucially how this may vary depending on the form of the loops and the actual machine under consideration.

The optimisation strategy adopted in this thesis is to apply the approach of iterative optimisation to finding good fusion partitions for array contraction on a single processor machine, with respect to performance. The responsibility for ensuring that a code is control flow deterministic, and the ultimate control of how much time to spend on optimisation, is left to the user. Using search simultaneously tackles both the problems with previous work, in that no unnecessary shortcuts are taken in searching for a solution, and some attempt is made to take into account the full baroque complexity of modern machines.

Almost all previous approaches to iterative optimisation deal exclusively with search spaces that are the Cartesian product of some number of options (e.g. array padding and tiling and unrolling factors for a loop [53]), with a notable exception being [66] which searches through a space including legal and illegal transformations. This work similarly deals with search spaces that are themselves nontrivial to generate (see Section 8.4.1). Also, loop fusion is rarely included in iterative optimisation work, with [66, 39] being two largely isolated examples. In the first of these papers loop fusion is implicitly included in the action of generated space-time mappings, but appears to be

applied in an ad hoc fashion with no mention of choosing fusion partitions etc (in fact, fusion is almost not mentioned at all) – the primary focus of the paper is on finding parallelisation transformations with good performance. In the second, a small experiment on four loops with no fusion preventing dependencies finds that fusing all loops together gives the best reduction in energy use, but the main emphasis is on tiling and unrolling. Again, there is no mention of fusion partitions. In both papers there is no mention at all of array contraction.

One specific piece of work that is interesting with respect to this thesis due to connections at different levels is [69], which considers a superset of the optimisations treated here, applies them to a similar problem domain (linear solvers and stencil codes), and uses empirical search to guide optimisation. Although it includes partial array contraction, the technique seems to be largely aimed at improving locality using fusion and a subsequent specialised version of tiling directed at the important loops derived from two algorithms, red-black Gauss-Seidel relaxation and multigrid (applied to stencils similar to those in Chapter 6). It does not fit into the previous overview as it does not construct and manipulate a graph based model of the expected benefit of fusion and contraction, and only applies them to a collection of loops in an ad hoc fashion. Contraction appears to be rarely applied in practice. Search over the space of parameters is only outlined. From the sparse details, it would appear that the search space is of the standard Cartesian kind, including yes/no options for contraction, some control over the degree of interleaving (fusion) of loops, and the space of tile shapes and sizes – there is no mention of trying different fusion partitions. Search is conducted using simulated annealing, with no obvious motivation for why it was chosen over a simpler technique.

8.4 Iterative Collective Loop Fusion

The rest of this chapter describes the novel contribution of this thesis to the field of compiler optimisations. To perform iterative loop fusion exhaustively we simply require a method of enumerating all the legal fusion partitions for a given LDG, and the means to empirically test their run-times. The size of the search space, that is the num-

ber of legal fusion partitions, almost always makes testing each point in it unfeasible, so there must be some method of selecting a subset of the search space to test. For example, under the assumption that each empirical test takes about five to ten minutes to enumerate, compile, run and collect results from, and that a week is the longest that we are willing to spend searching, we are limited to testing roughly one to two thousand separate fusion partitions when, even for small problems, the total number of legal fusion partitions is likely to run into many thousands.

Because generating computer representations of legal fusion partitions is much cheaper than testing them, the general idea is to generate the set of legal fusion partitions, cut the set down to an acceptable size using some criteria, and then empirically test for the best one. However, although generating legal fusion partitions is cheap, it is not free, and the total space of them is too large to even generate in practice. This stems from the general case of the problem having at least NP complexity, which is the motivation for using heuristics to approximate answers (to the idealised problem) in previous work. Consequently it is necessary to come up with some method of generating a manageable amount of the search space.

8.4.1 Generating legal fusion partitions

Although clusters within a fusion partition are not distinguished, it is useful to label them with identification (ID) numbers to reason about the enumeration of the fusion partitions for an LDG. Clusters are numbered from 1 to n giving a total ordering on the loops produced from a fusion partition.

The naive approach to generating fusion partitions of size n is to assign each node to a partition i with $1 \leq i \leq n$. The vast majority of these configurations will be illegal though, so they will have to be filtered for legality and, more importantly, a large number will have to be generated and tested to find each legal point which makes this a bad option if many legal points are to be generated. The total space (of legal and illegal fusion partitions) has size:

$$\sum_{p=1}^{n} \sum_{i=0}^{n} \sum_{j=0}^{i} \frac{(-1)^j}{i!} \binom{i}{j} (i-j)^p$$

for an LDG with n nodes, where the sum over p denotes the sum over all sizes of fusion partition. This space grows too rapidly with n to be able to generate and test all points, even if the legality test is very cheap.

An alternative is to find some algorithmic way of enumerating only legal fusion partitions. The approach in this thesis is based on node numbering, which is described below, followed by the enumeration algorithm.

8.4.1.1 Node numbering and range finding

Given a loop dependence graph, a target size of fusion partition, and a set of nodes with pre-assigned partition numbers, the forward node numbering procedure provides a test to determine the lower bound on the ID number of the partition to which any given (unassigned) node may belong.

Two directly connected nodes joined by at least one fusion preventing edge must belong to different partitions. Consequently, given any path from a source to a sink, the nodes along the path can be numbered to show the earliest partition that they may belong to (as determined by this path) by grouping the nodes into sets separated by fusion preventing edges and numbering the sets (and their elements) along the path consecutively, counting upward from one. If a set contains a pre-assigned node with a value different from the parent set, then the set is split into two with the second set starting with the pre-assigned node and labelled with its value. Numbering along the path continues as before counting upward from the new value. It is assumed that pre-assigned nodes always have a legal value - i.e. a value greater than or equal to the original parent set to which they belong.

If this procedure is repeated for all paths through the graph with each node being assigned the maximum value over all paths, then the final label P_{min} will denote the earliest possible partition that the node may belong to in this LDG with these pre-assigned nodes.

A pseudocode for the algorithm is provided in Figure 8.3. The description makes use of several simple utility functions:

- NODES(): returns the set of vertices from an aggregate data structure (either an LDG or a set of $(node, partitionID)$ pairs).

NUMBERNODESFORWARDS(*preassigned*, *LDG*)

Description: Labels each unassigned node in the LDG with the earliest partition that it may belong to.

Input: $\begin{cases} LDG\text{, a loop dependence graph} \\ preassigned\text{, a set of } (node,\ partitionID) \text{ pairs} \end{cases}$

Output: An integer label for each node as a set of (*node*, *partitionID*) pairs

(1) $sources := \{(v,\ partitionID = 1) \mid \forall v \in \text{SOURCES}(LDG) \setminus \text{NODES}(preassigned)\ \}$
(2) $assigned := preassigned \cup sources$
(3) $unassigned := \{v \mid v \in \text{NODES(LDG)} \setminus \text{NODES}(assigned)\ \}$
(4) **repeat**
(5) choose $v \in unassigned$ s.t. $\text{PARENTS}(v) \cap unassigned = \emptyset$
(6) $rank_v := 0$
(7) **foreach** $p \in \text{PARENTS}(v)$
(8) **if** $\forall e \in \text{JOINS}(v,\ p)$, FUSIONPREVENTING?(e) = **false**
(9) $rank_{v,p} := \text{RANK}(p, assigned)$
(10) **else**
(11) $rank_{v,p} := 1 + \text{RANK}(p, assigned)$
(12) $rank_v := \text{MAXIMUM}(\ \{\ rank_{v,p}\ \}\ )$
(13) $assigned := assigned \cup \{(v, rank_v)\}$
(14) $unassigned := unassigned \setminus \{v\}$
(15) **until** $unassigned = \emptyset$
(16) **return** $assigned \setminus preassigned$

Figure 8.3: Forward node numbering algorithm

- SOURCES(): returns the set of root (source) vertices in the LDG.
- PARENTS(): returns the parents of a vertex (in the current LDG).
- JOINS(): returns the set of edges that joins two vertices (in the current LDG).
- FUSIONPREVENTING?(): returns a Boolean depending on whether the edge is collapsible not.
- RANK(): returns the partition ID (integer) of a vertex from a set of $(node, partitionID)$ pairs.
- MAXIMUM(): returns the maximum from a set of integers.

The algorithm does not actually enumerate all the paths through the LDG. Instead it successively selects nodes from the unassigned set only after all their parents have been processed. The method of choosing the node is left unspecified, but at the very least a naive approach could be implemented that would check each node for readiness at most $O(n^2)$ times in the worst-case scenario[2].

Given a maximum number of partitions, the same numbering can be repeated in reverse working from sinks to sources, with each sink labelled with the maximum partition number from which one is subtracted each time a fusion preventing edge is passed. This gives NUMBERNODESBACKWARDS(), the result of which denotes the latest possible partition that a node may belong to, P_{max}. Taken together, the two procedures provide the range of partition IDs to which any unassigned node v may belong $P_{v,min} \leq ID_v \leq P_{v,max}$, and also the size of the range for that node $P_{v,max} - P_{v,min} + 1$. Any node with a range of sizes less than or equal to zero indicates that no legal fusion partitions of this size exist for this LDG. This information is provided by the RANGES() function, which essentially just calls NUMBERNODESFORWARDS() and NUMBERNODESBACKWARDS().

An example of the results produced by applying the RANGES() function to a small example problem is given in Figure 8.4. The labelling of the graph thus produced

[2]A similar algorithm to NUMBERNODESFORWARDS() , without the notion of accommodating pre-assigned nodes, can be found in an early paper on the subject [37]. However, the authors do not apply the same technique in reverse, as described here, and do not attempt to enumerate different fusion partitions.

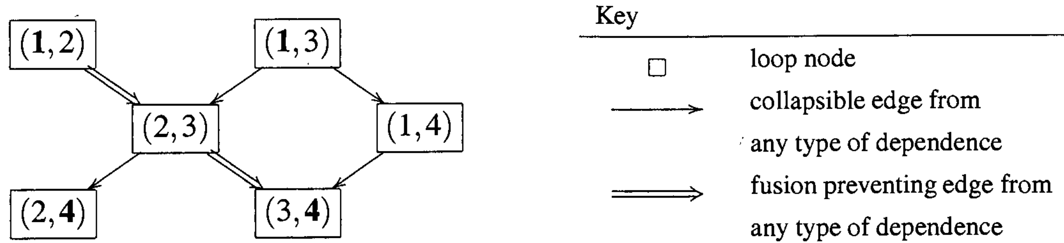


Figure 8.4: An example showing the results produced by RANGES() when calculating possible partitionings into four clusters for a graph containing both collapsible and fusion preventing edges. Each node is labelled with a (*minimum partition number, maximum partition number*) tuple, with numbers in bold indicating that the value results from the node being either a source or a sink in the graph.

shows for each node the earliest (minimum number) and latest (maximum number) cluster that it may belong to for the case of four partitions. Note that this is not the minimum number of partitions possible.

8.4.1.2 Enumeration algorithm

The enumeration algorithm successively generates the fusion partitions of a given size for an LDG. It starts by finding the ranges of the nodes in the LDG, then choosing a $(node, range)$ pair. For the chosen node, the algorithm chooses a value in its range, treats the $(node, value)$ pair as a pre-assigned node, and recursively calls itself. At each step, a check is made to ensure that either all partitions already have ≥ 1 nodes, or that each empty partition is still part of the range of an unassigned node. This prevents generation of fusion partitions with empty clusters, and prunes the enumeration search when it can be shown that any assignment from the current ranges must leave at least one partition empty.

For subsequent calls, a different value from the range of the last assigned node is chosen, until the range has been covered indicating that this recursive step is complete. Note that the ranges of unassigned nodes may change before each recursive function call, and that any unassigned node can be selected for enumeration within a call.

The enumeration algorithm is given in Figure 8.5. As well as the recursive call, it uses two other functions; RANGES(), explained in the previous section, and FU-

ENUMERATEFUSIONPARTITIONS(*LDG*, *size*, *fixed*)

Description: Enumerates the fusion partitions of an LDG

Input:
- *LDG*, a loop dependence graph
- *size*, the required size of fusion partition
- *fixed*, a set of (*node*, *partitionID*) pairs

Output: the set of fusion partitions of size *size* in *LDG*

(1) **if** NODES(*LDG*) $\setminus$ NODES(*fixed*) $= \emptyset$ **then return** FUSIONPARTITION(*fixed*)
(2) $fps := \emptyset$
(3) $ranges :=$ RANGES(*LDG*, *size*, *fixed*)
(4) **if** $\forall p \in \{1, \ldots, size\}\ \exists(v, p) \in fixed \bigvee \exists(v, r_{min}, r_{max}) \in ranges$ such that $(r_{min} \leq p \leq r_{max})$
(5) choose $(v, rank_{v,min}, rank_{v,max})$ from *ranges*
(6) **for** $i := rank_{v,min}$ **to** $rank_{v,max}$
(7) $newFixed := fixed \cup \{(v, i)\}$
(8) $fps := fps \cup$ ENUMERATEFUSIONPARTITIONS(*LDG*, *size*, *newFixed*)
(9) **return** fps

Figure 8.5: Fusion partition enumeration algorithm

SIONPARTITION(), which makes a fusion partition data structure from a list of (*node*, *partitionID*) pairs. In the current implementation there is no special criterion for choosing nodes to fix (they are taken in whatever order they are provided in by the function that calculates the ranges) or values from their ranges (currently they are taken sequentially, from bottom to top by the loop on line 6).

An example of how enumeration works is given in Figure 8.6, based on some initial steps of the algorithm applied to the graph from Figure 8.4. The nodes that are fixed and the values they are fixed at have been chosen to keep the demonstration manageable rather having the two examples as strictly consecutive steps of the algorithm.

8.4.1.3 Outline of correctness for enumeration algorithm

A numbering of a graph gives a set of ranges for its nodes. The Cartesian product of these ranges is the set of *configurations* for that numbering, and the set of configurations from the initial numbering of the graph the *initial configurations*. Note that not all configurations from the initial set are necessarily legal – when the ranges of a parent and a child overlap it is possible to generate a configuration in which the parent is assigned a later partition number than the child, which clearly violates data dependencies.

For all paths in the graph including a given node, the set of nodes that may appear before the given node on any of these paths is called its set of *ancestors*. Similarly, the set of nodes that may appear after it on any of these paths is called its set of *descendants*.

8.4.1.3.1 No double counting The enumeration procedure never generates the same configuration from the initial set twice. At a given recursive step (i.e. an invocation of ENUMERATEFUSIONPARTITIONS()) and for each value of the node under consideration (i.e. in the range $[rank_{v,min}, rank_{v,max}]$), assume that the corresponding recursive call to ENUMERATEFUSIONPARTITIONS() does not double count (i.e. a given recursive invocation never assigns the same values to the remaining unassigned nodes twice thus resulting in duplicate full configurations). If this is true, the only way that the same partial configuration can be generated twice for the union of the current node

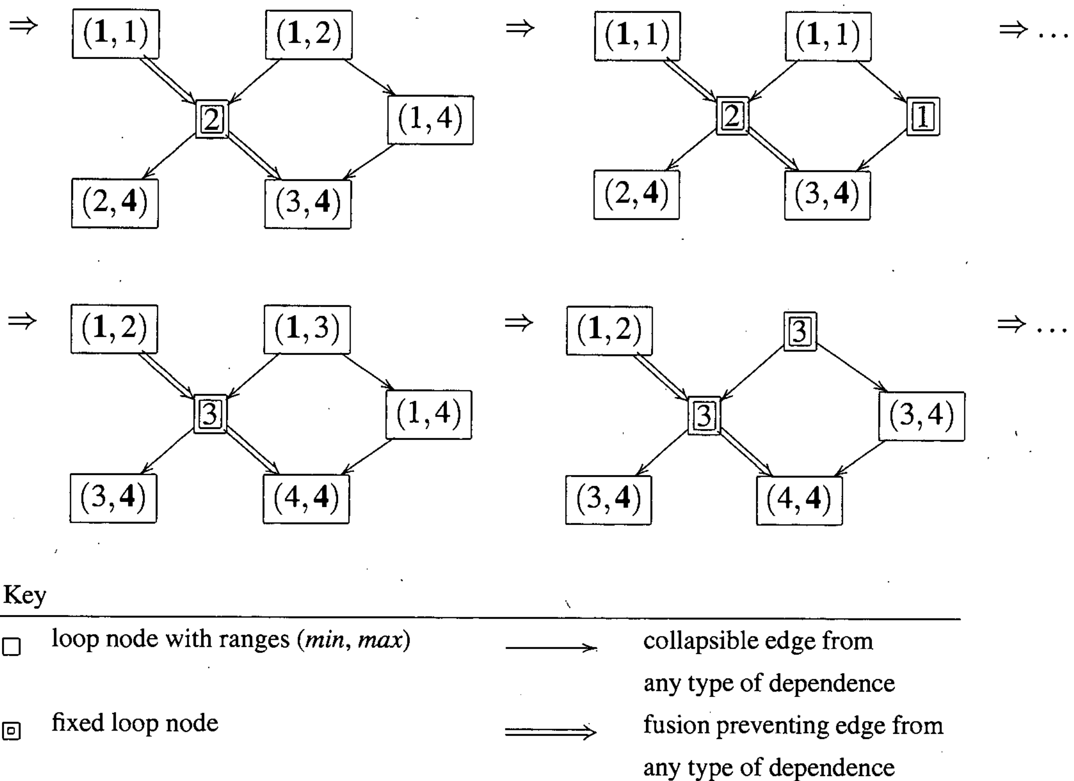


Figure 8.6: Two examples showing how ENUMERATEFUSIONPARTITIONS() proceeds. The initial graph in both cases (not shown, implicitly to the left of the figure) is the numbering taken from Figure 8.4. All graphs show the new numbering after the chosen node has been fixed, and in both final graphs (i.e. on the right) any further choice from the ranges of the unassigned nodes will produce a legal fusion partition (i.e. further calls to RANGES() will not adjust the ranges on the graph). After all of these combinations have been enumerated by the procedure in the top example, the relevant recursive steps will complete and a new value of the second fixed node will be chosen from its previous range (e.g. 2) followed by renumbering. After enumerating all of the values in the range of the second fixed node (and the subsequent recursive steps for other nodes etc), that step will complete and the fixed value of the first node will change to 3, as in the first graph of the second example. Choosing a different node for the next recursive step and continuing the process through several of its values gives the second example.

and the remaining unassigned nodes is if this recursive step has already completed and another recursive step is being executed on this node. However, if this is the case, at least one value of the previously fixed nodes that are not members of the partial configuration in question must have already changed, meaning that all full configurations generated that include these repeated partial configurations must be different. This is trivially true for the final node to be picked from a graph, and thus is true for all the other nodes inductively. This also shows why the order that the values from a range are used within any given recursive step is irrelevant so long as all the values are visited once and only once.

The only further problem is the possibility of multiple full configurations mapping onto the same final program, but this can only occur when at least one fusion partition is empty, and this possibility is explicitly avoided.

8.4.1.3.2 No illegal solutions Once a graph has been numbered, there must be at least one legal configuration for every value in the range of a given node. This can be achieved by assigning all ancestors of the node in question the minimum partition number in their respective ranges and all descendants the maximum – any value from the range of the node in question now gives a legal graph (provided we give the remaining nodes that are neither ancestors nor descendants some sensible values). Hence, it is legal to pick any one node and fix it at any of the values in its range provided by the initial numbering. By the same argument, it must be possible to renumber the graph, and thus fixing a subsequent node at a value from the new ranges must permit at least one legal configuration. This applies inductively to all further choices of nodes to fix and subsequent renumberings, and consequently all full configurations generated in this way must be legal.

8.4.1.3.3 No missed solutions The initial numbering of a graph is general, in that no legal solution can exist outside the ranges given. Similarly, any renumbering of a graph that includes fixed nodes must contain all legal solutions that can be generated from the ranges of the unassigned nodes. Consequently, selecting a given node from the initial graph, assigning it each of the values in its range, and renumbering for each value gives a collection of graphs that contain between them all legal solutions. The

same argument can be given inductively for each of the individual graphs with one node fixed at a given value, thus showing that all the legal solutions will be generated.

8.4.2 Search heuristics and search space reduction

Although generating legal fusion partitions is much cheaper than compiling and running their associated code, the total number of fusion partitions means that generating and storing all of them (i.e. the search space) before applying heuristics to select the empirical tests would take far too much time and space. Consequently, there must be some way of selecting a region of the search space to generate. The choice of this region is governed by the characteristics of the points we hope to find, and therefore stems from the search heuristics themselves.

There are two heuristics to select candidates to empirically test:

1. More array contraction is likely to be better.

2. A smaller size fusion partition (i.e. less clusters) is likely to better.

Both heuristics stem from the goal of improving memory performance, as discussed in Section 7.2.3.

The heuristics are not independent. Given some initial LDG in which all possible array contraction has been done, it is necessary to fuse some loops (i.e. choose a fusion partition) to uncover any more opportunities. Repeated application with a greedy approach does not necessarily apply though, in that the smallest possible fusion partition size may not contain the fusion partition with the most contracted arrays. Nevertheless, the assumption is made that, for a non-pathological LDG derived from an average program, more fusion and more contraction are likely to be related. This last assumption allows us to use the second heuristic to guide the generation of points in the space with the assumption that those points generated will include (the majority of) the good points as determined by the first heuristic.

The algorithm to generate legal fusion partitions allows us to specify the size of fusion partition, and quickly identifies sizes for which no fusion partition exists. This enables a user to (partially) prioritise the space based on fusion partition size and generate points starting with the smallest fusion partitions, moving upward in size if desired.

Although this gives no specified order in which to generate the elements of a particular size, in practice the number of legal fusion partitions gets much smaller toward each end of the size range, and this makes it far more likely that all the legal partitions of some small subrange at the bottom end of the size range can be generated (see Section 10.2 for the generation of all fusion partitions of size 3–6 for a given LDG).

As well as an order in which to generate the search space, there needs to be some halting criterion. This could either be expressed as an amount of space to search, such as a total number of points or a fusion partition size subrange, or a total amount of time to spend generating the search space. Currently it is expressed as a partition size subrange, the extents of which are provided by the user. In a similar vein, the subsequent use of heuristics to select points to test from those that have been generated must be formalised by giving some prioritisation when the number of generated points is larger than the number that can be tested. Currently, the limit on the number of points to test is given by the user. Although it is not used in the experiments in Chapter 10, one method to use in such a case is suggested here for completeness. The points are ranked first by the number of arrays contracted (more is better) followed by the size of the fusion partition (smaller is better). Tie-breaks between equally ranked points are decided in favour of first-come first-served.

8.4.3 Algorithm for generating test cases

Using the enumerating procedure, the overall algorithm for generating cases to test empirically is given in Figure 8.7.

The algorithm starts at small fusion partition sizes, and with each successive iteration the size of fusion partitions that are considered increases by one. Note that the amount of search space to generate and the number of points to try are arguments, as discussed previously. The function SELECTBEST() orders the set *total* based on the search heuristics (e.g. contraction, then partition size, then first come-first served) and then cuts it down to the first *maxCandidates* elements.

GENERATETESTCASES(*LDG*, *maxCandidates*, *minPartitions*, *maxPartitions*)
Description: Enumerates the fusion partitions of an LDG
Input:
- *LDG*, a loop dependence graph
- *maxCandidates*, the maximum number of fusion partitions to generate
- *minPartitions*, the minimum size of fusion partition to generate
- *maxPartitions*, the maximum size of fusion partition to generate

Output: fusion partitions of *LDG*

(1) *candidates* := ∅
(2) **for** *i* := *minPartitions* **to** *maxPartitions*
(3) *fps* := ENUMERATEFUSIONPARTITIONS(*LDG*, *i*, ∅)
(4) *total* := *fps* ∪ *candidates*
(5) *candidates* := SELECTBEST(*maxCandidates*, *total*)
(6) **return** *candidates*

Figure 8.7: Generation of test cases

8.4.4 Code generation

The only requirements on the code generated from the fusion partition of an LDG is that dependencies between partitions are respected in the final ordering of the loops generated from them, and similarly that the dependencies within a partition are respected in the ordering of the bodies from the original loops to form the body of the partition. The first requirement is automatically satisfied by ordering the loops according to the partition label sequence, and the second can be satisfied by a simple topological sort. Currently all legal orderings within a given partition are considered equivalent, under the assumption that the instructions within the body of a partition loop will be rescheduled by some later compilation stage and possibly at run-time by dynamic execution on an out-of-order processor, with any differences in performance being slight.

The placement of basic blocks must also be handled during code generation. The non-contractible dependence edges induced by the presence of basic blocks means that

it will always be possible to place them in at least one gap in between fusion partition loops. The simplest approach is to schedule them as early as possible when a choice exists. Data dependencies between basic blocks themselves must also be respected, and again this can be achieved by a topological sort of the basic block dependence graph.

8.5 Summary

This chapter has introduced a novel means of applying collective loop fusion and array contraction using iterative optimisation, which we call iterative collective loop fusion. The standard approaches to the problem attempt to transform the LDG based on simple metrics such as the resulting number of loops and the amount of array contraction possible, and implicitly rank all the possible different transformations that are equal under the metric as being equivalent. As such, the techniques produce one fusion partition candidate, and typically little or no consideration is made of the target architecture in the choice. Iterative collective loop fusion, on the other hand, employs a search over different fusion partitions and evaluates candidates using empirical measurement on an actual machine. This reflects the fact that the amount of contraction (or fusion partition size) is only a loose approximation to the fitness of an individual fusion partition candidate, and that the optimum choice is likely to be different for different architectures.

The search problem is similar to standard iterative optimisation problems in that it is a very large space, but is dissimilar in that the space of transformations is not a simple Cartesian product of options and is itself nontrivial to generate. As a result, the generation of points in the transformation space is guided by the heuristic of starting with small fusion partitions, and the choice of points to empirically test from those generated is guided by the heuristic of prioritising those fusion partitions with the most array contraction and the smallest size.

The following chapters will describe the prototype implementation of this technique (Chapter 9) and present results collected using it (Chapter 10).

Chapter 9

Prototype and LDG Construction

In this thesis it is assumed that the LDG has already been constructed, both in the theoretical treatment and the prototype implementation used for experiments. This chapter contains a brief outline of how to recover the necessary information for Aldor, with reference to standard techniques and the specific idiosyncrasies of the language in question. After this there follows a description of the prototype used for the experiments.

9.1 LDG Recovery

The information required to construct an LDG is as follows:

- Control flow, to identify program sections from which an LDG can be constructed and the loops in that section.
- Loop index variable ranges and strides, to be able to test for conformability.
- All statement dependencies, to construct edges in the LDG.

These are discussed briefly in the following sections, followed by the outline of an approach to cope with the problems introduced by `envEnsure` instructions.

9.1.1 Control flow

The dependence structure of the program is determined by which instructions may be executed, and therefore control flow must be known (or approximated) to recover it. In addition, it must be shown that control flow in a program section conforms to the requirements given in Chapter 8.

9.1.1.1 Function calls

Function calls within the program section of an LDG are not automatically illegal. However, without interprocedural analysis, we have to assume that a function call may depend on any variables visible to it, and similarly that it may write to any of those variables. This will not include unaliased objects pointed to by purely local (i.e. stack allocated) references unless they are passed as arguments, but will include anything that can potentially be aliased by any member of an environment, such as a lexically scoped reference. The possibility of reading/writing those visible variables may induce dependencies from/to loops in the parent function and alter variables such as those used for loop control (and may therefore affect conformability). Similarly, a second function call may be dependent on the first etc. Although some of this information may be recoverable with an interprocedural analysis, the problem is potentially severely exacerbated by the functional aspects of the language, where the use of closures could necessitate a higher order control flow analysis to know which function is being called. Hence, due to the associated difficulties with recovering potential data dependencies and side-effects, allowing any kind of function call within the program section of an LDG is unlikely to be feasible. In addition, the generation of code for a fusion partition requires the building of custom loops, so there is little point in leaving in place functions that contain loops.

For these reasons it makes sense to position collective loop fusion after the generation of FOAM and the action of the inliner. This also has the benefit of recovering control flow by converting into simple loops the potentially complex `generator` constructs that would otherwise require a higher order analysis. Relying on the inliner to turn interprocedural problems into local ones (a general strategy adopted by the current compiler) is simple, but very sensitive to how inlining is done. As such, issues with

the current inlining strategy ought to be highlighted – see Section 9.3.2.

9.1.1.2 Loop recovery

One side-effect of this positioning of the transformations is that control flow gets lowered to the level of labels and branches due to the way in which FOAM is generated. As a result, information about loops will have to be recovered using some kind of structural analysis, along with some induction variable recognition (for both issues see [65]). Induction variable recognition will be simple as the binding of induction variables to values in `for` constructs is very restricted.

9.1.1.3 `envEnsure` instructions

Unfortunately, requiring program sections to be free of function calls is not enough of a restriction on control flow to eliminate possible problems with dependencies and side-effects. Any call to a function that reads from/writes to its lexical environment must be preceded by an `envEnsure` instruction (see Section 2.2.5), which still remains after the function has been inlined. The effect of an `envEnsure` instruction is at least as bad as a function call, and is probably even less amenable to analysis due to the possible triggering of multiple lazy objects at run-time. Requiring a program section to be free of `envEnsure` instructions is almost certainly too restrictive however, as they are frequently littered throughout generated FOAM code. In particular, given a domain whose functions contain loops that refer to an outer scope for the range and stride values (see Section 9.1.2), any program section created by inlining copies of these functions will contain an `envEnsure` before each loop due to the lexical references.

A method of removing `envEnsure` instructions using a combination of static optimisations and run-time tests on domains is given in Section 9.1.4.

9.1.2 Loop index variable ranges and strides

Part of the legality test for loop fusion is the requirement of conformability. For this, the induction variable range and stride of both loops must be known to be equal. This is trivial in the case of statically known numeric constants, but requires analysis in

the case of symbolic constants/variables and involves the implementation of lexical scoping by the compiler.

The first example we consider is that of two loops derived from two inlined functions from the same domain, where the loop control values (i.e. range and stride) belong to the scope of the domain. This is a very natural style for certain domains where all functions iterate over a fixed size object, e.g. loops over a vector (see Figure 2.2 for an example). Each loop in the inlined code is preceded by an `envEnsure` instruction, where the FOAM format (i.e. the type) of the environment being acted on is statically known in both cases, and therefore known to be the same.

For non-parameterised domains there can only be one domain of a given type, and so it follows that the lexical references refer to the same symbolic variable/constant. If the symbolic values are defined as constants then the loops must be conformable. If they are variables, some further analysis must be performed to show that they cannot change between uses (which again brings in control flow and the potential side-effects of functions and `envEnsures`). The case is more complex for parameterised domains, as there may be more than one domain object of the same type, and therefore the lexical references may refer to different symbolic constants/variables from different environments. One way around this is to insert dynamic checks in a similar fashion to those suggested for `envEnsures` in Section 9.1.4.

Interestingly, it ought to be possible to check that two parameterised domains are in fact the same object, and simultaneously ensure more lexically scoped symbolic values are defined as constants, by using the existing type system. All that is required is to encourage users to write parameterised domains with domain scope values as parameters (*unlike* Figure 2.2). The compiler already knows how to type-check expressions using parameterised domains to ensure that the same domain is referred to, and the implementation of the type system implies that arguments to parameterised domains are effectively considered as constants within the domain. However, this information is not currently transmitted down to the level of FOAM code.

The second example is similar to the first, but involves two loops taken one each from separate domains. An example of this would be a loop to apply a stencil from an operator domain, and a loop from a vector domain. In order for the loops to share

common lexical variables (for range, stride etc) and thus reduce the problem to being essentially the same as the first example, they would now have to refer to variables at the level where the domains themselves are defined – i.e. outside of domain scope. This is a less natural style. Again though, the existing type system ought to be able to help. The type-checking mechanism can ensure that two different domains have the same value for some parameter, and it ought to be possible to transmit the equivalence of two such symbolic constants to the level of FOAM.

9.1.3 All statement dependencies

Given a program section that conforms to the necessary restrictions and information on the loops it contains, the nodes of an LDG can be constructed. Discovering dependencies between these nodes that result from their statements requires alias analysis for array references, and dependence testing (an introduction to these subjects can be found in [65], although not in the context of LDGs; the standard tool for dependence testing is the Omega test [70]).

Alias analysis for Aldor ought to be simplified by the fact that the original operations are pure (i.e. they each allocate a new array to hold their results), objects cannot overlap in memory and thus partial aliases are not possible. In addition, for the types of program section considered here (i.e. taken from a recurrence), any object that is written to must have been allocated in the function due to the use of pure functions (assuming they have been inlined). Destructive update hints would also allow the compiler to immediately recycle objects, even within the same loop as they are read, as long as the relevant antidependencies are introduced into the LDG and not violated.

Static dependence testing for the programs covered in this thesis would be very simple once the conformability of two loops is known (see Section 7.3.3 for a description of the dependence structure).

9.1.4 Guarded sections

This section describes an approach to dealing with the `envEnsure` instruction in generated FOAM by creating sections of code where they have been removed.

Domains are initialised at most once. Executing an envEnsure instruction on a pointer to an environment that already exists does nothing. Consequently, it ought to be possible to eliminate a large number of them that can be shown statically (by means of data flow analysis) to be dominated by another envEnsure applied to the same environment. However, some may remain. In this instance, a section of code without envEnsure instructions can be created by duplicating it and splitting control flow such that either the original section of code or a new version with all envEnsure instructions removed is executed, depending on the status of all the environments that may be touched within the original section. If all the environments exist, the new section can safely be executed. This *guarded section* can be optimised more intensively than the original, as suggested above, as more precise information about the potential alteration of lexical environments is available.

An arbitrary section of code can be guarded, provided all the environments that may be touched within it are available to be tested at the head of the section. This includes unknown closures that are available at the start of the section, but unknown closures produced by other functions or fetched from nonconstant lexical variables during the section may not be safe (although this could be attacked with further analysis). Guarded sections can contain arbitrary branching, but sections with no forward branches have the benefit that the unguarded version will be executed at most once. This dovetails neatly to the program section for an acyclic LDG, as discussed in Section 8.1.

9.2 Prototype Implementation

The prototype implementation can be roughly split into two parts, the construction and manipulation of the loop dependence graph and its fusion partitions, and the final step of generating Aldor code for a fusion partition on the LDG. The whole prototype is written in Aldor. Note that construction of the LDG refers here simply to the building of the data structure using constructor functions rather than the analysis of a piece of code to recover its LDG.

9.2.1 LDG data structures

The construction of the LDG and fusion partition data structures is done using several libraries, which can be separated into layers, each building on the next with further specialisation. The bottom layer is general utilities and data structures for undirected and directed graphs, such as nodes, (directed) edges etc, and algorithms such as depth first search, topological sorting etc. To this are added the extra components necessary for an LDG, namely labelled edges (collapsible, non-collapsible etc). The fusion partition data structure over LDGs is built on top of this, along with the associated algorithms such as node numbering and enumeration. The final layer concerns the specifics of code generation, and associates loop bodies with loop nodes, specifying the uses of arrays (which arrays are read and written, and whether the result is a temporary), that give rise to the data dependencies already represented in the LDG. It also contains the methods to generate code from a fusion partition, part of which is the discovery of contractible temporaries.

9.2.2 Code generation

Generating code for a given fusion partition on an LDG is relatively simple. Each loop node in the LDG is labelled with the code from its body in a simple abstract syntax tree (AST) form. The code represented by the AST consists of variables and operations from the scalar and/or Wilson subdomain level, with the body of the stencil term operation to calculate the result at a specified site also available as a single function so that it may be incorporated into an arbitrary loop. The partition nodes are processed in order based on their ID number, and code generation proceeds one partition at time. To generate the code for a given partition, the loop nodes in its sub-graph are topologically sorted, the loop control code is written, and the bodies of the loops in this partition are then written out (unparsed) in order to form one large composite body.

There are several minor points of interest in code generation. The first is *preallocating space* for uncontracted temporaries by adding it to the lexical environment of the function from which the LDG is taken. This means that temporaries are allocated once for the entire run of the program, rather than having to allocate space for them for

each call of the function with the associated garbage collection overhead. Secondly, any array that has been labelled as contracted is replaced with a scalar temporary in the AST before it is unparsed. The last is *loop rerolling*, which applies to the Wilson problem. If a partition does not involve the Wilson stencil term in any way, then any operation involving spinor objects reduces to repeatedly applying a scalar operation to every element of the spinor object in turn. As such, a loop of dimension n over spinor objects can be replaced by a loop of dimension $n \times 12$ over complex double floats. This transformation is intended to ease pressure on the instruction cache.

The result of code generation is an Aldor source file consisting of loops and operations from the scalar domain/Wilson subdomain stub files, and some small files associated with temporary preallocation. These source files are included into the body of a driver function in the original code using preprocessor directives, and the whole lot is compiled to form part of an executable.

9.3 Issues with Prototype

There are a handful of issues with the prototype. The following sections describe several problems that arise due to limitations of the current compiler, and their workarounds. All workarounds exist within the language itself – thus they are mostly presentation or modularity issues. The final section describes a potential problem with iterative optimisation in the context of Aldor's run-time system and the prototype, and how it is taken into account.

9.3.1 Constant folding

Currently the Aldor compiler does not do constant folding for double precision floatingpoint values, despite the optimisation existing for single precision values. This causes a slight inefficiency in some functions from the `Projector` package, which can be circumvented by rewriting multiplication by a constant factor of i in terms of a special function that does the same thing.

9.3.2 Action of the inliner

In the case of type parameters to a functor, the information required for inlining functions from the parameter type only becomes available once the functor is instantiated – i.e. when domains are assembled together. In theory, given the static information in a top-level file that plugs together elements from a library of parameterised domains, the compiler could instantiate all domains and produce flat codes by using aggressive inlining (assuming a restricted use of the domain mechanism). There are a number of problems with this in practice though.

The inlining pass starts from the top-level in the call tree and expands child functions (including parameterised domains) based on some criteria, and subsequently adds any new child functions to the pool of candidates considered for inlining (see Section 2.3.1.1). In practice, the inliner usually runs out of steam before reaching the bottom level of the domain hierarchy, leaving the most basic domains to be handled via indirection. This is very inefficient.

Because of these problems with the inliner, some driver domains have no parameters. This is a simplification of the original design of the domains (see Appendix D), where type parameters are profitably used in numerous places. These include the Krylov space recurrence algorithm that very naturally generalises over the vector, operator and scalar domains that it manipulates. Avoiding type parameters means that the inliner has enough information to operate on each domain when it is compiled separately, in effect reversing the action of the inliner to bottom-up rather than top-down.

The use of parameterisation (and dependent types) is still present in its original form in the category hierarchy however, as its presence there does not affect code generation.

9.3.3 Inlining of generated code

There is no direct control over the compiler at the source level, so the only means of controlling the extent of optimisations is through coarse settings on the command line interface that are coupled in some way to the algorithms used by the compiler. After the removal of domain parameters as described above, inlining for the simple operator

codes is not generally a problem. However, even with the most aggressive settings the compiler sometimes has difficulty with the Wilson code, probably because the subdomain functions that are being inlined, and specifically the (body of the) stencil term itself, are much larger than their counterparts for the simple operator code. This problem manifests as a sensitivity to the command line settings, with a fine balance between not managing to fully inline all the stencil term and causing the compiler to crash or fail to terminate.

The remedy employed for the Wilson problem is to compile a special version of the body of the stencil term separately with aggressive settings and allow it to be called as a function from within the generated loops. The stencil term body takes the source vector, the gauge field, kappa and the index tuple of the site as arguments, and returns the value of a 4-spinor object manually flattened into a collection of machine double floats. A normal 4-spinor object cannot be returned due to the associated allocation and garbage collection overheads which would otherwise swamp performance. This flattening simulates what the emerger would to if the stencil term were properly inlined.

9.3.4 Emerging and unboxing hints

The emerger usually works well and removes all allocations for loops with double float running sums (see Section 2.3.1.3) and general use of domains whose representation uses a tree of boxed objects with multiple levels, such as the 4-spinor domain (which is a record of colour vectors, which themselves are records of complex double floats, which in turn are records of double floats).

The emerger does not however work fully for the accumulation of a complex double float scalar for an inner product, removing only one level of boxing. This leaves the allocation of two boxed double floats per iteration of the loop, which is enough to destroy the overall performance of the code. The remedy for complex double floats is to insert a custom function call, called an *unboxing hint*, into the source. The scalar used for the running sum is passed as an argument to the unboxing hint immediately after the inner product/norm loop, and the result becomes the value returned by the function. The unboxing hint itself does nothing except copy the contents of its boxed argument into two machine variables, and then copy the values from those variables

into a newly allocated complex double float which is then returned. The action of the inliner and emerger completely removes the hint function, but its existence in the source prompts the emerger to fully flatten the use of the running sum within the loop.

9.3.5 Data cache associativity

As mentioned in Section 7.2.3, loop fusion may increase the amount of live data for the body of a loop, thereby putting pressure on cache associativity. The form of the loops makes associativity problems unlikely as the amount of live data even for fused loops is very small and the caches on the architectures used have more associativity than simple direct mapping (see Section 10.3). However, it cannot be arbitrarily ruled out. The allocation of memory is entirely controlled by the run-time system, which means that potential associativity problems may occur "randomly" across runs of a binary, if the pattern of allocation is not deterministic. Checks for this behaviour were performed with several sets of tests which turned up little or no variation in the run-time, so this effect was considered not to be a problem (for the architectures concerned).

A possible means of attacking this problem in future is to alter the run-time system to be aware of the mapping that the cache uses so that the element of nondeterminism is removed, and the allocation of objects to lines in the cache could be exposed to enable a compiler to arrange object allocation to its own ends (e.g. to improve performance by getting rid of associativity conflicts).

9.4 Summary

The first half of this chapter summarised techniques to recover an LDG from the FOAM code generated by the Aldor compiler for programs derived from the algorithmic framework developed in this thesis. They include recovering control flow that has been lowered to the level of labels and branches, identifying conformable loops in the case of symbolic loop bounds, and recovering data dependencies between statements (and loop iterations). Most of the analyses are straightforward and well represented in the literature, but the implementation of lexical scoping and unpredictable control flow introduced by run-time artefacts of the type system require some more careful

consideration. Full investigation of these issues is left as future work.

The second half of the chapter introduced the prototype implementation (in Aldor) of iterative collective loop fusion used for the experiments detailed in Chapter 10. It consists of several libraries for the construction and manipulation of loop dependence graphs, and a system for generating Aldor source code to represent the resulting fusion partition. Due to the limitations of the current compiler, several simple modifications had to be made to the harness surrounding the generated code and the generated code itself. These included removing domain parameters and specialising a function due to problems with the inliner, and adding hints to circumvent a problem with the environment emerger.

Chapter 10

Experimental Results

This chapter presents results collected using the iterative collective loop fusion algorithm applied to an example taken from the application presented earlier in the thesis. The experiments are conducted on two machines with substantially different architectures and with three different operators. The results are compared against alternative methods of collective loop fusion (greedy and max-flow min-cut), and completely different methodologies such as using a different language (C/Fortran), and using assembly kernels.

The goals of the experimental work are as follows:

- To show that temporal locality is an important consideration for the style of programs discussed in this thesis, in that targeting it through loop fusion and array contraction provides significant speedups over the original programs.
- To show that iterative loop fusion gives better performance than the two other algorithms (under some assumptions about how they choose their solution).
- To motivate the use of search to compensate for the under-specification of the choice of transformation using contraction amount and/or partition size in the standard model. This is based on showing that transformations that result in the same amount of contraction and/or partition size can give substantially different performance.
- To motivate the use of search to compensate for the distance of optimum solu-

tions on the standard model (using the metrics of fusion and contraction) from the actual problem. This is based on showing that the heuristics are a reasonable guide but do not automatically give the best transformations, and that the best transformation differs for changes in hardware and the sub components of the LDG (changing the operator does not change the LDG itself).

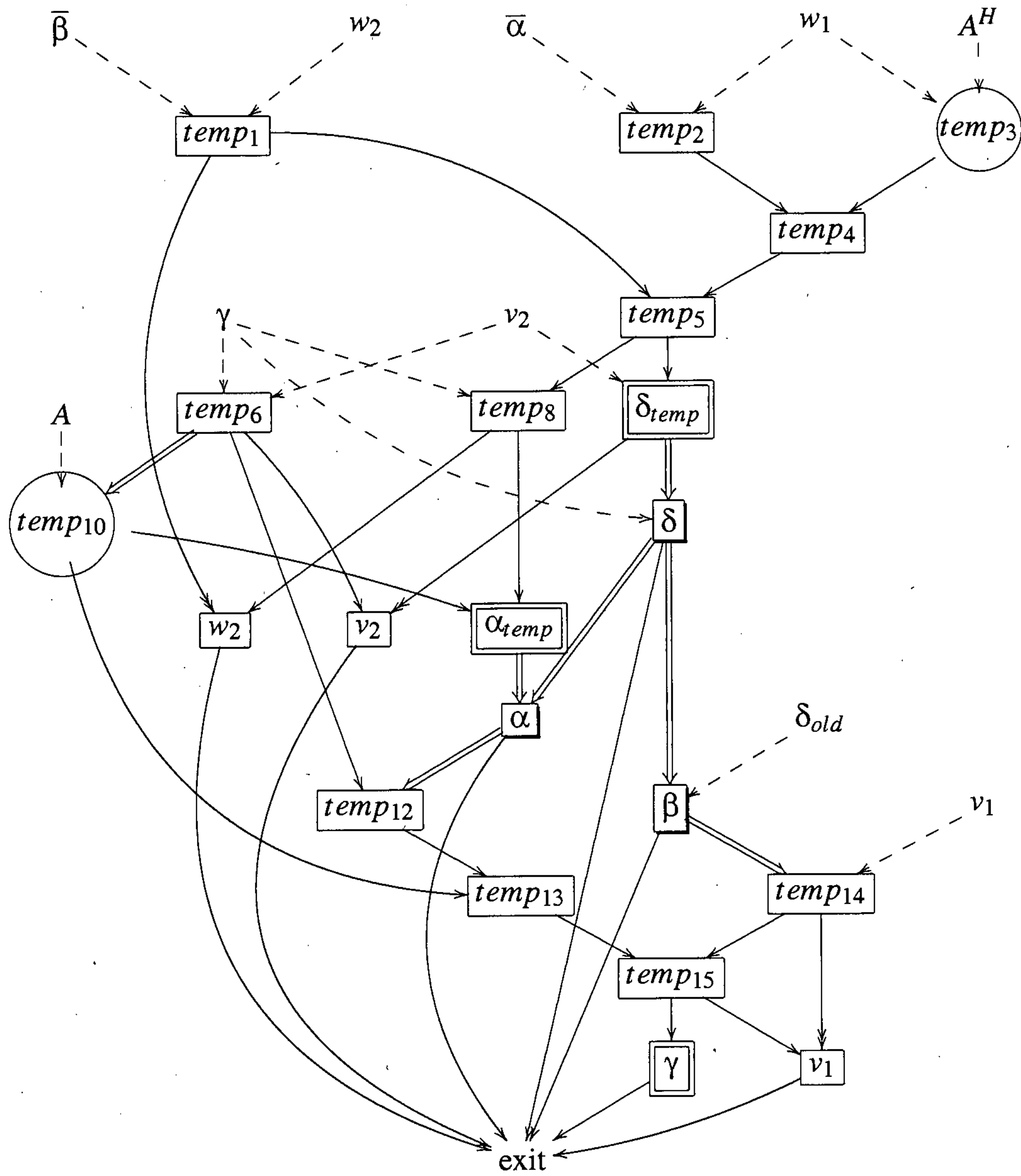
- To provide some approximate quantitative comparison against alternative methodologies.

The results show significant benefits from our technique, which gives a speedup of up to 3.7 over the original code. In addition, it outperforms the alternative methods with a speedup of up to 1.41 over them, and is comparable to the alternative methodologies.

10.1 Example LDG

The example in this chapter is derived from the code for the general step of the two-sided Krylov space update given in Figure 5.7. A decorated version of the associated LDG is presented in Figure 10.1. A number of changes have been made from the true LDG construct used for the experiments to make it more readable:

- The LDG is presented with the scalar basic blocks added rather than simply the edges resulting from dependencies carried by them.
- Dependencies to a pseudo-node called "exit" have been added to show when a value is still live at the end of the program section. This information is necessary to know if array contraction is possible, but it is not strictly speaking represented in the LDG as a node.
- The type of each loop node is identified.
- Input dependencies to data live-in to the program section have been added.
- Some redundant dependencies have been deleted to reduce clutter.



Key

	data live-in to LDG region	- - - →	input dependence
□	simple node	——→	true dependence
▣	reduction node	——↠	anti dependence
○	operator application (stencil) node	⟹	fusion preventing true dependence
❏	basic block		

Figure 10.1: LDG from two-sided Lanczos algorithm

Table 10.1: Basic LDG properties

property	value
total number of loop nodes	18
number of simple nodes	13
number of reduction nodes	3
number of operator application notes	2
total number of edges	24
number of fusion preventing edges	4
number of true dependencies (including fusion preventing)	21
number of antidependencies	3
number of output dependencies	0
minimum number of partitions	3
total number of array temporaries	12
maximum number of contracted arrays (as determined by search)	10

This LDG was constructed by hand for the purpose of the experiments, using the prototype described in Section 9.2. The different types of loop node and the form of the actual dependencies between them are described in Section 7.3.2. In this instance, the fact that possible dependencies from a simple node to an operator application node are obscured by the use of a dynamic data structure (the offset table) is irrelevant, as the loops are not directly fusible even if the exact dependencies are known.

Some information on the constructed LDG is given in Table 10.1 (note that the number of nodes and edges differs from Figure 10.1 as the scalar basic blocks and exit node do not exist in the actual LDG).

Table 10.2: Fusion partition information

FP size	no. legal FPs	time to enumerate (in minutes)	no. FPs with n contracted arrays				
			10	9	8	7	6
3	80	< 1	2	24	39	13	2
4	3557	< 1	4	174	960	1395	792
5	63801	< 4	2	366	4974	17066	22362
6	633799	< 57	0	307	10350	71951	178862

10.2 Enumeration of Fusion Partitions

The times taken to enumerate fusion partitions (FPs) of a particular size presented in Table 10.1 were recorded using a moderately loaded departmental compute server. They are intended as a rough guide as the implementation of the enumeration algorithm itself is not particularly efficient, and the time taken to test individual partitions dominates the overall cost of the approach.

Table 10.2 shows that the general method of starting with small fusion partition sizes is a good approach to the problem of enumerating and testing the points in the loop fusion/array contraction space. The (*contraction amount, partition size*) pair sets (see Section 8.2.1) in the bottom corner of the search space are of a reasonable size for this LDG and, from the points that were actually enumerated, it is possible to find some fusion partitions with the most contraction even with a very small amount of search in the very bottom corner of the space (i.e. partitions of size 3). Verifying the approach against the number of fusion partitions with maximum contraction in the whole space would be interesting but is almost certainly impossible due to its size, thus preventing a complete evaluation of the heuristics.

10.3 Evaluation Environment

10.3.1 Machines

The machines used for the experiments in this chapter were a 1 GHz Pentium 3 (Coppermine) and a 2.6 GHz Pentium 4 (Northwood). Both architectures have split level 1 instruction/data caches and a unified level 2 cache, with 4-way set associativity in the level 1 data cache and 8-way set associativity in the unified level 2 cache. The Coppermine has 32 kB each for the level 1 caches, 256 kB of level 2 cache, 256 MB of RAM and a 133 MHz frontside bus (FSB), whereas the Northwood has 8 kB each for the level 1 caches, 512 kB of level 2 cache, 512 MB of RAM and an 800 MHz FSB. As well as different clock rates and memory hierarchies, the two processors have substantially different internal organisation, including functional unit characteristics, pipeline – 12 stages for the Coppermine, and 20 stages for the Northwood – and slightly different instruction sets (different short vector extensions). The operating system in both cases is version 2.4.20 of the Linux kernel.

The choice of machines used for the experiments was limited by the development status of the Aldor compiler. Although the compiler and its associated run-time system have been ported to various UNIX platforms in the past, at the time of the experiments the primary supported platform was x86/Linux. Currently, the two major suppliers of binary compatible x86 processors for mid- to high-performance workstations are AMD and Intel. AMD CPUs were not used to due to their reliance on direct mapped caches and the possible problems this could pose for collecting performance results (see Section 9.3.5). The only other recent Intel CPUs that could be used are those based on the Prescott and Dothan[1] cores, which bear some similarity to the Northwood and Coppermine cores respectively, but these are left for future work alongside gathering results from more different architectures (see Section 11.2.5).

[1]Or later Pentium M derivatives.

10.3.2 Compilers

The compilers were the Aldor compiler, version 1.0.1 generating C, the Intel C compiler `icc` version 8.0 (used to compile the output from the Aldor compiler and any C based codes), the Intel Fortran compiler `ifc` version 8.0, and the GNU C compiler `gcc` version 3.2.2 (used to compile the assembly code discussed in Section 10.6.2.3).

Optimisation switches for the Aldor compiler were chosen as follows:

- `-Q9` – to give a high level of optimisation and allow large amounts of inlining.
- `-inline-all -inline-limit=15` – set inlining to very aggressive.

Optimisation switches for the `icc`/`ifc` were chosen as follows:

- `-O2` or `-O3` – C code generated from Aldor was compiled using `-O2` to invoke general-purpose low-level optimisations such as register allocation, scheduling, common subexpression elimination etc. but avoid high level loop transformations such as fusion, distribution, tiling, interchange etc. This option can also enable vectorisation and unrolling (see Section 10.7.3). For the Fortran programs, the choice between the two was made empirically to give the best performance. In practice there was no difference in code generated.
- `-xK` (Pentium 3) or `-xN` (Pentium 4) – code generation specifically targeted at the architecture in question (including SSE1/2 instructions).
- `-static` – static linking of libraries.
- `-align` – use aligned loads where possible.
- `-prefetch` – insert software prefetch instructions.
- `-ip` or `-ipo` – do intra-file (for `icc` on generated C) or inter-file (for `ifc` on Fortran) interprocedural optimisations.
- `-p` – instrument for profiling.

The low level code generated by `ifc` and `icc` are discussed in Section 10.7.3. Optimisation switches for `gcc` were not used as the compiler only serves to stitch together assembly macros.

10.3.3 BLAS routines

Any reference to BLAS routines refers to the level 1 BLAS binaries from the ATLAS project [96] (version 3.4.2) for the respective machines. Note that these routines are highly tuned assembly rather than compiler generated code, unlike the non-binary distribution or the high-level (2 and 3) BLAS routines generated by local search using a machine and its compiler.

10.3.4 Generation of timing results

Profiles of executables were generated by instructing the Fortran/C compilers to instrument the code for profiling, and processing the results using the GNU profiling tool `gprof` version 2.13.90.0.18. A single performance figure is taken as the total time reported in the profile, and this measure is used everywhere except in Section 10.6.2 where a breakdown provided by the profile is discussed. Profiling is done when only a single figure is required in order to check that the amount of time spent in domain initialisation and garbage collection is negligible.

Times for a given code were generated by supplying a numerically difficult problem (but not one that is ill-posed enough to cause floatingpoint exceptions) and running the Krylov space generation procedure or full iterative solver for 1000 iterations. These long runs were used to minimise noise in the results from sampling inaccuracies and minimise the relative cost of run-time domain initialisation. All times are given in seconds throughout the rest of this chapter.

Instrumented code is generated by the compiler at the entry and exit of any function that is compiled for profiling. Aldor code generation for the 3D and 4D operator programs (see below for a description of the programs used for experiments) results in a single C function that is invoked once per iteration. The code for the Wilson-Dirac operator program contains one additional function call within an iteration to the oper-

ator itself, which takes a significant amount of time relative to the rest of the iteration. Consequently, potential interference from code instrumentation was considered to be minimal. The use of library routines in Section 10.6.2 introduces more instrumentation code at call sites (although the library code itself is not instrumented), but this was again considered minimal due to the size of the vectors that each routine manipulates for the Wilson-Dirac problem (their minimum size is 243kB per vector for a 6^4 grid).

10.4 Iterative Search Experiments

The iterative search experiments show how iterative collective loop fusion might be used for the example LDG, and also give a general picture of the loop fusion problem in a concrete setting. This amounts to testing the sets of (*contraction amount*, *partition size*) pairs in Table 10.2 with reasonable sizes – i.e. the top row, the first column and most of the second column. Throughout the rest of the thesis, (*contraction amount*, *partition size*) is abbreviated as (c, p) – for example, $(9c, 4p)$ denotes the set of fusion partitions with four clusters and nine contracted arrays, and $(10c, 3\text{-}5p)$ denotes the union of $(10c, 3p)$, $(10c, 4p)$ and $(10c, 5p)$.

Sets of (c, p) points were tested for different operator types (3D, 4D and Wilson-Dirac as described in Chapter 6), machines (Pentium 3/4), and data set sizes. Note that the Wilson-Dirac operator is only tested on the Pentium 4 as the Pentium 3 was insufficiently powerful to run problems of a reasonable size and does not have the SSE2 instructions used by the assembly control in Section 10.6.2. Data set sizes are given as the size of an individual vector, in terms of the length of one side of the uniform grid that the vector represents. For the 3D and 4D problems, each site in the grid is 16 bytes, whereas for the Wilson-Dirac problem each site is 192 bytes (ignoring the gauge field) – see Chapter 6. Vector size ranges are intended to give examples of problems that are feasible to solve on an individual workstation. For each size range and machine, the smallest vector size fits within the level 2 cache and the largest vector size is larger than the level 2 cache.

Note that for any (c, p) set that gets tested, all of its points are executed, so the heuristics for choosing a subset of points to test within a given (c, p) set are not used.

However, the general approach of generating the elements of (c, p) sets with smaller fusion partitions and/or more contraction is kept.

The results are grouped by operator, then machine, then data set size. Points on the plots are labelled by the set they come from (i.e. a different point type for each (c, p) set). This is denoted in the legend by the amount of contraction r and the size of the partition s respectively in the shorthand form r**c**s**p**. The vertical axis denotes execution time, so points closer to the bottom of a plot give better performance. The horizontal axis denotes the rank of a point within a (c, p) set. Results for a (c, p) set are sorted based on execution time before plotting (fusion partitions are not generated in an order correlated with performance), so partitions of the same rank on different plots do not necessarily correspond to the same fusion partition. The ranking is reversed (i.e. larger number is *better*) to prevent all the important points bunching up in one corner, so points further to the right within a (c, p) set have better performance. In addition, the result of some sets are smeared out along the x-axis to improve visibility.

10.4.1 3D operator

10.4.1.1 Pentium 3

The following three plots show the performance of fusion partitions from various (c, p) sets for the 3D stencil problem, on the Pentium 3. The different plots correspond to different data set sizes.

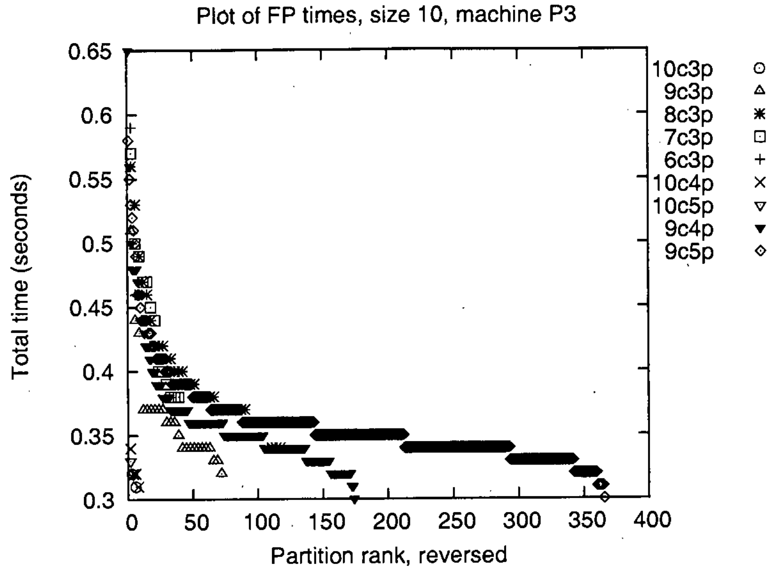


Figure 10.2: Performance of fusion partitions from various (*c*, *p*) sets for the 3D problem on the Pentium 3 with grid size 10^3

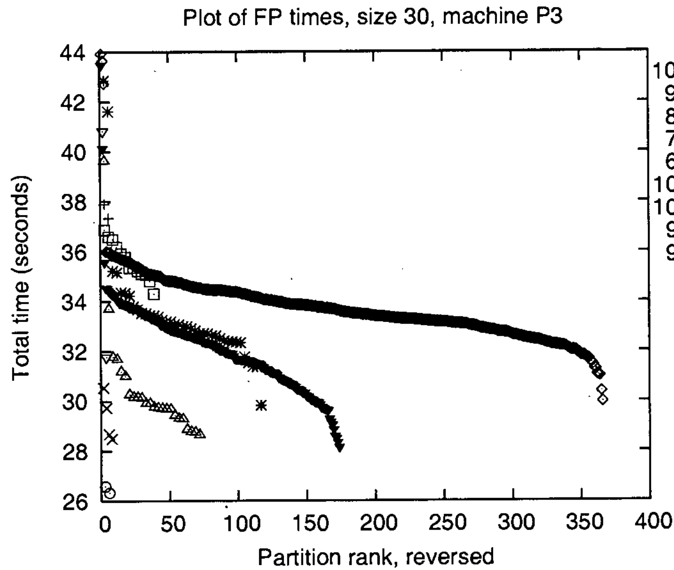


Figure 10.3: Performance of fusion partitions from various (*c*, *p*) sets for the 3D problem on the Pentium 3 with grid size 30^3

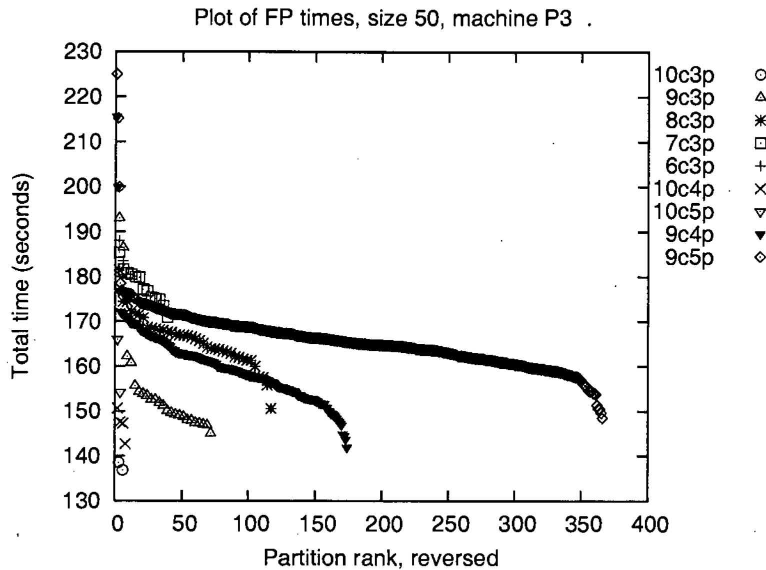


Figure 10.4: Performance of fusion partitions from various (c, p) sets for the 3D problem on the Pentium 3 with grid size 50^3

In the latter two plots, the two members of ($10c$, $3p$) down in the bottom left corner have the best performance, whereas in the first plot the best members of ($9c$, $4p$) and ($9c$, $5p$) are ranked equal best. The step-like banding of performance in the first plot arises from quantisation effects due to the executable having a short run-time.

For the two large (c, p) sets, the plots show fairly large difference in performance between the best and worst points. There is a tendency for differences between neighbouring points to be small across the majority of the set, but they get larger at either extreme with the largest jumps occurring between the worst performers (toward the top of the plots on the left-hand side). This is also reflected in the smaller (c, p) sets. The amount of array contraction appears to be the biggest factor in determining performance when comparing the best performers from different sets, with more contraction being better. Some results run counter to this though – for example, the best fusion partition from ($8c$, $3p$) in the second plot is better than the best fusion partition from ($9c$, $5p$). Also, there is significant overlap between sets over their full range, and size of fusion partition does play a role. For example, there are many members of ($9c$, 3-$5p$) that are better than the worst member of ($10c$, $5p$) in the third plot, and in all

three plots the best member of $(9c, 4p)$ is better than the best member of $(9c, 3p)$ even though its fusion partitions are larger.

Changing data set size does not appear to dramatically affect the overall shape of a set, but does appear to alter the position of the sets relative to one another. Note that the sorting of sets by performance before plotting means that the points on different plots in the same position do not necessarily correspond, and therefore the same shape of (c, p) set on different plots may result from a completely different order of rank for the fusion partitions involved.

10.4.1.2 Pentium 4

The following three plots show the performance of fusion partitions from various (c, p) sets for the 3D stencil problem, on the Pentium 4. The different plots correspond to different data set sizes.

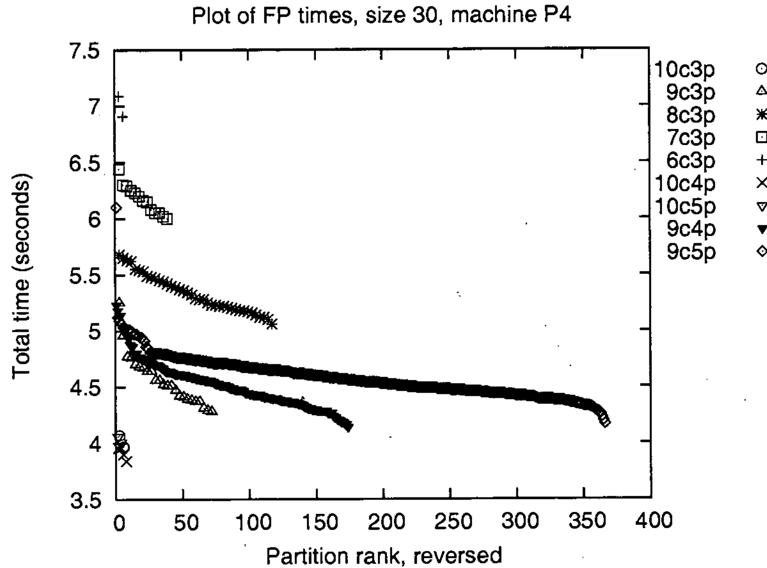


Figure 10.5: Performance of fusion partitions from various (c, p) sets for the 3D problem on the Pentium 4 with grid size 30^3

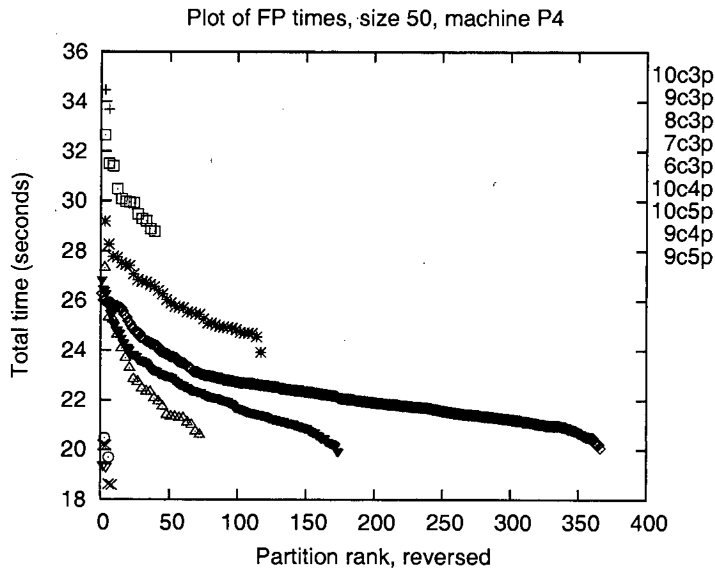


Figure 10.6: Performance of fusion partitions from various (*c*, *p*) sets for the 3D problem on the Pentium 4 with grid size 50^3

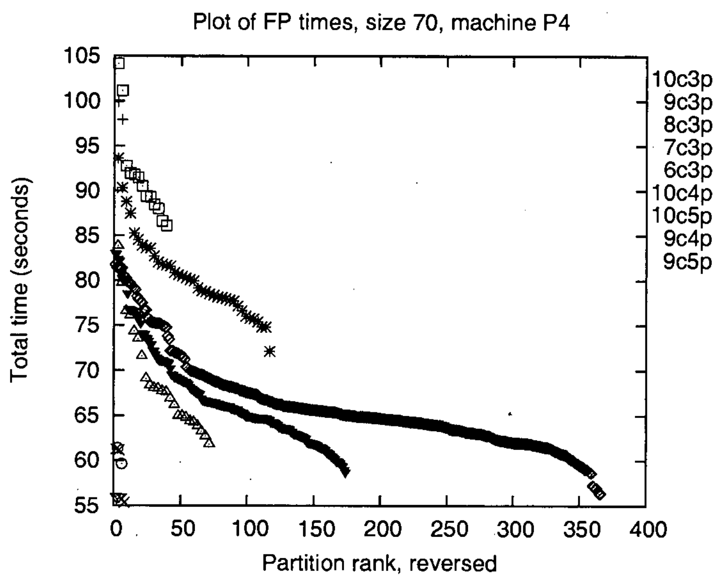


Figure 10.7: Performance of fusion partitions from various (*c*, *p*) sets for the 3D problem on the Pentium 4 with grid size 70^3

For the Pentium 4, the larger fusion partitions with the maximum contraction (i.e. members of (10*c*, 4-5*p*)) give the best performance across all data set sizes, unlike the Pentium 3 where members of (10*c*, 3*p*) give the best performance for larger problem sizes. Apart from this, the analysis of the plots for the Pentium 3 largely carries over to the Pentium 4. However, the difference in performance between neighbouring points at either end of a set is less and the separation between sets is somewhat clearer. Also, array contraction seems more dominant in determining performance, and there is less overlap between the (*c*, *p*) sets with the maximum contraction (10) and those with less. Somewhat surprisingly, the best members of (9*c*, 5*p*) are close to the absolute best performance for the larger data set size.

10.4.2 4D operator

10.4.2.1 Pentium 3

The following three plots show the performance of fusion partitions from various (*c*, *p*) sets for the 4D stencil problem, on the Pentium 3.

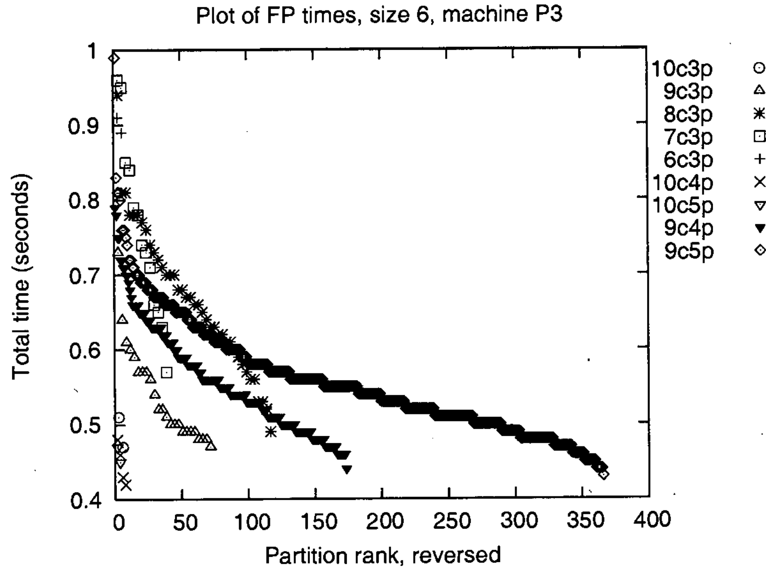


Figure 10.8: Performance of fusion partitions from various (*c*, *p*) sets for the 4D problem on the Pentium 3 with grid size 6^4

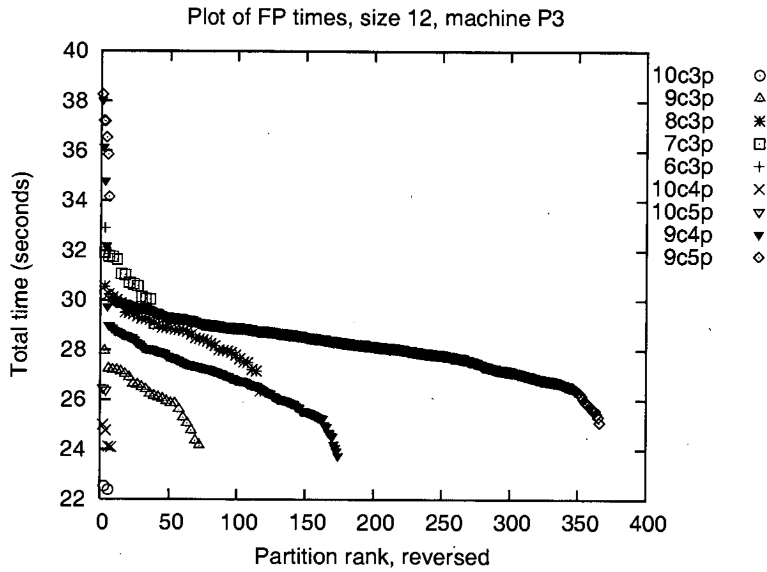


Figure 10.9: Performance of fusion partitions from various (*c*, *p*) sets for the 4D problem on the Pentium 3 with grid size 12^4

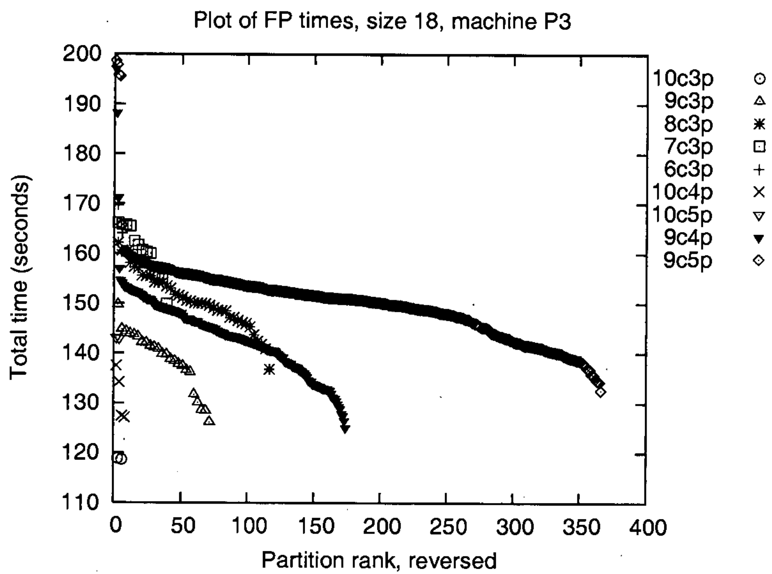


Figure 10.10: Performance of fusion partitions from various (*c*, *p*) sets for the 4D problem on the Pentium 3 with grid size 18^4

Broadly similar patterns to those described in the analysis for the 3D operator on the Pentium 3 apply to these plots for the 4D operator. However, the best result in the first plot is a fusion partition with maximum contraction from (10c, 4p) , rather than (10c, 3p) in the latter two plots.

10.4.2.2 Pentium 4

The following three plots show the performance of fusion partitions from various (c, p) sets for the 4D stencil problem, on the Pentium 4. The different plots correspond to different data set sizes.

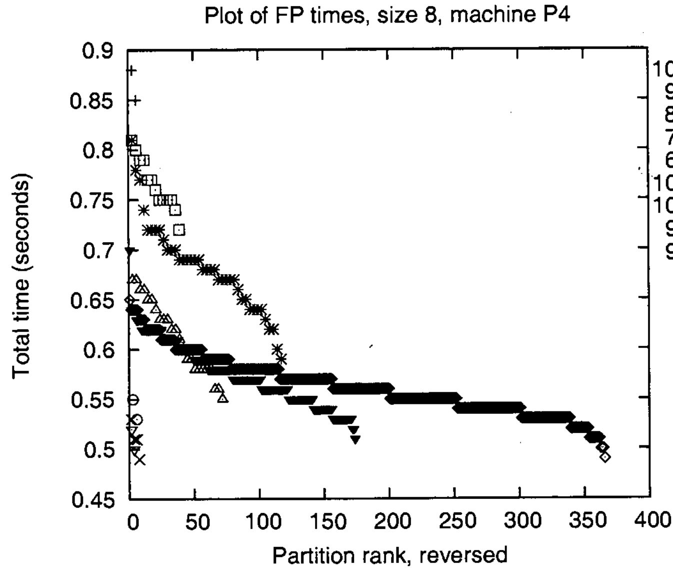


Figure 10.11: Performance of fusion partitions from various (c, p) sets for the 4D problem on the Pentium 4 with grid size 8^4

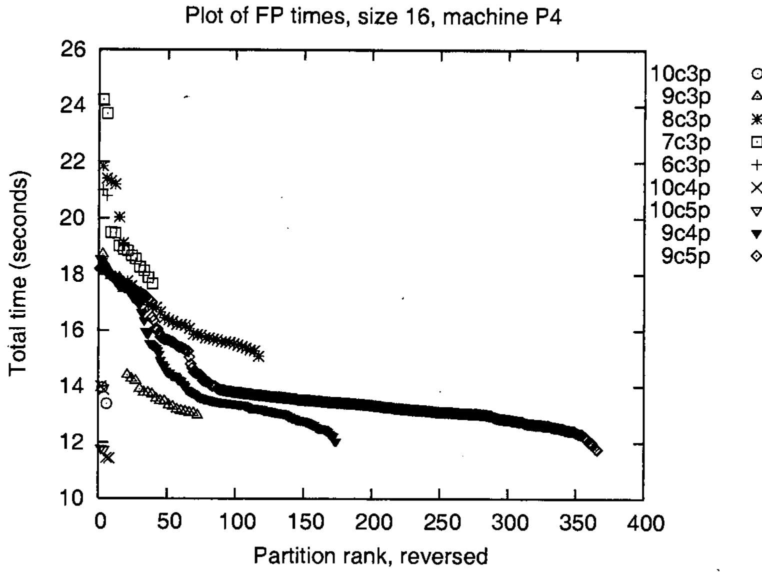


Figure 10.12: Performance of fusion partitions from various (*c*, *p*) sets for the 4D problem on the Pentium 4 with grid size 16^4

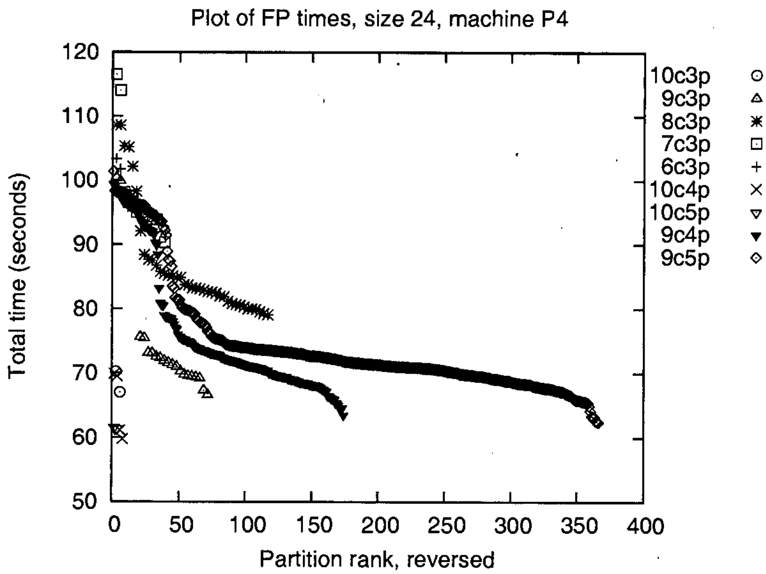


Figure 10.13: Performance of fusion partitions from various (*c*, *p*) sets for the 4D problem on the Pentium 4 with grid size 24^4

The plots are similar to those for the 3D operator on the Pentium 4. The latter two plots show a lessening of the separation between the worst performers in a given set, giving a sharper step down between ranks 50 and 100, and all three plots suggest that the influence of array contraction is less dominant as there is more overlap amongst sets. The best performance comes from the larger fusion partitions with the maximum contraction, (10*c*, 4-5*p*) , across all data set sizes, but interestingly the best members of (9*c*, 4*p*) and (9*c*, 5*p*) come close. The first plot shows similar quantisation effects to earlier plots from small data set sizes on the Pentium 3.

10.4.3 Wilson-Dirac operator

10.4.3.1 Pentium 4

The following five plots show the performance of fusion partitions from various (*c*, *p*) sets for the Wilson-Dirac stencil problem, on the Pentium 4. The different plots correspond to different data set sizes. Results for an extra (*c*, *p*) set are plotted for this operator (namely (9*c*, 6*p*)) to investigate the anomalous performance behaviour.

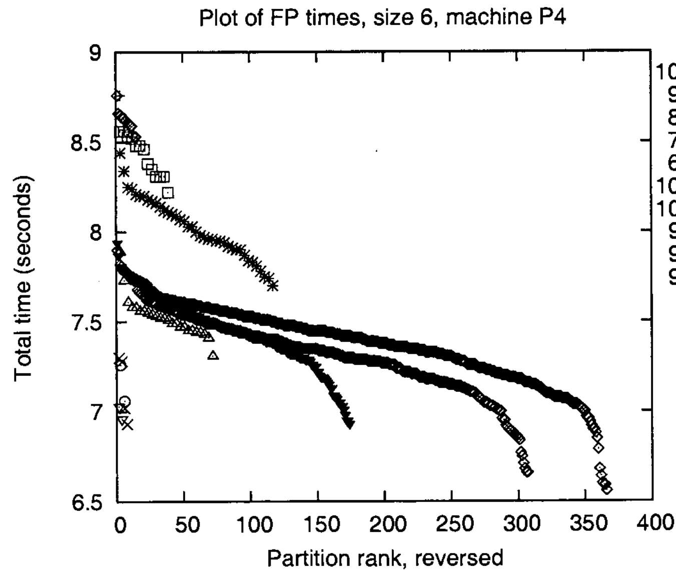


Figure 10.14: Performance of fusion partitions from various (*c*, *p*) sets for the Wilson-Dirac problem on the Pentium 4 with grid size 6^4

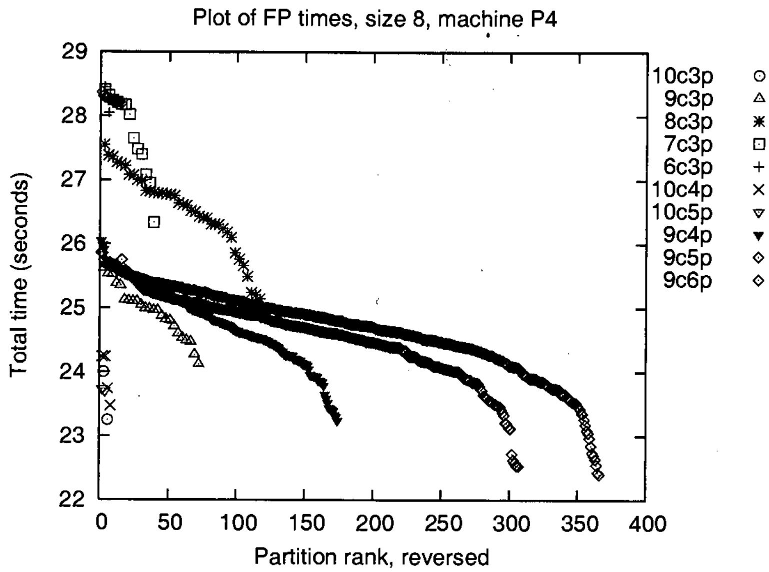


Figure 10.15: Performance of fusion partitions from various (c, p) sets for the Wilson-Dirac problem on the Pentium 4 with grid size 8^4

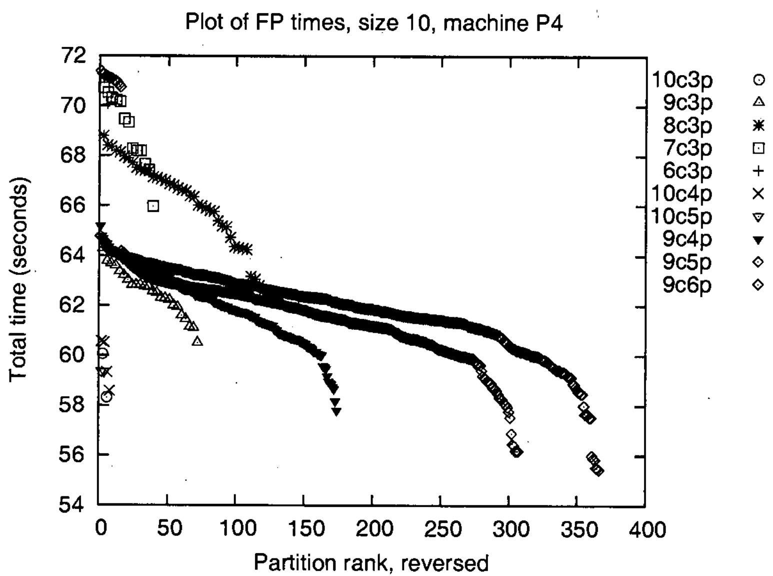


Figure 10.16: Performance of fusion partitions from various (c, p) sets for the Wilson-Dirac problem on the Pentium 4 with grid size 10^4

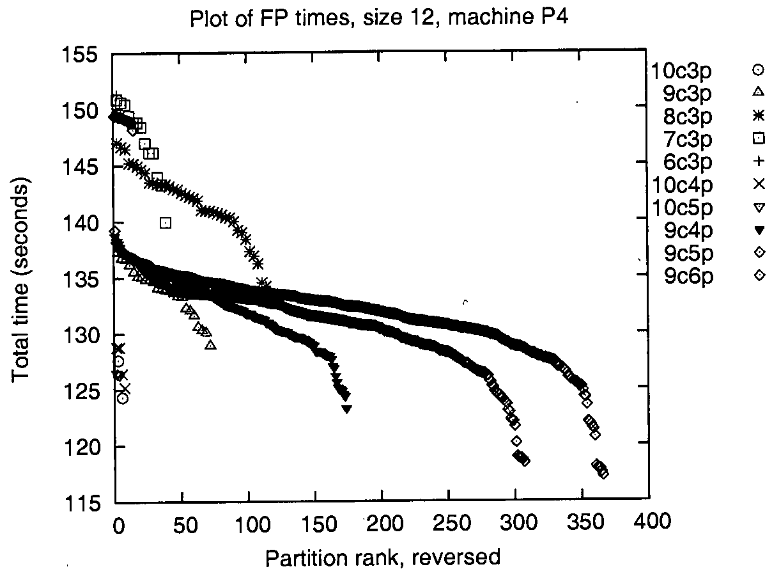


Figure 10.17: Performance of fusion partitions from various (c, p) sets for the Wilson-Dirac problem on the Pentium 4 with grid size 12^4

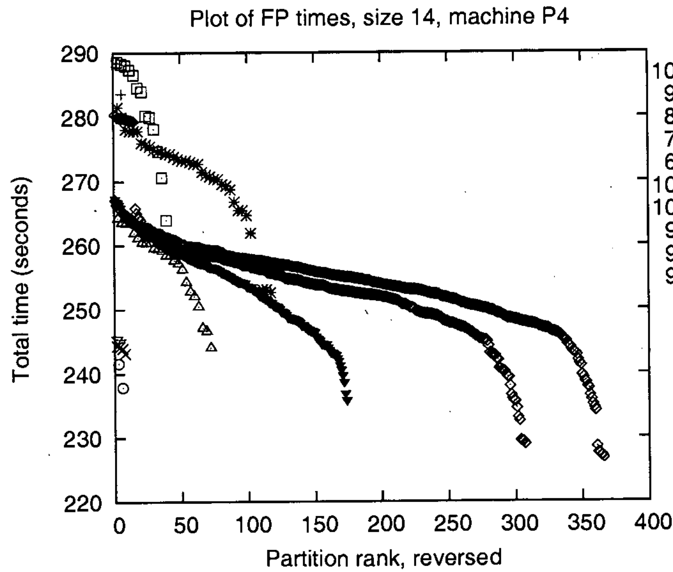


Figure 10.18: Performance of fusion partitions from various (c, p) sets for the Wilson-Dirac problem on the Pentium 4 with grid size 14^4

The above plots show that, somewhat surprisingly, the best performance is no longer given by a fusion partition with the maximum contraction (although the difference is not pronounced), but by members of $(9c, 6p)$. See Section 10.7.2. Also, performance across a (c, p) set is less consistent than for the other problems. This can be seen by the more pronounced dip as performance nears the best result for any given set (i.e. toward the right hand side), which gets more exaggerated as the data set size increases, to the point of having a separate cluster of "best" points.

10.4.4 Variability within (c, p) pair sets

This section aims to give some idea of the potential inaccuracy of assuming that all members of a (c, p) set produce equal performance on an actual machine.

Variability within (c, p) sets is presented as the time of the worst result as a percentage of the best for a combination of machine, operator type, and data set size. Only operator and machine are given as separate axes, as including data set size as well would make the tables too large. Instead, the largest variation out of all the data set sizes is used. The results are presented in Table 10.3, and clearly show the degree of inaccuracy resulting from treating all members of a (c, p) set as equal. Any broader grouping, such as contraction amount alone, will be even less accurate. Of particular interest are the sets of size 3 and 4 with the maximum contraction (10), as these sets usually contain the best result.

Examining the results shows that search is a useful addition to loop fusion and array contraction, due to the under-specification of the standard approach to the problem – that is, relying purely on contraction amount or fusion partition size. In the particular instance of the LDG presented here with the 3D and 4D operators, the three (c, p) pair sets containing the best results measured by contraction are small (eight elements in total) and have fairly low variability across their combined members. However, it is clear that in general the larger (c, p) pair sets have high variability (up to 130%), so different LDGs with larger numbers of fusion partitions with the maximum contraction stand more to gain by the application of the technique. Also, *all* the sets with the maximum contraction ought to be tested as the smallest fusion partitions with the maximum contraction are frequently not the best, and LDGs that have lots of different

Table 10.3: Variability within (*c*, *p*) pair sets (as % of best)

	Pentium3		Pentium4		
(c, p) pair set	**3D**	**4D**	**3D**	**4D**	**Wilson-Dirac**
($10c$, $3p$)	3	8	4	4	3
($9c$, $3p$)	59	55	35	50	8
($8c$, $3p$)	64	91	29	44	11
($7c$, $3p$)	50	68	20	37	9
($6c$, $3p$)	28	3	2	3	2
($10c$, $4p$)	9	14	10	22	5
($10c$, $5p$)	28	4	1	4	1
($9c$, $4p$)	116	79	41	56	14
($9c$, $5p$)	93	130	46	62	20
($9c$, $6p$)					31
($10c$, 3-$5p$)	55	21	10	22	5

size fusion partitions with the maximum possible contraction ought to see significant benefit. For example, the maximum variation across different size partitions with the maximum amount of contraction (i.e. ($10c$, 3-$5p$)) is 55%, which gives a more accurate picture of the need for search to find a good solution. The amount of potential gain is proportional to effort – small sets are cheap to search, and larger sets typically have more variability that must be dealt with – so the technique introduces little overhead when there is less gain to be exploited.

10.4.5 Variability across setting

Variability across operator/machine/data set size shows the necessary limitations of any static approach that makes its decisions based purely on the LDG. Assuming that some hypothetical static method picks the fusion partition that gives the best known result in a given setting (i.e. combination of operator/machine/data set size such as the 3D operator on a Pentium 3 with data set size 50), that particular fusion partition may not be the best for other settings (e.g. 4D operator on a Pentium 4 with data set size

Table 10.4: Best case time for 3D vs. FP from best case for other settings

machine	size	best	best case FP from other: operator (4D)	operator (Wilson)	machine
Pentium3	10	0.30	0.32	0.32	0.34
	30	26.3	29.7	33.25	30.5
	50	136.8	147.2	162.0	147.2
Pentium4	30	3.83	3.90	4.48	4.55
	50	18.5	18.6	21.2	21.7
	70	55.34	55.7	59.4	62.7

70). This can be highlighted by picking a given setting, collecting the fusion partitions that were found to be the best in other settings and trying them on this problem. The discrepancy between the worst result from the other "best" fusion partitions and the actual best for this problem gives some measure of how any method that chooses its result based purely on information from the LDG lacks portability across operator type, machine etc.

Rather than varying all the options at once, we limit ourselves further by only changing one of the three (i.e. operator, data set size, machine) at a time, which gives an even stronger result. This excludes the Wilson problem however, as the experiment was not done on the Pentium 3, and so comparing it against results for that machine necessarily involves changing both operator and machine. The results are given in Tables 10.4, 10.5 and 10.6. Data set size is not given as a separate option as it tended to give little or no variation.

The results show that picking the optimum LDG transformation in a given setting can lead to missing the optimum in a different setting by $> 30\%$. An important advantage of search is that it offers portability by coping with the different choice of best fusion partition as the data set size, machine or program (operator) changes.

Table 10.5: Best case time for 4D vs. FP from best case for other settings

			best case FP from other:		
machine	**size**	**best**	**operator (3D)**	**operator (Wilson)**	**machine**
Pentium3	6	0.41	0.51	0.50	0.46
	12	22.3	28.8	28.0	24.9
	18	118.6	152.4	156.6	137.5
Pentium4	8	0.49	0.50	0.52	0.55
	16	11.4	11.4	12.4	13.3
	24	59.75	61.1	65.8	67.0

Table 10.6: Best case time for Wilson vs. FP from best case for other settings

			best case FP from other:	
machine	**size**	**best**	**operators (3/4D) P3**	**operator (3/4D) P4**
Pentium4	6	6.55	7.26	7.01
	8	22.3	24.6	23.7
	10	55.4	61.5	59.3
	12	117.2	130.8	126.3
	14	226.7	253.6	243.9

10.5 Control Experiments

To give some context for the best results achieved using iterative collective loop fusion in the previous section, we compare the results with some control experiments. The first control is simply the original Aldor programs with no fusion or contraction to show whether the optimisations improve program performance. The second set of controls are alternative (static) loop fusion strategies taken from the literature. The third set of controls are completely different methodologies. These include a functionally equivalent program written in Fortran, and combinations of assembly code and C code.

10.5.1 Naive control

This is simply the original program with no fusion (and hence no contraction) applied. The results are given in Tables 10.7, 10.8 and 10.9.

Loop fusion and array contraction clearly bring important benefits on these machines. The range of speedups against the naive control over the different experiments is $2.1 - 3.7$ for the 3D/4D problem, and $1.38 - 1.43$ for the Wilson problem.

10.5.2 Other methods of collective loop fusion

To give some idea of how other methods from the literature compare to fusion partition enumeration, we implemented versions of the max-flow min-cut heuristic method presented in [37], and the simple pair wise greedy algorithm presented in [52]. The more complex version of the algorithm presented in the latter paper is not considered as it is functionally equivalent to the simple method, and the time to compute the fusion partitions calculated by the controls is not the main focus, but rather which fusion partition they produce.

In both control algorithms not all steps are completely specified, so how they are implemented will affect which fusion partition they produce. The most extreme of the two is the greedy algorithm. Given an LDG in which all edge weights are equal, there may be many pair wise collapsing actions that are ranked equally at a given step of the algorithm, and therefore many different fusion partitions could be generated depending

Table 10.7: Times from best search and control methods for the 3D problem

machine	size	best search	greedy	max-flow min-cut	naive
Pentium3	10	0.30	0.37	0.37	0.72
	30	26.3	33.7	32.7	69.5
	50	136.8	186.3	163.6	329.6
Pentium4	30	3.83	4.3	5.1	9.4
	50	18.5	21.3	24.8	43.8
	70	55.3	64.2	76.6	122.3

on how a selection is made from equally ranked alternatives. Thus, fusion partitions of different sizes with different numbers of contracted arrays are possible depending on this degree of freedom. For the max-flow min-cut method, any subgraph that must be partitioned may have a number of different minimum cuts that are ranked equal. So, this method gains over the greedy algorithm in that it will always produce fusion partitions of the same size and contract the same number of arrays, but the particular choice of fusion partition with that degree of contraction is still not specified.

This makes providing a precise set of control experiments difficult. Instead, we attempted to measure the cases that can result by varying this degree of freedom, and compare them to enumeration. For the both algorithms, these points were determined by some brute force search over the tree of possible transformation sequences resulting from multiple equally ranked choices at any given step. The whole space of choices could not been enumerated as it is too large [2]. The results are given in Tables 10.7, 10.8 and 10.9.

From the relatively small amount of space that could be searched in the time available, the greedy algorithm always produced a $(9c, 3p)$ fusion partition, and max-flow min-cut produced a $(8c, 3p)$ fusion partition.

The results show that iterative collective loop fusion is consistently better than the greedy or max-flow min-cut methods. The argument is complicated by the fact

[2]Although many different sequences of choice for the greedy algorithm result in the same fusion partition, it is not possible to directly list the distinct fusion partitions that will be created without considering the choices involved in generating them.

Table 10.8: Times from best search and control methods for the 4D problem

machine	size	best search	greedy	max-flow min-cut	naive
Pentium3	6	0.41	0.56	0.58	1.53
	12	22.3	26.6	28.4	55.2
	18	118.6	141.3	148.5	291.1
Pentium4	8	0.49	0.67	0.68	1.08
	16	11.4	13.1	15.6	24.4
	24	59.7	69.7	79.2	126.9

Table 10.9: Times from best search and control methods for the Wilson problem

machine	size	best search	greedy	max-flow min-cut	naive
Pentium4	6	6.55	7.48	8.00	9.42
	8	22.3	24.9	26.4	31.9
	10	55.4	63.0	66.8	79.1
	12	117.2	133.9	140.5	164.3
	14	226.7	261.9	270.9	314.6

that the results given by these methods are not entirely specified, but it holds under some reasonable assumptions about their efficacy. Our technique offers a speed up of 1.36 over worst-case greedy and 1.41 over worst-case max-flow min-cut for the 3/4D problem, and 1.15 and 1.22 respectively for the Wilson problem.

10.6 Other methodologies

This set of controls is intended to give some idea of how iterative collective loop fusion as a tool compares against other approaches. Two alternatives are considered. The first is a version of the QMR algorithm implemented in Fortran (the full solver had to be used for the comparison, as described below). The second is a set of experiments based on the original Aldor program for the Krylov update step considered in the previous sections, with alterations to use high-performance binary BLAS routines to manipulate vectors.

10.6.1 Fortran 3D stencil

This program is a version of the 3D stencil code written completely in Fortran 77 and compiled from source. Fortran is widely used for numerical scientific computation, so this control provides important context for the optimisation results.

QMRpack is a well-known and freely available implementation of the QMR algorithm [33]. The code does not come with any implementation of the operator, but is designed to be joined to code to calculate applications of an operator by means of callbacks. Vector operations are handled using standard level 1 BLAS style routines, supplied as and compiled from source. Note that this means that some degree of fusion is already built in due to the form of the BLAS routines.

The original code from QMRpack had to be modified/supplemented in several ways. Firstly, as there is no operator we added Fortran code to apply the stencil. Secondly, the code is designed to skip certain steps of the algorithm based on floatingpoint error tolerances, and these branches were removed to make the amount of "work" (in terms of vector operations) fixed and equal to that of the Aldor code. Finally, all the source files for the whole program were compiled using the cross-file inlining options

Table 10.10: Times for linear solve (search vs. Fortran)

machine	size	best search	Fortran
Pentium3	10	0.58	0.36
	30	43.4	64.1
	50	209.1	303.7
Pentium4	30	5.26	7.20
	50	24.5	33.6
	70	71.6	95.6

of `ifc` to remove any artificial barriers to optimisation by the compiler.

One important difference needs to be highlighted for this control experiment. All other experiments in this chapter deal with an LDG derived from a Krylov space update procedure. This is primarily because this procedure dominates execution time for a two-sided solver and gives by far the largest LDG with the richest structure, and hence the most interesting and challenging optimisation problem. There is no reason why the same technique could not be applied to the update steps for the other recurrences, but a lack of time prevented us from performing the experiments. However, because QMRpack is an entire solver, the comparison must be performed against a full Aldor solver assembled from the three separate recurrences. For this we used the best fusion partition of the two sided Lanczos process discovered by search, along with the search recurrence and solution update recurrence (from the template) described in Chapter 5, each of which were separately fused and contracted by hand. Both of the search and solution update recurrence can be trivially fused into a single loop with the maximum possible array contraction. The results are presented in Table 10.10.

Although it is hard to give a precise measure when comparing substantially different approaches to a problem, it is clear from the results that the performance of programs compiled in Aldor using iterative collective loop fusion is at the very least competitive for these types of problem with an equivalent program written entirely in Fortran and compiled by a mature commercial optimising compiler. The Aldor version does better in all but one case, with the relative performance gain being ≈ 1.46 on the

Pentium 3 (ignoring the smallest problem size) and ≈ 1.35 on the Pentium 4. For more details on the code generated by the two different approaches, see Section 10.7.3.

The direct cost of using the higher order features that remain after optimisation must be fairly small, as the Aldor version outperforms the Fortran version (with no higher order features) in all but one case. Based on this, it is reasonable to suggest that problems with higher order features are more likely to arise indirectly through extra load on the garbage collector (if the programmer is not managing storage) or lost opportunities to do optimisation across components. This argument can be extended to Aldor programs in general.

The splitting of the algorithm into three recurrences does not introduce any more barriers than implementing the algorithm using BLAS style routines, so, if barriers between components are already enforced by the use of opaque libraries (see below), Aldor is a prime candidate for writing the glue code. However, it should be noted that these experiments tend to play down the overheads of the language (such as the run-time costs of domains and very large code sizes) as the active portions of the programs are small, and they are run for a long time to minimise noise in the profiles. The overhead is revealed when contrasting the results for Aldor and Fortran for the smallest problem size.

10.6.2 Use of high-performance libraries

These control experiments take the original Aldor program for the general update step of the two sided Lanczos algorithm and replace the manipulation of vectors with high-performance BLAS routines (which is equivalent to introducing a certain amount of fusion by hand).

Three different versions of the operator are used with this harness – the original written in Aldor, a version written in C and a version written in assembler. These three experiments are intended to provide a comparison between the "bottom-up" approach of starting with an algorithm in some language and substituting compute intensive parts of the program with small sections of extensively tuned code against the less intensive but more global approach of fusion/contraction. Although the original Aldor program is used as the harness, this is largely irrelevant as almost no time is spent

outside of the work intensive routines (i.e. the operator and BLAS routines), so the version with the operator written in C can be can be considered equivalent to starting from an implementation purely in C and adding binary BLAS routines.

The results for all three versions are given in Table 10.11, along with a comparison to the best search result. A breakdown of where the time is spent in each of three versions is given in Table 10.12.

10.6.2.1 Aldor + BLAS

This control is an implementation of the two-sided Lanczos process using the Wilson-Dirac operator, where vector operations have been replaced by high performance BLAS binaries.

10.6.2.2 C + BLAS Wilson-Dirac stencil

This program is the same as that above, but the Wilson-Dirac operator has been replaced by a version written in C [82].

10.6.2.3 Assembly code + BLAS Wilson-Dirac stencil

This is similar to the C based version described above, but represents a further step along the path of local specialisation by using an application/machine-specific hand written assembly routine for the delta term [59]. As only the delta term is supplied, calculating the full stencil term needs a subsequent axpy, and this is performed using a BLAS routine. This is reflected in the breakdown, which shows significantly more time spent in the BLAS routines than for the other two versions.

The results show that, for the Wilson problem, using optimising transformations and compiling from Aldor source is competitive with the approach of starting from Aldor or C and substituting generic high-performance routines where possible, giving a small speed up of around $3-7\%$. It does less well against the combination with the assembly based operator. These results are discussed in detail in Section 10.7.4.

Table 10.11: Times for alternate methodologies vs search, Pentium 4

size	best search	BLAS augmented version		
		Aldor	C	asm
6	6.55	6.77	6.94	4.84
8	22.3	23.74	24.51	17.82
10	55.4	59.33	61.54	45.63
12	117.2	125.50	130.00	96.06
14	226.7	241.84	251.61	179.52

Table 10.12: Breakdown for alternate methodologies, Pentium 4

size	stencil or delta term			BLAS			rest of code		
	Aldor	C	asm	Aldor	C	asm	Aldor	C	asm
6	4.84	5.56	3.18	1.85	1.33	1.59	0.08	0.05	0.07
8	17.69	18.87	10.48	5.99	5.50	7.25	0.06	0.14	0.09
10	44.47	46.73	26.7	14.67	14.60	18.71	0.19	0.21	0.22
12	94.27	99.34	55.73	30.85	30.25	39.94	0.38	0.41	0.39
14	183.32	192.37	104.72	57.72	58.41	74.01	0.8	0.83	0.79

10.7 Discussion of Results

This section presents some broader discussion of the result presented previously.

10.7.1 Interloop locality in Aldor

Interloop locality is likely to be important for the performance of Aldor programs on cache based architectures as a result of the modular structure that the language encourages. This can be extended to general cross-component optimisations, and other goal functions for which temporal locality is important.

10.7.2 Search

As well as giving portability and dealing with variation within (c, p) sets, search is a useful addition to loop fusion and array contraction due to the non-triviality of the problem. By this it is meant that maximum array contraction does not always give the best result (and neither does minimum size). An example of this is the Wilson problem. For these experiments the extra benefit of searching the larger fusion partitions is not great, $\approx 5\%$, but different LDGs may provide examples where there is larger benefit to be had when searching away from the sets with maximum contraction.

It is not obvious why this pattern occurs for the Wilson problem. The best fusion partition always places each stencil term on its own in a separate loop, and again it is not immediately obvious why this gives the best performance. An important distinguishing factor for the Wilson problem is that the code required to execute the stencil is much larger than the equivalent for the 3D and 4D problems, as well as requiring many more operations. Consequently, instruction cache behaviour and the use of loop rerolling (see Section 9.2.2) may all be important, alongside secondary factors such as data prefetch hardware etc.

10.7.2.1 The greedy algorithm

The comparison of searching for good fusion partitions vs. a simple greedy algorithm probably flatters the greedy method somewhat. This is because the LDG does not have

any bad local minima that the greedy method might get stuck in, and the variability of the (c, p) set that it reaches is artificially low. A general investigation of potential problems with the greedy approach requires some notion of an "average" LDG though – see the future work in Chapter 11.

10.7.2.2 Best fusion partitions

A handful of fusion partitions recurred as the best choice in several different settings, mostly for different operators on the same machine. This suggests that there may be further features of a fusion partition other than just its size and the amount of contraction that consistently indicate that it is likely to be a good choice for a given machine.

10.7.3 Code generation

This section highlights some points about code generation for the target machines.

10.7.3.1 The architectures

The Pentium 4 has a small architectural register file, a high clock rate and corresponding long relative access latency to main memory, and short vector instructions for floatingpoint arithmetic (SSE2). The first two points suggest that there are likely to be significant latency hiding benefits from issuing software prefetch instructions (although some of this may be covered by the limited automatic hardware prefetching), and the third suggests that proper utilisation of short vector instructions will give a significant boost to performance on floatingpoint intensive programs. The Pentium 3 is similar, but has a lower clock rate and less latency problems, no hardware prefetching, and its short vector instructions (SSE1) are only single precision and therefore not suited to the programs in this thesis.

For a given section of code, the potential return from specialised code generation is likely to be large for the Pentium 4. The exploitable potential for the Pentium 3 is likely to be less given that only software prefetching is applicable and the latency to main memory is less severe. However, the lack of hardware prefetching may make the latency problem more important.

10.7.3.2 Code generated by `icc` and `ifc`

Examining the output of `icc` resulting from Aldor generated C shows that vectorisation is not done for the Pentium 4 (it is not possible for the Pentium 3). The compiler produces "scalar" SSE2 instructions, which, strictly speaking, are from the short vector instruction set, but they only operate on one rather than two operands at a time. This apparently redundant strategy is understandable in light of the well-documented poor performance of the Pentium 4 on `x87` instructions. The lack of vectorisation is hardly surprising given the output of the Aldor compiler and the potential difficulties of recovering alias information from it. However, vectorisation was not performed by `ifc` for the Fortran programs either, and this was unchanged by adding `-fno-alias` (i.e. assume no aliasing in the program) which is more surprising. Software prefetch instructions were rarely issued in any of the generated assembly code, and no loop unrolling appeared to be done by either compiler.

The amount of inlining performed on the Fortran codes by `ifc` was small, being limited to all but the smallest routines. In addition, it did not perform any high level loop restructuring (including fusion/distribution) even with `-O3` enabled, possibly as a result of the lack of inlining. No inlining of any of the significant routines was formed by `icc`, probably due to the large amount of inlining done by the Aldor compiler.

The failure of `icc` to produce vectorised and adequately prefetched code from the generated C affects the discussion on local code tuning (below) and prompted several of the suggestions for future work (see Chapter 11).

10.7.4 Local tuning and code generation

This section presents a detailed discussion of the results of comparing iterative collective loop fusion against the substitution of sections of code with BLAS routines or assembly (for the operator) on the Pentium 4.

10.7.4.1 Generated code versus BLAS routines

The machine code for the level 1 BLAS binaries is reasonably assumed to be substantially better than that which a standard compiler would produce from equivalent source

code. This is due to the authors having information that the compiler does not, as well as more sophisticated strategies to get better performance, and the high relative gain available from targeting certain features of the Pentium 4 when compared to a compiler that does not (see Section 10.7.3). However, the tuning effort is limited to very small components in isolation and misses opportunities for optimisation across the boundaries between them. Compiling from Aldor (via `icc`) using fusion/contraction thus pits the exploitation of global knowledge with comparatively weak subsequent code generation against local tuning.

Given the form of the level 1 BLAS routines, and assuming they are perfectly prefetched and use the short vector instructions, their operation can be broken down into three stages. The first is filling the logical pipeline by issuing loads/prefetches and waiting for the first operands to appear, the second is when the pipeline is full and the arithmetic is being done in conjunction with loads/prefetches and stores, and the third is draining the pipeline when only arithmetic and stores are left to do. The floatingpoint operations are an unavoidable cost, so the avoidable costs that collective fusion/contraction cuts out probably result from under-utilisation of the floatingpoint unit. This occurs during the pipeline fill stage, and may occur during the middle stage if the floatingpoint unit is not the limiting factor on the bandwidth of the pipeline.

10.7.4.2 Interpreting the results

Examining the breakdown for Aldor + BLAS tells us that $\approx 24-27\%$ of the time is spent in the BLAS routines for this program. In the fused case, the best fusion partition places both stencil calculations on their own in separate loops, so although the profile does not give us the exact cost of the stencil term, it is reasonable to assume that it takes the same amount of time as for Aldor + BLAS. By subtracting the expected time spent in the stencil from the result for the best search, the improvement of fusion/contraction over tuned BLAS routines can be calculated as $\approx 25\%$.

There are several possible explanations for why the global approach is competitive with local tuning in this context. The start-up costs will be small given large enough vectors, so the real difficulty with local tuning is likely to be that the logical pipeline is

limited by the bandwidth of the memory subsystem[3]. By eliminating loads and stores (or if not eliminating them, ensuring that they hit in the cache which will have much higher bandwidth), this is precisely the difficulty that loop fusion/array contraction avoids under the assumption of global perfect prefetching (i.e. no latency costs). In practice, the code produced by fusion/contraction will suffer some latency costs itself as it is not perfectly prefetched, and the potential benefit is limited by failing to generate short vector instructions, which means that the floatingpoint unit will become a bottleneck much sooner than it ought to. Despite this, fusion/contraction compares favourably, and the elimination of these low-level problems may bring significant benefit and result in substantial further gains over local tuning.

The situation is somewhat different for the tuning of the operator. Although the assembly is unlikely to be as highly tuned as the BLAS routines, it does exploit register tiling, double precision short vector instructions, and some software prefetching. By calculating the cost of the stencil term as the cost of the delta term from the breakdown plus the extra time spent in the BLAS routines compared to the other two methods, the assembly based operator is ≈ 1.5 times faster than the Aldor version, and this is enough to give an overall speed up of $\approx 1.3 - 1.4$ over Aldor + BLAS and $\approx 1.2 - 1.3$ over the best search result. Removing prefetch instructions from the assembly kernel degrades its performance by $\approx 10\%$. Together, these facts suggest that, in contrast to the level 1 BLAS routines, the balance of the component between loads/stores and floatingpoint operations is probably such that the floatingpoint unit rather than the memory subsystem is the bottleneck. So, in this instance the real problem is the failure to generate short vector instructions from the Aldor generated C.

10.8 Summary

This chapter has shown that:

- Collective loop fusion and array contraction is an important technique for ex-

[3]There are also the potential problems of structural conflicts between issuing loads/prefetches/stores and floatingpoint instructions in the processor, and instruction cache spill, but given the architecture in question and the size of the routines, these are unlikely.

ploiting interloop locality in highly modular Aldor programs such as those developed in this thesis, giving speedups of up to 3.7.

- Iterative collective loop fusion does better than the other collective loop fusion strategies and brings in addition portability across hardware and problem type. It gives speedups of up to 1.36 over the greedy method, and 1.41 over max-flow min-cut.

- Starting from a modular program in a high level language such as Aldor and using iterative collective loop fusion gives better performance than an equivalent program written from scratch in Fortran, with speedups of up to 1.47. Hence, elegance of expression need not mean sacrificing performance.

- Our transformation strategy is better than starting from the original program or an equivalent version in C and substituting high-performance BLAS libraries for vector manipulation, giving an estimated speedup of around $24-27\%$ over the library routines themselves. Even in a situation where one single loop dominates execution time and the assembly version of that loop is ≈ 1.5 times faster than the Aldor version, our global technique is only a factor of $\approx 1.2-1.3$ worse in terms of overall performance. It is not obvious how to address the architecture specific low-level code generation issues with the Aldor version of the operator to improve its performance compared to the assembly version whilst maintaining the portability model of the current compiler (i.e. generating standard C code). However, if this problem could be resolved, our technique is likely to provide overall better performance and obviate the need for laborious machine specific hand tuning.

Chapter 11

Conclusion

This chapter collects together the conclusions dotted throughout the text into a meaningful whole, and adds some extra comment. After this there follows some suggestions for follow-up work.

11.1 Summary

In this thesis we have shown how to express the modularity inherent in a family of numerical algorithms (the Krylov space-based iterative solvers) by using the advanced abstraction mechanisms of Aldor (domains, categories etc) to build an algorithmic framework. This represents a significant improvement over the standard approach of collapsing structure into a recipe with a set of choices already made, by making the structure explicit, eliminating redundant replication of code, and allowing rapid assembly of different algorithms by combining pieces. This modularity is expressed at different levels, ranging from independence of the algorithms with respect to implementations of scalar, vector and operator domains, to the ability to mix and match methods of generating a projected operator and decomposing it into factors.

We have argued that the direct (and to a lesser extent indirect) costs of the use of higher order language features to structure the algorithms by joining separate recurrences is small. Conversely, we have shown that the abstraction with respect to the vector and operator domains has a large indirect cost on a cache based architecture

arising from the lack of temporal locality in the resulting programs, even when the direct cost (coming from separation into functions/domains and use of simple higher order features such as generators etc) is removed. We have characterised this problem as an extreme form of a similar problem (i.e. interloop locality) affecting numerical programs written in standard third-generation languages[1]. We have argued that the severity of the problem is what makes interloop rather than intraloop locality (where it exists) a priority for investigation in the context of the language and the solvers, and suggested that *cross-component* optimisations, of which this problem is an example, are likely to be important in general to languages such as Aldor.

We have adopted the loop dependence graph (LDG) formalism from the literature and shown how to enumerate fusion partitions of a given size, and how this can be used to collect fusion partitions (of that size) with a given amount of contraction – i.e. how to generate (*contraction amount*, *partition size*) sets ((c, p) sets). This is used as the basis for empirical selection of fusion partitions based on actual performance. We have given heuristics to suggest which (c, p) sets to prefer (more contraction, followed by smaller size), a heuristic method for systematically finding the elements of the preferred (c, p) sets in a large space (start from small fusion partitions and work upward), and suggested a heuristic approach to choose points to test from a (c, p) set that is too large to test exhaustively (first come first served). The overall technique, parameterised by how much of the space to enumerate, and how many points to empirically test, is called *iterative collective loop fusion*.

We have demonstrated empirically that cross-component optimisations achieved using collective loop fusion/contraction can provide a speedup of up to 3.7 in this context. We have also shown that it is difficult to pick the right transformations based purely on static information, given that performance varies within a (c, p) set (up to $\approx$ 130%) and across different problems and machines, by testing complete sets of fusion partitions on an LDG extracted by hand from the iterative solver framework. This exposed the crudeness of using (c, p) set (or just contraction) as a metric to indicate performance. At the same time this shows how iterative collective loop fusion could be used to find a good fusion partition in the case where several (c, p) sets chosen by

[1] They suffer the problem to a lesser extent as a programmer will usually implement some degree of fusion already by hand.

the programmer are exhaustively enumerable and testable. We have also shown that in some cases large amounts of search are necessary to find a good fusion partition when the heuristics are less accurate (such as for the Wilson problem).

We have compared iterative collective loop fusion to other methods of collective loop fusion (greedy and max-flow min-cut) with an emphasis on the weakness that arises from relying on simple metrics. This leads to a speedup of up to 1.36 and 1.41 respectively. We have also given some measure of how any static method that relies solely on (c, p) as a metric will be limited if the machine or program changes. We have given some comparison against other methodologies such as using Fortran or hand-tuned assembly. Using Aldor with iterative collective loop fusion does better on the whole, except in comparison against the assembly Wilson-Dirac operator – we have suggested that this is most likely due to SIMD vectorisation issues, which it is not easy to attack directly when generating standard C code.

11.2 Future directions

This work could easily be extended in a number of largely orthogonal directions. We give a brief outline of some possibilities below.

11.2.1 Framework design

For the category framework in Chapter 5, there are numerous fairly simple extensions that can be investigated. These include (but certainly are not limited to):

1. Look-ahead to deal with breakdown in the two-sided Lanczos process.
2. Using k-step restarting for the long recurrence methods.
3. Partial pivoting to deal with breakdown in an LU decomposition.
4. Preconditioning.
5. Methods based on the normal equations – *CGNR*, *CGNE* etc.
6. Nested Krylov space algorithms.

7. Various different types of halting condition.

8. Incorporating eigensolver algorithms.

The first two of these deal with the interface between the Krylov space and the projected system, where the projected matrix is no longer just tridiagonal but has varying upper band width from step to step. This may also have an impact on the interface to the search vectors, in a similar manner to the third item. The fourth item ought to be easy to add with small wrappers, as already outlined in Section 3.6.4. The next three items may require some broader adaptation of the framework with some mechanism to pass more information around, outside the interface provided by the current pieces. The design could also be cleaned up by removing the abuse of the valuation domain, and pinpointing the root of the type system problems discussed in Section 5.3.1.1. The last item would require more work, but is certainly a natural extension of the modular approach.

11.2.1.1 Alternative factorisations

A possible use of the framework would be to investigate alternative factorisations of the projected systems. There are good reasons for the standard couplings of orthogonality condition and matrix decomposition. The *LU* decomposition is cheap to compute, and will only break down when the Galerkin condition cannot be satisfied for a given step (see Section 3.4.3). The *QR* decomposition directly gives a solution to the projected least squares problem from the minimum residual condition without having to form the normal equations, and also provides the recurrence residual. Using the *QR* decomposition for the minimum error condition reduces the projected system to something more manageable, and allows the use of search vectors in the short recurrence version. Nevertheless, it is possible to use different factorisations for any given condition.

Separating the algorithm into components at the program level may enable investigation of techniques that make some of the assumptions behind the standard parings redundant. For instance, the added numerical stability of a *QR* factorisations may be an advantage for a short recurrence Galerkin algorithm that has the ability to skip steps where the orthogonality condition cannot be satisfied. This is similar to the idea of

allowing partial pivoting to cope with one type of breakdown in an *LU* factorisation.

11.2.2 Solver domains

For the solver domain implementations in Chapter 5, the simplest extension would be to add more Krylov space generating algorithms such as those in [73] and [41], or possibly algorithms based on Householder transformations or selective re-orthogonalisation. In addition, there remains some work to be done to flesh out the full implementation of the minimum error orthogonality condition (and associated search vector recurrence), and the long recurrence and incomplete orthogonalisation methods.

11.2.3 Operators

The category structure for the operators presented in Chapter 6 is not as developed as it could be. The obvious future direction is to adapt the initial work to properly capture the structure of the Wilson-Dirac operator (for example using [30] as a starting point) and other operators and linear systems of interest to mathematical physics. This in turn would feed into the design of the linear solvers package. Some of the interesting issues are discussed in [43, 35], including red-black preconditioning, γ_5-Hermiticity, choice of algorithm, and the interaction with molecular dynamics.

11.2.4 Iterative collective loop fusion

There are several ways in which the approach outlined in Chapter 8 could be extended and or adapted:

- Experiment with different search heuristics, or attempt to refine the heuristics to reduce the amount of empirical evaluation that is necessary. The repeated occurrence of some fusion partitions as the best performers in different settings suggests that within a (c, p) pair set there may be some characteristics of a fusion partition that make it more likely to do well regardless of operator/machine/problem size. If these characteristics could be discovered, they could be used to further narrow down the amount of search necessary to find good results. Some

possibilities in this regard are hand analysis or automatic feature extraction tools from artificial intelligence.

- As well as cutting branches of the search tree based on empty partitions, it ought to be possible to significantly speed up search by having a cut-off based on the number of contracted arrays – that is, once the number of contracted arrays is guaranteed to exceed the required amount, further search along that branch of the space can be abandoned.

- Investigate methods of dealing with less well behaved loops (e.g. non-conformable). This could include using standard transformations such as flattening/peeling/shifting etc. to preprocess the code, or using a more general abstraction such as affine transformations. Another improvement in this vein would be to develop techniques that can cope with branching within the program section. This extension to more complex control flow could be approached by generating multiple transformed versions of the original code and selecting execution at run-time based on the evaluation of the branch conditions.

- Develop a more rigorous way of dealing with enumerated slices of the search space that are too big to test exhaustively by empirical experiment.

11.2.5 Empirical results

The following are some suggestions to broaden the empirical results given in Chapter 10.

11.2.5.1 More precise results

A closer analysis of the performance of different fusion partitions by detailed simulation or the collection of data from hardware event counters would give a better idea of how search is balancing trade-offs and making its gains. This would also give a better idea of how exactly local tuning gets its advantage (in the case of the assembly operator) or fails to exploit the full performance of the machine (with the BLAS routines).

Information derived from such an analysis would also feed into the search for more precise heuristics (see below).

11.2.5.2 More architectures

Provided that the Aldor compiler and run-time system could be ported, transferring the experiments to other architectures ought to be straightforward. This would provide interesting further results in terms of the variability across machines of what constitutes a good fusion partition, and the ability of search to cope with this. Other x86 compatible architectures could also be included.

A comparison against hand tuned code (such as the BLAS routines) on a machine that does not require the use of short vector instructions to achieve reasonable floating-point performance would also prove interesting. In some sense it is the counterpart of studying how well the optimisations do when vectorisation is added for architectures that require it (see below).

11.2.5.3 More LDGs

It would be nice to extend this work with further experiments on different LDGs. However, for this to be relevant to Aldor (or languages like it), the benchmarks would have to be taken from programs written in a natural style rather than standard benchmarks transplanted from C or Fortran. This puts such an extension well outside the scope of an individual project, as multiple different benchmarks would be necessary, and is likely to require the effort of a community of developers. An alternative would be to do some experiments on randomly generated synthetic LDGs.

Having more LDGs would also allow a better evaluation of the technique against the greedy approach (and to a lesser extent max-cut min-flow).

11.2.6 Other optimisations

Searching for a good fusion partition targets locality between loops, and this approach suffers from Amdahl's law when one loop takes $\approx 70\%$ of the total execution time, as in the case of the Wilson problem. However, optimisation of the delta term (or general

code for that matter) with low-level code generation techniques, such as vectorisation, was considered a lower priority in the context of this thesis due to the following reasons:

- The issue is not really specific to this type of language in any way, but rather to a particular architecture. The same problems arise when compiling from e.g. C.
- This would be hard if not impossible to do in a portable way for a compiler that achieves its portability by generating standard C.

Nonetheless, the interaction of iterative collective loop fusion with other optimisations, especially vectorisation if it is necessary (and possible) and the stencil tiling outlined in Appendix B, would be interesting indeed[2].

Improving the performance of loops resulting from iterative collective loop fusion would also be interesting. The additional optimisations would include at least software pipelining, loop unrolling and software prefetching, although there may be interaction with others as well (such as tiling, padding etc). A related subject is the interaction of inlining with collective loop fusion, given that it is likely to be used as a preprocessing step for LDG recovery. The most natural way of incorporating other optimisations would be to apply the methodology of iterative optimisation to give portability etc (see the literature outlined in Section 8.3.2).

11.2.7 Other languages

Iterative collective loop fusion as a technique in itself could be ported for use with other languages or in other settings. For use with more traditional languages such as Fortran or C, the standard style of codes will probably mean that preprocessing techniques such as scalar expansion and loop distribution will be necessary to prevent artificial dependencies in the LDG, and that the total achievable benefit will be less (as some fusion/contraction has already been done by hand), as mentioned in Section 7.3.4. Porting to languages that have similar modularity issues to Aldor, such languages with array statements (e.g. Fortran 90), or object oriented languages (e.g. C++), may avoid

[2]Especially in the wider context of QCD simulations, where techniques such as red-black preconditioning put even greater emphasis on the efficiency of the stencil routine.

most of these problems. The main advantage of porting would be instant access to a large number of benchmarks and machines to test the technique on.

Another application for the technique is in the setting of automatic loop based parallelisation, which occurs frequently in the literature. Loop fusion is used here primarily to reduce the overhead of barrier synchronisation between loops. In this instance, contraction would only be applied to the subsection of each array that belongs to each processor, rather than reduction to a single scalar.

Appendix A

Conjugate Gradients and the Lanczos Type Product Methods

This appendix gives the relationship of the framework used in this thesis to some other members of the family of Krylov subspace based iterative solvers – namely conjugate gradients, biconjugate gradients, and the Lanczos type product methods.

A.1 Conjugate gradients

Arguably the most popular iterative method for Hermitian operators is *CG* (conjugate gradients), and one of the most popular short recurrence methods for non-Hermitian operators is its two-sided cousin, *BiCG*. The *CG* (*BiCG*) algorithm is very closely related to the algorithm given in Chapter 3 based on the Hermitian (two-sided) Lanczos process, the Galerkin condition and the *LU* decomposition, but the Krylov basis is generated by coupled two-term recurrences rather than three-term recurrences.

Because of this, *CG* and *BiCG* don't easily fit into the framework that is developed in this thesis. What were previously separate pieces, that is the generation of the basis vectors, the matrix decomposition and the updating of the search vectors, are now inextricably coupled. This is why it is forsaken in favour of a less traditional approach.

It is also possible to generate the Krylov basis using linked two-term recurrences in the modular version, but this comes at the cost of extra vector storage and manipula-

tions. In addition, the linked two-term recurrences implicitly make use of the Galerkin condition and consequently will break down if it cannot be satisfied at every step for an indefinite operator. This becomes an issue if a modular algorithm is developed that avoids this problem with the Galerkin condition by modifying the components from the projected system onward, as the potential for breakdowns is re-introduced in the Krylov space generating component.

A.2 The Lanczos type product methods

The original Lanczos type product method was *CGS* introduced by Sonneveld [81], and most other product methods are variations upon this theme. The algorithm is derived from *BiCG* by considering the polynomials in A generated by the recurrences and algebraically manipulating (squaring) them. It fundamentally relies on the coupled two-term recurrences of *BiCG*.

In *CGS*, the Hermitian transpose of the operator is not used, and this is handy if the operator is both non-Hermitian and its Hermitian transpose is expensive or impossible to generate. Although we still generate the same scalars, we no longer explicitly generate the residual vectors or the search vectors from *BiCG*, and so some other method of recovering the candidate solution must be found. The approach in *CGS* is to take the vectors generated by the squared recurrence for the *BiCG* residual vectors, and impose them as the residual vectors of the algorithm by updating the candidate solution appropriately. Note that this doesn't require the inverse of the operator because of the way in which the *CGS* residuals are generated, which in turn relies on its derivation from coupled two-term recurrences and the Galerkin condition.

Imposing the result of the squared *BiCG* residual recurrence as the residual of the *CGS* algorithm means that it no longer obeys any simple orthogonality condition, and is not directly related to any decomposition of the projected matrix from the Arnoldi relations. The residual is now taken from a Krylov subspace that is twice the size of the original *BiCG* Krylov subspace, and it is reasonable to assume that the approximate solution generated in this manner might be a better one – put another way, the work done in applying the operator a second time is not "wasted" in that it goes toward

updating the residual, in contrast to the application of the transpose in *BiCG*. However, the scalar factors for the *CGS* Krylov space are still directly related to those taken from *BiCG*.

For two-sided methods, there is no particular reason why the polynomial for the dual Krylov space has to be the same as that of the original one, as long as the scalars that the algorithm relies on can still be derived. For *BiCG*, this is irrelevant, as the only function of the dual space is to produce the scalars, and so changing the polynomial would make no material difference. For a product method though, the dual polynomial is used to determine the residual, and so changing it changes the approximate solution produced by the algorithm. This is a degree of freedom that can be used to improve the approximation generated, and the many follow-ups to Sonneveld's work consist of various methods of defining the dual polynomial, such that the necessary scalars can still be produced and the residual vector is hopefully better in some sense. In practice, an appropriate product method will usually provide a significantly better approximation for the same number of operator applications, and this is important as the cost of the the operator application is almost always the single largest cost in the algorithm.

Because of the way in which they are derived, the Lanczos product type methods do not fit into the framework used in this thesis. It is possible to take the two-sided Lanczos process and square it, but in order to be able to calculate the search vectors and the approximate solution in the same way as before, it is also necessary to calculate the original Krylov basis, and this means using an extra application of the operator per iteration step [21]. Hence, there is little point in having this extra expense unless the Hermitian transpose of the operator is not available, and there is some good reason for not wishing to use one of the standard product methods. It would be interesting to develop a framework for the product methods where the pieces constituted recurrences for certain polynomials, but that will certainly have to be left to future work.

A.3 Functional parallelism and product methods

As mentioned in Section 3.3.3.1, the vector sequences in the two-sided Lanczos process evolve in parallel, whereas the applications in a product method are sequentialised

due to the presence of inner product operations in between them. While the product methods usually take many fewer steps (operator applications) to converge than the two-sided methods, they may not converge twice as fast. Indeed, the choice of iterative method to use for a non-Hermitian problem is usually considered to be problem dependent and very much an open question [73, 41].

This induces an interesting trade-off. On a large enough parallel machine, where the vector sequences from the two-sided Lanczos process could be computed in parallel, one step of a two-sided process ought to take roughly half the time of one step of a product method. Although a two-sided method is likely to take more steps to converge than a product method, it may converge quicker in terms of wall clock time. It ought to be noted however, that the margin of difference between a two sided method and a product method is likely to be small (i.e. a product method may take almost half the number of iterations of a two sided method) [34], and thus the benefits of using a parallel two-sided method may not be great compared to the amount of extra computing resources required. Nonetheless, exploitable parallelism should be noted under the assumption that available compute resources tend to get cheaper very quickly, and it is easier to exploit them than to develop new algorithms.

The above reasoning translates to sequential machines with a cache hierarchy due to temporal locality. The lack of synchronisation points between the application of operators in two-sided methods means that they can be overlapped, allowing the reuse of data for an operator with a concrete representation that is common to both the original and adjoint representations. Where an operator has no concrete representation, there is still an advantage in terms of greater flexibility to re-order computation.

Appendix B

Re-use in the Operator Application

This appendix briefly summarises "stencil tiling" in the context of the operators considered in Chapter 6, and a cache based architecture. Re-use in a 3D stencil operation (for a pure stencil) is covered in [71] (along with techniques for choosing tiles sizes for direct mapped caches based on simple models). The idea can be illustrated by considering vectors that represent some regular three-dimensional cube composed of two dimensional slices, where the loop that calculates the stencil is constructed to traverse the sites in the three-dimensional cube slice by slice. Calculating one slice of the result vector requires at most three neighbouring slices of the source vector.

The following simplifications are used for the discussion:

- The cache allocates space for writes – this assumption is conservative, as a write-allocate policy uses more cache than no-write-allocate.
- There are no associativity conflicts – this is not a conservative assumption, but only requires 4-way set associativity to guarantee the premise (in the absence of self-conflicts within a slice due to awkward grid dimensions). It may be satisfied for 2-way set associativity depending on the degree of overlap between address ranges, and is least likely for direct mapped caches.
- The cache eviction policy is FIFO – this assumption is not as conservative as random replacement, but it makes the analysis easier.

- The points in a slice can be traversed in an arbitrary order – this is conservative, and requires that all of the slices must fit in the cache simultaneously.
- Possible effects of cache line size are ignored, i.e. no unwanted data is ever loaded into the cache. This is not fully conservative, but it is reasonable to expect the addresses within a given slice to be consecutive, which only leaves very small effects at its beginning and end. Cache line effects may also make the reloading of partial data from a site more expensive than necessary if the data has been evicted from the cache. However, this is only relevant for missed opportunities for re-use of $SU(3)$ matrices from the Wilson operator, as in other cases all data is used from a site, and would have to be loaded anyway (see later).
- Boundary conditions are ignored.

All possible re-use is captured provided the four entire slices in use can fit simultaneously into the cache. If not, locality can be improved by dividing the cube along the axis perpendicular to the slices and changing the problem into computing the application of the stencil for successive subsections of the cube, the dimensions of which are chosen to ensure that four slices thereof will fit into cache.

B.1 3D Pure Stencil with One Level of Cache

In the general case where sites in the vector have size s bytes and the machine has a single level of cache of size c bytes, given a cube of size n, tiling improves locality when $4n^2s > c$. To keep the discussion simple, we require both vectors necessary to compute the stencil to fit into the next level of memory (of size m bytes) after cache, otherwise misses at that level of the memory hierarchy may dominate the overall performance characteristics of the program. This implies $2n^3s < m$, and so:

$$\begin{aligned} nc/2 &< 2n^3s \\ nc &< 2m \end{aligned}$$

showing that the memory must be $O(n)$ larger than the cache. As the size of the cache c grows, n grows like $O(\sqrt{c})$ and to satisfy the constraint m must grow like $O(c\sqrt{c})$. One possible way of extending this to systems with multi-level memory hierarchies is to consider each pair of adjacent levels separately using the same approach.

The argument given above implies why re-use along only one axis of the space is considered important, as n would have to be *much* larger for a single line in the space (rather than a slice) to fill the cache and make tiling in two dimensions necessary. Additionally, the memory would have to be of size $O(n^2)$ for the vectors to fit, and taken together these two factors suggest that the ratio of cache size to memory would rapidly become unrealistic.

The cost of not tiling when $4n^2s > c$ is that (ignoring the extremes of the cube) all three slices of the source vector will have to be loaded to calculate the slice of the result due to the cache eviction policy. Each slice of the source, and therefore the whole vector itself, is loaded three times rather than one, thus missing a factor of two re-use.

B.2 4D and Concrete Operators

The idea outlined above can trivially be extended to four dimensional stencils, where three neighbouring cubes (3-cubes) of the total space (4-cube) must fit simultaneously into cache, so tiling has an effect when $4n^3s > c$. Again the memory must be $O(n)$ larger than the cache, but for a given size of cache n will be smaller and thus the amount of memory needed to hold the entire problem at the level after the cache will be smaller. A related point is that tiling will have an effect at much lower values of n.

The discussion can be extended to the Wilson problem by also considering the space requirements and re-use of the operator on a per site basis. The gauge field associates four $SU(3)$ matrices with each site in the space. However, the way the gauge field is touched differs from the way that the vector is accessed, where the calculation for a given 3-cube of the vector touches all the information from the sites in the current and both neighbouring 3-cubes. In contrast, for a given axis of the 4-cube one $SU(3)$ matrix associated with the current site and one matrix associated with the site one

step backward along that axis is used. To calculate a 3-cube of the target vector thus requires all the $SU(3)$ matrices associated with the sites in the current cube plus one matrix for each site in the previous 3-cube (and none of the matrices from the sites in the next 3-cube). That is, each $SU(3)$ matrix is only used twice.

Given the assumed cache policy, two full 3-cubes of the gauge field must fit into memory in addition to the vector cubes to avoid any reloading, i.e. $4n^3 s_v + 2n^3 s_{op} < c$, where s_v is the per site size (in bytes) of vector information and s_{op} is the per site size of the operator (i.e. four times the size of an $SU(3)$ matrix). Failure to tile when n becomes large enough requires the vector to be reloaded as before, but only requires the gauge field to be loaded $1\frac{1}{4}$ times rather than once, as only the re-use along one direction of one axis of the 4-cube (i.e. one $SU(3)$ matrix per site) is lost.

B.3 Operator Tiling in Practice

To give some idea of what this means in practice, consider the simple 3D operator problem with complex double float elements at each site (s is 16 bytes), and a machine with two megabytes of cache (c is 2×2^{20} bytes). Tiling becomes important when:

$$\begin{aligned} 4n^2 \times 16 &> 2 \times 2^{20} \\ n &> 182\, sites\, (approx) \end{aligned}$$

which is already a large factor, although not unreasonably big (requiring the memory to be at least $\frac{1}{2}nc = 182\text{MB}$ large). However, the super linear growth in memory requirement suggests that as cache sizes grow the effects of misses at the next level of memory hierarchy may come into play.

For 4D problems, the threshold for n will be lower due to the cubic term. Because of the gauge field and the fact that each site in the vector is larger, the threshold of n for the Wilson problem will be much lower.

Appendix C

Categories

This appendix contains verbatim Aldor code for the category hierarchy discussed in Chapter 5. One macro definition[1] is used, namely `SI` to stand for `SingleInteger` from `axllib`.

C.1 Linear Algebra Categories

The convention for naming parameters throughout this section is straightforward. The valuation and ground field parameters are named as such, and capital letters are used to stand for linear spaces (i.e. domains of vectors).

```
define OrderedField: Category == Join(Field, OrderedRing);

define FieldWithValuation(Valuation: OrderedField) : Category == Field with {

        valuation: % -> Valuation;
        coerce: Valuation -> %;
}


define Module(R : Ring) : Category == AbelianGroup with {

        * : (R, %) -> %;          ++ Left multiplication by a scalar
        * : (%, R) -> %;          ++ Right multiplication by a scalar

        default
          (v : %) * (a : R) : % == a * v
}
```

[1]For a description of Aldor macros, see [93]

```
define LinearSpace(GroundField : Field) : Category == Module(GroundField) with {

        / : (%, GroundField) -> %;        ++ Division by a scalar

        default
          (v : %) / (a : GroundField) : % == (1/a) * v
}


define LinearAlgebra(GroundField : Field) : Category
== LinearSpace(GroundField) with Monoid;


define LinearSpaceWithDual(GroundField : Field,
                           DualSpace : LinearSpace(GroundField)) : Category
== LinearSpace(GroundField) with {

        * : (DualSpace, %) -> GroundField;      ++ apply a linear functional
        apply : (DualSpace, %) -> GroundField;

        default
          apply(a : DualSpace, b : %) : GroundField == a * b
}


define NormedLinearSpace(Valuation : OrderedField,
                         GroundField : FieldWithValuation(Valuation)) : Category
== Join(LinearSpace(Valuation), LinearSpace(GroundField)) with {

        norm : % -> Valuation;
        norm : % -> GroundField;
}


define NormedLinearSpaceWithDual(Valuation : OrderedField,
                                 GroundField : FieldWithValuation(Valuation),
                                 DualSpace : NormedLinearSpace(Valuation,
                                                               GroundField)) : Category
== Join(NormedLinearSpace(Valuation, GroundField),
        LinearSpaceWithDual(GroundField, DualSpace));


define InnerProductSpace(GroundField : Field) : Category
== LinearSpaceWithDual(GroundField, %) with;
```

```
define NormedInnerProductSpace(Valuation : OrderedField,
                               GroundField : FieldWithValuation(Valuation)) : Category
== Join(NormedLinearSpaceWithDual(Valuation, GroundField, %),
        InnerProductSpace(GroundField)) with {

        normsq : % -> Valuation;
        normsq : % -> GroundField;

        default {
          import from GroundField;
          normsq(a : %) : GroundField == a * a;
          normsq(a : %) : Valuation == valuation(normsq a);
        }
)


define GroupAction(GroundField : Field,
                   V : LinearSpace(GroundField)) : Category
== Group with {

        apply : (%, V) -> V;
        * : (%, V) -> V;
        \ : (%, V) -> V;

        default
          (A : %) \ (v : V) : V == inv(A) * v;
)


define LinearMapping(GroundField : Field,
                     V : LinearSpace(GroundField),
                     W : LinearSpace(GroundField)) : Category
== with {

        * : (GroundField, %) -> %;        ++ multiplication by a scalar
        * : (%, GroundField) -> %;
        / : (%, GroundField) -> %;        ++ division by a scalar

        * : (%, W) -> V;                  ++ action on a vector
        apply : (%, W) -> V;
        explicitMapping : % -> (W -> V);  ++ create a general function
                                          ++ ('forget' some type info)

        default {
          (A : %) * (a : GroundField) : % == a * A
          (A : %) / (a : GroundField) : % == (1/a) * A
          apply(A : %, w : W) : V == A * w
```

```
        explicitMapping(a : %) : W -> V == (u : W) : V +-> a u;
    }
}


define LinearOperator(GroundField : Field,
                      V : LinearSpace(GroundField)) : Category
== LinearMapping(GroundField, V, V) with {

    * : (%, %) -> (V -> V);        ++ allow composition of operators by 'forgetting'
                                   ++ type info - doesn't require the category to
                                   ++ be a proper monoid

    default
        (a : %) * (b : %) : (V -> V) ==
                (v : V) : V +-> explicitMapping(a) explicitMapping(b) v;
}


define LinearOperatorWithDual(GroundField : Field,
                              V : LinearSpace(GroundField),
                              W : LinearSpaceWithDual(GroundField, V)) : Category
== Join(LinearOperator(GroundField, V), LinearOperator(GroundField, W)) with {

    * : (%, V) -> (W -> GroundField);  ++ allow the action on the dualspace to
                                       ++ be reduced to an explicit (lazy) function

    default
        (A : %) * (v : V) : (W -> GroundField) == (w : W) : GroundField +-> { v(A w); }
}


define LinearOperatorOnInnerProductSpace(GroundField : Field,
                                         V : InnerProductSpace(GroundField)) : Category
== LinearOperatorWithDual(GroundField, V,
                          V pretend LinearSpaceWithDual(GroundField, V)) with {

    adjoint : % -> %;
    * : (V, %) -> V;                            ++ multiply by adjoint
    bilinearForm : % -> (V, V) -> GroundField;  ++ define a bilinear form by
                                                ++ currying over the operator

    default {
        (v : V) * (A : %) : V == adjoint(A) * v;
        bilinearForm(A : %)(v : V, w : V) : GroundField == w (A v);
    }
}
```

```
define HermitianLinearOperator(Valuation : OrderedField,
                               GroundField : FieldWithValuation(Valuation),
                               V : NormedInnerProductSpace(Valuation,
                                                           GroundField)) : Category
== LinearOperatorOnInnerProductSpace(GroundField, V) with {

        quadraticForm : % -> V -> Valuation;  ++ define a norm-like quadratic form
                                              ++ by currying over the operator

        default{
          quadraticForm(A : %)(v : V) : Valuation == {
                  import from GroundField;
                  valuation(bilinearForm(A)(v, v));
          }
          adjoint(A : %) : % == A;        ++ operator is self adjoint
          (v : V) * (A : %) : V == A v;
        }
}


define PositiveDefiniteHermitianLinearOperator(Valuation : OrderedField,
                GroundField : FieldWithValuation(Valuation),
                V : NormedInnerProductSpace(Valuation, GroundField)) : Category
== HermitianLinearOperator(Valuation, GroundField, V) with {

        innerproduct : % -> (V, V) -> GroundField;  ++ HPD matrix defines a proper
                                                    ++ inner product
        norm : % -> V -> Valuation;                 ++ which in turn defines a norm

        default {
                innerproduct(A : %)(v : V, w : V) : GroundField ==
                   bilinearForm(A)(v, w);
                norm(A : %)(v : V) : Valuation == quadraticForm(A)(v);
        }
}


define OperatorAlgebra(GroundField : Field,
                       V : LinearSpace(GroundField)) : Category
== Join(LinearAlgebra(GroundField),
        LinearOperator(GroundField, V)) with;


define IndexedVector(GroundField : Field) : Category
== LinearSpace(GroundField) with {

        index : (%, SI) -> GroundField;
        apply : (%, SI) -> GroundField;
```

```
        canonicalBasisVector : (SI, GroundField) -> %;
        unitCanonicalBasisVector : SI -> %;

        default {
                apply(v : %, i : SI) : GroundField == index(v, i);

                import from GroundField;
                unitCanonicalBasisVector(i : SI) : % == canonicalBasisVector(i, 1);
        }

}


define FiniteIndexedVector(GroundField : Field) : Category
== LinearSpace(GroundField) with {
        size : % -> SI;
}


define Matrix(GroundField : Field,
              V : LinearSpace(GroundField),
              W : IndexedVector(GroundField)) : Category
== LinearMapping(GroundField, V, W) with {

        apply : (%, SI) -> V;
        column : (%, SI) -> V;

        default {
           apply(A : %, i : SI) : V == column(A, i);
        }
}


define SquareMatrix(GroundField : Field,
                    V : IndexedVector(GroundField)) : Category
== Join(LinearOperator(GroundField, V),
        Matrix(GroundField, V, V)) with {

        apply : (%, SI, SI) -> GroundField;
}


define UpperTriangularMatrix(GroundField : Field,
                             V : IndexedVector(GroundField)) : Category
== SquareMatrix(GroundField, V) with;
```

```
define LowerTriangularMatrix(GroundField : Field,
                             V : IndexedVector(GroundField)) : Category
== SquareMatrix(GroundField, V);


define LowerBanded : Category == with {
        lowerBandWidth : % -> SI;
}


define UpperBanded : Category == with {
        upperBandWidth : % -> SI;
)


define UpperHessenbergMatrix(GroundField : Field,
                             V : IndexedVector(GroundField)) : Category
== Join(LowerBanded,
        SquareMatrix(GroundField, V)) with {

        default {
                lowerBandWidth(A : %) : SI == 1;
        )

)


define BandedUpperHessenbergMatrix(GroundField : Field,
                             V : IndexedVector(GroundField)) : Category
== Join(UpperHessenbergMatrix(GroundField, V),
        UpperBanded) with;


define LowerHessenbergMatrix(GroundField : Field,
                             V : IndexedVector(GroundField)) : Category
== Join(UpperBanded,
        SquareMatrix(GroundField, V)) with {

        default {
                upperBandWidth(A : %) : SI == 1;
        )

)


define BandedUpperTriangularMatrix(GroundField : Field,
                             V : IndexedVector(GroundField)) : Category
== Join(UpperTriangularMatrix(GroundField, V), UpperBanded) with;
```

```
define TriDiagonalMatrix(GroundField : Field,
                         V : IndexedVector(GroundField)) : Category
== Join(LowerHessenbergMatrix(GroundField, V),
        UpperHessenbergMatrix(GroundField, V)) with;
```

- The use of the `pretend` keyword in the definition of the linear operator on an inner product space is due to the problems with typing the `InnerProductSpace` category as a `LinearSpaceWithDual` of itself, as discussed in Section 5.3.1.1.

- The `explicitMapping` functions are a way of forgetting type information, by turning objects into general functions. These functions can exist because losing information and turning what may be a strict function into a lazy object is never a problem. The converse is more problematic, that is having a constructor for a category that takes the general function and then pretends it is e.g. a linear operator. There may be no way of checking a function for a given property, or such a check may be horrendously inefficient, and expanding a lazy object into a strict representation may not terminate etc. Similar reasoning is the basis of the philosophy of never including constructors into general categories, applied throughout the code – instead, they are attached as anonymous extensions when typing a specific domain.

- Linear mappings/operators are not vector spaces, as some conceivable domains that one may wish to type using these categories do not have that structure, such as a domain of nonsingular matrices. Similarly, the group action category is not a linear space as the group may not be closed under linearity. However, the operator and group action categories share the notion that they can act on some other type of object, and the linear mappings category introduces the functions for linearity by hand. Introducing more roots into the hierarchy such as generic *mapping*, *action*, and *linear* categories would help clean this up. An affine space category would also be a useful addition.

C.2 Problem Specific Categories

Due to the large number of parameter domains to categories in this section, including linear spaces and domains of matrices, the naming conventions are different to the previous section. The domain of coefficients for the projected linear operator, ground field and valuation are labelled directly as such. The matrix of Krylov space basis vectors, and the domains of elements used to construct the projected system and search recurrence are labelled with a name composed of a letter related to those used in Chapter 3 (with lowercase letters for domains of vectors and uppercase letters for domains of matrices) and the suffix Dom. In the instance of multiple identifiers in Chapter 3 being used for elements of the same domain (e.g. the vectors y and z) one of the labels is chosen arbitrarily.

```
define KrylovSpace(Valuation : OrderedField,
                  GroundField : FieldWithValuation(Valuation),
                  Coeffs : FieldWithValuation(Valuation),
                  yDom : IndexedVector(Coeffs),
                  Vector : Join(LinearSpace(Coeffs),
                                NormedLinearSpace(Valuation, GroundField)),
                  Operator : LinearOperator(GroundField, Vector),
                  VDom : Matrix(Coeffs, Vector, yDom),
                  HDom : UpperHessenbergMatrix(Coeffs, yDom)
) : Category
== with (
        basis : % -> VDom;
        coefficients : % -> HDom;
        operator : % -> Operator;
        startVector : % -> Vector;
}

define DirectLUSolve(GroundField : Field,
                    zDom : IndexedVector(GroundField),
                    HDom : UpperHessenbergMatrix(GroundField, zDom),
                    UDom : UpperTriangularMatrix(GroundField, zDom)) : Category
== with (
        directLU : (HDom, zDom) -> (UDom, zDom);
}

define DirectQRSolve(GroundField : Field with ( sqrt : % -> % ),
                    zDom : IndexedVector(GroundField),
                    HDom : UpperHessenbergMatrix(GroundField, zDom),
                    RDom : UpperTriangularMatrix(GroundField, zDom)) : Category
== with {
        directQR : (HDom, zDom) -> (RDom, zDom, zDom);
}
```

```
define LongRecurrenceKrylovSpace(Valuation : OrderedField,
                GroundField : FieldWithValuation(Valuation),
                Vector : NormedInnerProductSpace(Valuation, GroundField),
                Operator : LinearOperatorOnInnerProductSpace(GroundField;
                                                              Vector),
                yDom : FiniteIndexedVector(GroundField),
                VDom : Matrix(GroundField, Vector, yDom),
                HDom : UpperHessenbergMatrix(GroundField, yDom)
) : Category
== KrylovSpace(Valuation,
              GroundField,
              GroundField,
              yDom,
              Vector,
              Operator,
              VDom,
              HDom) with {

        orthonormalKrylovBasis : (Operator, Vector) -> %;


        default {
           iterativeSolve(correction : (HDom, Valuation) -> yDom)
                         (A : Operator,
                          x : Vector,
                          b : Vector) : Vector == {

               import from VDom;

               if x = 0
                 then r := b;
                 else r := b - A x;
               rNorm : Valuation := norm r;
               K := orthonormalKrylovBasis(A, r/rNorm);

               H := coefficients(K);
               V := basis(K);

               y := correction(H, rNorm);
               x := x + V y;

               return x;
           }
        }

}
```

```
define ShortRecurrenceKrylovSpace(Valuation : OrderedField,
                                  GroundField : FieldWithValuation(Valuation),
                                  Coeffs : FieldWithValuation(Valuation),
                                  yDom : IndexedVector(Coeffs),
                                  Vector : Join(LinearSpace(Coeffs),
                                                NormedLinearSpace(Valuation,
                                                                  GroundField),
                                                with { dispose! : % -> (); }),
                                  Operator : LinearOperator(GroundField, Vector),
                                  VDom : Matrix(Coeffs, Vector, yDom),
                                  HDom : UpperHessenbergMatrix(Coeffs, yDom)
) : Category
== KrylovSpace(Valuation, GroundField, Coeffs, yDom, Vector,
               Operator, VDom, HDom) with {

        iterativeSolve : ((Operator, Vector) -> %,
                          (HDom, VDom, Valuation) -> (VDom, SI -> Boolean)) ->
                                  (Operator, Vector, Vector) ->
                                          Vector;

        default {
           iterativeSolve(krylovBasis : (Operator, Vector) -> %,
                          correction : (HDom, VDom, Valuation)
                                 -> (yDom, VDom, SI -> Boolean))
                         (A : Operator,
                          x : Vector,
                          b : Vector) : Vector == {

               import from SI, VDom;

               if x = 0
                 then r := b;
                 else r := b - A x;
               rNorm : Valuation := norm r;
               K := krylovBasis(A, r/rNorm);
               H := coefficients(K);
               V := basis(K);

               (z, P, lastIteration?) := correction(H, V, rNorm);
               for i in 1.. repeat {
                       xNew := x + z(i) * P(i);
                       dispose!(x); x := xNew;
                       if lastIteration?(i) then break;
               }
               return x;
           }
        }
}
```

```
define ArbitraryBasisKrylovSpace(Valuation : OrderedField,
                   GroundField : FieldWithValuation(Valuation),
                   Vector : NormedInnerProductSpace(Valuation, GroundField)
                       with { dispose! : % -> (); },
                   Operator : LinearOperatorOnInnerProductSpace(GroundField, Vector),
                   yDom : IndexedVector(GroundField),
                   VDom : Matrix(GroundField, Vector, yDom),
                   HDom : BandedUpperHessenbergMatrix(GroundField, yDom)
) : Category
== ShortRecurrenceKrylovSpace(Valuation,
                              GroundField,
                              GroundField,
                              yDom,
                              Vector,
                              Operator,
                              VDom,
                              HDom) with {

        incompletelyOrthogonalKrylovBasis : SI -> (Operator, Vector) -> %;
}


define BiorthogonalBasisKrylovSpace(Valuation : OrderedField,
                   GroundField : FieldWithValuation(Valuation),
                   DualSpace : NormedLinearSpace(Valuation, GroundField)
                       with { dispose! : % -> (); },
                   Vector : NormedLinearSpaceWithDual(Valuation, GroundField, DualSpace)
                       with { dispose! : % -> (); },
                   Operator : LinearOperatorWithDual(GroundField, DualSpace, Vector),
                   yDom : IndexedVector(GroundField),
                   VDom : Matrix(GroundField, Vector, yDom),
                   TDom : TriDiagonalMatrix(GroundField, yDom)
) : Category
== ShortRecurrenceKrylovSpace(Valuation,
                              GroundField,
                              GroundField,
                              yDom,
                              Vector,
                              Operator,
                              VDom,
                              TDom) with {

        biorthogonalKrylovBasis : DualSpace -> (Operator, Vector) -> %;
        biStartVector : % -> DualSpace;
}
```

```
define HermitianOperatorKrylovSpace(Valuation : OrderedField,
                GroundField : FieldWithValuation(Valuation),
                Vector : NormedInnerProductSpace(Valuation, GroundField)
                    with { dispose! : % -> (); },
                Operator : HermitianLinearOperator(Valuation, GroundField, Vector),
                yDom : IndexedVector(Valuation),
                VDom : Matrix(Valuation, Vector, yDom),
                TDom : TriDiagonalMatrix(Valuation, yDom)
) : Category
== ShortRecurrenceKrylovSpace(Valuation,
            GroundField,
            Valuation pretend FieldWithValuation(Valuation),
            yDom,
            Vector,
            Operator,
            VDom,
            TDom) with {

        orthonormalKrylovBasis : (Operator, Vector) -> %

}


define SearchVectorRecurrence(Coeffs : Field,
                zDom : IndexedVector(Coeffs),
                Vector : LinearSpace(Coeffs),
                VDom : Matrix(GroundField, Vector, zDom),
                RDom : BandedUpperTriangularMatrix(GroundField, zDom)) : Category
== with {
        recurrence: (V : VDom, R : RDom) -> VDom;
}
```

- The type of a parameter to a category can be the union of a named category and an anonymous extension. This is used, for example, to provide the square root function necessary for computing Givens rotations used in `DirectQRSolve`, which is not provided by a generic `Field`.

- As mentioned in Chapter 5, it may be desirable to use a different type to represent the ground field of the general vector space and the coefficients of the projected matrix. This is captured by having both types as parameters to a general `KrylovSpace`, ensuring that the vector space is also a linear space over the coefficient type, and using the derived categories to specify the type of the coefficients – i.e. either the ground field (for the biorthogonal or incompletely orthogonal methods) or the valuation domain (for the

Hermitian method). This means that the three derived categories do not have one of the parameters to the general short recurrence Krylov space category. When the coefficient type is the valuation (i.e. for the Hermitian method) the `pretend` keyword is used to assert that it is a field whose valuation is itself to circumvent the difficulty with incorporating this constraint into the type requirement for the valuation domain parameter. This is discussed in Section 5.3.1.1.

- The `dispose!` function appears in the template algorithm contained in the short recurrence Krylov space category due to reasons discussed in Section 5.3.1.2.

Appendix D

Domains

This appendix contains code extracts to further illustrate the design of various domains, including those involved with the solver algorithms themselves (discussed in Chapter 5) and the implementations of the linear systems (discussed in Chapter 6).

Unlike the categories in Appendix C, code in this appendix is abridged for conciseness. Some directives such as `import` and `inline` have been dropped, some functions (and related exports) have been omitted when they are similar to those already included or are simple enough to need no explanation, and the less important tests for common errors have been removed. Macro abbreviations[1] are also used, namely:

- `SI` for `SingleInteger` (same as Appendix C)
- `DF` for `DoubleFloat`
- `CDF` for `ComplexDoubleFloat`
- `CV` for `ColourVector`
- `SpF` for `SpinorField`

Another difference from Appendix C is that the code has been presented in its original form as opposed to that which is actually used for the experiments. More specifically, a number of alterations to the code that had to be made due to problems with the current compiler are not present. These include removing the parameterisation of certain domains, the manual unboxing of the result of the Wilson-Dirac stencil term (both due to problems with the inliner), unboxing

[1]For a description of Aldor macros, see [93]

hints for reduction operations and workarounds for the lack of constant folding on double precision floatingpoint values, all of which are discussed in Section 9.3. One further detail that has been left out of the code is a workaround to sidestep the problems that the current compiler has with type-checking domains whose domain representation is a function. This applies to the "lazy" matrices/vectors.

D.1 Scalar Domains

```
extend DoubleFloat : Join(OrderedField, FieldWithValuation(DF), Module(SI)) with {

  sqrt : % -> %;
  conjugate : % -> %;

} == add {

  valuation( x : % ) : DF == abs(x);
  coerce(a : DF) : % == a pretend DF;
  conjugate(x : %) : % == x;
  sqrt(a : %) : % == sqrt(a)$DoubleFloatElementaryFunctions;
}


ComplexDoubleFloat : Join(FieldWithValuation(DF), Module(SI)) with {

  sqrt: % -> %;
  conjugate: % -> %;

} == Complex(DF) add {

  Rep == Complex(DF);

  valuation(a : %) : DF == {
    imag a = zero() => abs(real a);
    real a = zero() => abs(imag a);
    sqrt(norm(rep a))@DF;
  }

  sqrt(a : %) : % == per (sqrt(rep a)$DoubleFloatElementaryFunctions);
  conjugate(a : %) : % == per (conjugate rep a);
}
```

- The `valuation` operation for complex double floats tests for the simple cases when either the real or imaginary parts are zero to avoid using the square root operation un-

necessarily, as it can reduce numerical stability. The valuation of a complex number with zero imaginary part occurs for inner products that implement the normsq to the valuation domain using the default method in NormedInnerProductSpace.

- Both the double float and complex double float domains are extensions of the original domains taken from the axllib library.

- The conjugation operation for the double float domain exists to satisfy the requirements of the parameterised solvers, but does nothing.

D.2 Wilson-Dirac Subdomains

```
ColourVector : Join(Module(SI), InnerProductSpace(CDF)) with {

  bracket : (CDF, CDF, CDF) -> %;
  apply : (%, SI) -> CDF;

} == add {

  Rep == Record(a : CDF, b : CDF, c : CDF);

  apply(v : %, i : SI) : CDF == {
    i = 0 => (rep v).a;
    i = 1 => (rep v).b;
    i = 2 => (rep v).c;
    never;
  }

  bracket(a : CDF, b : CDF, c : CDF) : % == per [a, b, c];

  (a : SI) * (v : %) : % == [a * v(0), a * v(1), a * v(2)];

  (a : CDF) * (v : %) : % == [a * v(0), a * v(1), a * v(2)];

  (v : %) + (w : %) : % == [v(0) + w(0), v(1) + w(1), v(2) + w(2)];

  (v : %) * (w : %) : CDF == {
    v(0) * conjugate(w(0)) + v(1) * conjugate(w(1)) + v(2) * conjugate(w(2))
  }
}
```

```
Spinor4 : InnerProductSpace(CDF) with {

  apply : (%, SI) -> CV;
  bracket : (CV, CV, CV, CV) -> %;

} == add {

  Rep == Record(a : CV, b : CV, c : CV, d : CV);

  bracket(a : CV, b : CV, c : CV, d : CV) : % == per [a, b, c, d];

  apply(s : %, i : SI) : CV == {
    i = 0 => (rep s).a;
    i = 1 => (rep s).b;
    i = 2 => (rep s).c;
    i = 3 => (rep s).d;
    never;
  }

  (a : CDF) * (v : %) : % == [a * v(0), a * v(1), a * v(2), a * v(3)];

  (v : %) + (w : %) : % == [v(0) + w(0), v(1) + w(1), v(2) + w(2), v(3) + w(3)];

  (v : %) * (w : %) : CDF == {
    v(0) * w(0) + v(1) * w(1) + v(2) * w(2) + v(3) * w(3);
  }
}
```

```
SU3 : GroupAction(CDF, CV) with {

  bracket : (CDF, CDF, CDF,
             CDF, CDF, CDF,
             CDF, CDF, CDF) -> %;

} == add {

  Rep == Record(a: CDF, b: CDF, c: CDF,
                d: CDF, e: CDF, f: CDF,
                g: CDF, h: CDF, i: CDF);

  bracket(a: CDF, b: CDF, c: CDF,
          d: CDF, e: CDF, f: CDF,
          g: CDF, h: CDF, i: CDF) : % == per record(a, b, c, d, e, f, g, h, i);
```

```
  inv(M : %) : % == (
    A := (rep M);
    [ conjugate(A(a)), conjugate(A(d)), conjugate(A(g)),
      conjugate(A(b)), conjugate(A(e)), conjugate(A(h)),
      conjugate(A(c)), conjugate(A(f)), conjugate(A(i))];
  }

  (M : %) * (v : CV) : CV == {
    A := (rep M);
    [ A(a) * v(0)  +  A(b) * v(1)  +  A(c) * v(2),
      A(d) * v(0)  +  A(e) * v(1)  +  A(f) * v(2),
      A(g) * v(0)  +  A(h) * v(1)  +  A(i) * v(2)];
  }

  (M : %) \ (v : CV) : CV == {
    A := (rep M);
    [ conjugate(A(a)) * v(0)  +  conjugate(A(d)) * v(1)  +  conjugate(A(g)) * v(2),
      conjugate(A(b)) * v(0)  +  conjugate(A(e)) * v(1)  +  conjugate(A(h)) * v(2),
      conjugate(A(c)) * v(0)  +  conjugate(A(f)) * v(1)  +  conjugate(A(i)) * v(2)];
  }
}



Projector : with {

  gamma1pos : (SU3, Spinor4) -> Spinor4;
  gamma1neg : (SU3, Spinor4) -> Spinor4;
   ...
  gamma4pos : (SU3, Spinor4) -> Spinor4;
  gamma4neg : (SU3, Spinor4) -> Spinor4;

} == add {

  i ==> complex(0.0, 1.0);

  gamma4pos(U : SU3, s : Spinor4) : Spinor4 == {
    u0 := U * (s(0) + i * s(2));
    u1 := U * (s(1) - i * s(3));
    return [u0, u1, (-i) * u0, i * u1];
  }

  gamma4neg(U : SU3, s : Spinor4) : Spinor4 == {
    u0 := U \ (s(0) - i * s(2));
    u1 := U \ (s(1) + i * s(3));
    return [u0, u1, i * u0, (-i) * u1];
  }
}
```

- In keeping with the rest of the code, constructors (here presented as `bracket` functions) are added in anonymous category extensions.
- Among the missing details are the packed array functions that are used by the `SpinorField` domain, and the 0 element for the linear spaces.
- The subdomains (and the vector and operator domains that use them) are not parameterised. It may be useful to introduce some degree of parameterisation, for example with the aim of being able to use the same domains for a different gauge theory (which would involve different size gauge matrices and colour vectors etc). However, other forms of parameterisation may be less meaningful. For instance, it is less obvious how to parameterise over the scalar domain.

D.3 Vector and Operator Domains

```
Vector3D : Join(NormedInnerProductSpace(DF, CDF) with {

  apply : (%, SI) -> CDF;
  apply : (%, SI, SI) -> CDF;
  set! : (%, SI, CDF) -> CDF;

} == add {

  Rep == PackedArray(CDF);
  dim ==> xDimension * yDimension * zDimension;

  set!(v: %, i: SI, a: CDF): CDF == set!(rep v, i, a);

  apply(v: %, i: SI): CDF == apply(rep v, i);


  apply(v : %, i : SI, mu : SI) : CDF  == {
    latticeDimensions : SI == 3;
    pointsPerLatticeDim : SI == 2;
    entriesPerSite : SI == latticeDimensions * pointsPerLatticeDim + 1;
    centre : SI == (entriesPerSite quo 2) + 1;

    jump : SI == entriesPerSite;
    index := offsetTableGlobal((jump * (i-1)) + centre + mu);
    return (rep v)(index);
  }
```

```
  (x: CDF) * (a: %) : % == {
    result : % := new();
    for i in 1..dim repeat result(i) := x*a(i);
    result
  }

  (a: %) + (b: %) : % == {
    result : % := new();
    for i in 1..dim repeat result(i) := a(i) + b(i);
    result
  }

  (v: %) * (w: %): CDF == {
    ip : CDF := 0;
    for i in 1..dim repeat ip := ip + v(i) * conjugate w(i);
    ip
  }

  normsq(a: %): CDF == v * w;
  norm(a: %): CDF == sqrt(normsq(a)@CDF);
  norm(a: %): DF == valuation(norm(a)@CDF);
}



SimpleOperator3D : LinearOperatorOnInnerProductSpace(CDF, Vector3D)
== add {

  Rep == Record(kappa : CDF);
  dim ==> xDimension * yDimension * zDimension;

  apply(A : %, v : Vector3D, i : SI) : CDF == {
     r := v(i, 1) + v(i, -1) +
          v(i, 2) + v(i, -2) +
          v(i, 3) + v(i, -3)
            - 6*v(i);
          return rep(A).kappa * r;
  }


  (A : %) * (v : Vector3D) : Vector3D == {
    u : Vector3D := new();
    for i in 1..dim repeat u(i) := apply(A, v, i);
    return u;
  }

  adjoint(A : %) : % == per record(conjugate (rep A).kappa)
}
```

- The simple 4D operator is a simple generalisation of the one given above.

- Constructor and destructor functions (`new` and `dispose!`) have been left out as they are very simple (they both directly call equivalents from the underlying domain representation).

- The grid dimensions (`xDimension`, `yDimension` etc) used by the vector and operator domains are lexically scoped constants. This scheme illustrates one approach to the problem of permitting symbolic constants whilst still being able to prove the conformability of loops from separate domains, as discussed in Section 9.1.2. A simpler scheme would be to require loop dimensions to be known compile-time constants, in which case proving conformability is trivial. The offset table (`offsetTableGlobal`) is also a lexical variable.

- Rather than writing multiple loops, vector/operator functions could be written in terms of higher order functionals such as `map` etc. However, the loops are already so concise that not much would be gained in terms of presentation, and implementing any functionals themselves as loops leads to the same FOAM code after inlining, so the differences are marginal.

- The simple operators support an explicit `adjoint` operation as it is cheap, unlike, for example, explicitly taking the adjoint of an element from the Wilson-Dirac domain.

```
SpinorField : NormedInnerProductSpace(DF, CDF) with {

  apply : (%, SI) -> Spinor4;
  apply : (%, SI, SI) -> Spinor4;
  set! : (%, SI, Spinor4) -> Spinor4;

} == add {

  Rep == PackedArray(Spinor4);
  dim ==> tDimension * xDimension * yDimension * zDimension;

  set!(v: %, i: SI, a: Spinor4): Spinor4 == set!(rep v, i, a);
  apply(v: %, i: SI): Spinor4 == apply(rep v, i);

  apply(v : %, i : SI, mu : SI) : Spinor4  == {
    latticeDimensions : SI == 4;
    pointsPerLatticeDim : SI == 2;
    entriesPerSite : SI == latticeDimensions * pointsPerLatticeDim + 1;
```

```
    centre : SI == (entriesPerSite quo 2) + 1;

    jump : SI == entriesPerSite;
    index := offsetTableGlobal((jump * (i-1)) + centre + mu);
    return (rep v)(index);
  }


  (v: %) * (w: %): CDF == {
    ip : CDF := 0;
    for i in 1..dim repeat ip := ip + v(i) * w(i);
    ip
  }
}



NaturallyOrderedWilsonDiracOperator : LinearOperatorWithDual(CDF, DualSpF, SpF)
== add {

  Rep == Record(kappa : CDF, gaugeField : PackedArray(SU3));
  dim ==> tDimension * xDimension * yDimension * zDimension;


  apply(U : PackedArray(SU3), i : SI, mu : SI) : SU3 == {
    latticeDimensions : SI == 4;
    pointsPerLatticeDim : SI == 2;
    entriesPerSite : SI == latticeDimensions * pointsPerLatticeDim + 1;
    centre : SI == (entriesPerSite quo 2) + 1;

    offsetJump : SI == entriesPerSite;
    gaugeJump := latticeDimensions;

    { mu > 0 => index := (gaugeJump * (i-1)) + mu;
      mu < 0 => {
        lookup := (offsetJump * (i-1)) + centre + mu;
        index := gaugeJump * offsetTableGlobal(lookup) - mu;
      }
      never;
    }

    return U(index);
  }


  apply(A : %, v : SpF, i : SI) : Spinor4 == {

    U := (rep A).gaugeField;
```

```
    k := rep(A).kappa;

    r : Spinor4 := zero();
    r := r + gamma1pos(U(i, 1), v(i, 1))
           + gamma1neg(U(i, -1), v(i, -1));
    r := r + gamma2pos(U(i, 2), v(i, 2))
           + gamma2neg(U(i, -2), v(i, -2));
    r := r + gamma3pos(U(i, 3), v(i, 3))
           + gamma3neg(U(i, -3), v(i, -3));
    r := r + gamma4pos(U(i, 4), v(i, 4))
           + gamma4neg(U(i, -4), v(i, -4));

    return v(i) - k * r;
  }


  (A : %) * (v : SpF) : SpF == {
    u : SpF := new();
    for i in 1..dim repeat u(i) := A(v, i);
    return u;
  }
}
```

- The Wilson-Dirac vector and operator domains are very similar to the simple 3D domains. As a result, only a small amount of code is given for them to highlight the important differences.
- In the vector domain, the main differences are the number of dimensions, the element type of the loops (`Spinor4` objects rather than elements of `CDF`) and the fact that the inner product operation is implemented in terms of inner products on the elements rather than multiplication and conjugation. Having loops over `Spinor4` objects is what led to to implementing *loop rerolling* to keep the code size of the loops down.
- In the operator domain the differences include an index function to retrieve elements of the gauge field, and a more complex stencil term written using functions from the `Projector` package. The stencil term for the adjoint action has been omitted, but is very similar. Note, however, that it acts on members of `DualSpF` – this domain has been omitted, as it is an empty wrapper around the original `SpinorField` domain, which implements the action of the dual space using the inner product operation after casting the dual vector as a member of `SpinorField`. The domain exists only to satisfy the type requirements of the Wilson-Dirac operator being a `LinearOperatorWithDual`.

D.4 Solver Domains

```
LazyVector(GroundField : Field) : IndexedVector(GroundField) with {

  bracket : (SI -> GroundField) -> %;

) == add {

  Rep == SI -> GroundField;

  bracket(f : SI -> GroundField) : % == per f;

  index(v : %, i : SI) : GroundField == (rep v)(i);

  apply(v : %, i : SI) : GroundField == index(v, i);

  canonicalBasisVector(i : SI, coeff : GroundField) : % == {
    [(j : SI) : GroundField +-> if j = i then coeff else 0;];
  }
}


LazyMatrix(GroundField : Field,
           V : LinearSpace(GroundField),
           W : IndexedVector(GroundField)) : Matrix(GroundField, V, W) with {

  bracket : (SI -> V) -> %;

}== add {

  Rep == SI -> V;

  bracket(f : SI -> V) : % == per f;

  column(A : %, i : SI) : V == (rep A)(i);
}


LazyTriDiagMatrix(GroundField : Field,
                  yDom : IndexedVector(GroundField)
                    with { bracket : (SI -> GroundField) -> %; }
) : TriDiagonalMatrix(GroundField, yDom) with {

  bracket : (SI -> Record(u:GroundField, d:GroundField, l:GroundField)) -> %;

} == add {
```

```
  Rep == SI -> Record(u:GroundField, d:GroundField, l:GroundField);

  bracket(f : SI -> Record(u:GroundField, d:GroundField, l:GroundField)) : % == per f;

  column(A : %, j : SI) : yDom == {
    r := (rep A)(j);
    [(i : SI) : GroundField +-> {
      i = j => r.d;
      i+1 = j => r.u;
      i-1 = j => r.l;
      0;
    }];
  }

  apply(A : %, i : SI, j : SI) : GroundField == {
    i < j-1 => 0;
    i > j+1 => 0;
    r := (rep A)(j);
    j = i+1 => r.u;
    j = i => r.d;
    j = i-1 => r.l;
    never;
  }
}


ThreeBandedRFactor(GroundField : Field,
                   yDom : IndexedVector(GroundField)
                     with {bracket : (SI -> GroundField) -> %;}
                  ) : BandedUpperTriangularMatrix(GroundField, yDom) with {

  bracket : (SI -> Record(d:GroundField, u1:GroundField, u2:GroundField)) -> %;

}== add {

  Rep == SI -> Record(d:GroundField, u1:GroundField, u2:GroundField);

  bracket(f : SI -> Record(d:GroundField, u1:GroundField, u2:GroundField)) : % == per f;

  column(U : %, j : SI) : yDom == {
    r : Record(d:GroundField, u1:GroundField, u2:GroundField) := (rep U)(j);
    [(i : SI) : GroundField +-> {
      i = j => r.d;
      i+1 = j => r.u1;
      i+2 = j => r.u2;
      0;
    }];
  }
```

```
  apply(U : %, i : SI, j : SI) : GroundField == {
    r : Record(d:GroundField, u1:GroundField, u2:GroundField) := (rep U)(j);
    i = j => r.d;
    (i+1) = j => r.u1;
    (i+2) = j => r.u2;
    0;
  }

  upperBandWidth(A : %) : SI == 2;
}
```

- The matrix domains outlined above are very simple wrappers around one of the recurrences described later in the appendix. Although they are typed using the `Matrix` category (or some derivative thereof) they do not support any of the exports demanded by `LinearMapping`. There are two reasons for this. The first is that using a lazy representation for a domain makes it more difficult to implement operations that manipulate elements – this applies to the multiplication/division by a scalar. The second is that calculating the result of the linear mapping itself is not obvious when the size of both the matrix and vector may be unbounded, as the standard procedure for forming a linear combination of the columns of the matrix will not terminate.

- Constructors are added as part of an anonymous category extension, either to the domain itself or to its parameters as required.

```
BiKrySpc(Valuation : OrderedField,
         GroundField : FieldWithValuation(Valuation)
           with { conjugate: % -> %; },
         DualVector : NormedLinearSpace(Valuation, GroundField)
           with { copy : % -> %; dispose! : % -> (); },
         Vector : NormedLinearSpaceWithDual(Valuation, GroundField, DualVector)
           with { copy : % -> %; dispose! : % -> (); },
         Operator : LinearOperatorWithDual(GroundField, DualVector, Vector),
         yDom : IndexedLinearSpace(GroundField),
         VDom : Matrix(GroundField, Vector, yDom)
           with { bracket : (SI -> Vector) -> %; },
         TDom : TriDiagonalMatrix(GroundField, yDom)
           with { bracket : (SI -> Record(u:GroundField,
                                          d:GroundField,
                                          l:GroundField)) -> %; }
  : BiorthogonalBasisKrylovSpace(Valuation, GroundField,
                                 DualVector, Vector, Operator,
                                 yDom, VDom, TDom)
} == add {
```

```
Rep == Record(A : Operator,
              start : Vector, biStart : DualVector,
              V : VDom,
              T : TDom);

biorthogonalKrylovBasis(argBiStart : DualVector)
                       (A: Operator, argStart: Vector) : % == {

  cachedBiStart := copy argBiStart;
  cachedStart := copy argStart;

  normStart : Valuation := norm cachedStart;
  if not ((normStart - 1) * (normStart - 1) << 1 ) then {
    error "[BiKrySpc]␣Start␣vector␣not␣normal";
  }

  AH ==> A;    -- convention to indicate use of Hermitian transpose

  local{
    v1 : Vector; v2 : Vector;
    w1 : DualVector; w2 : DualVector;
    alpha : GroundField; beta : GroundField; gamma : GroundField;
    delta : GroundField; deltaOld : GroundField;
    state : SI := 0;
  }

  goToState!(i : SI) : () == {

    free { state; v1; v2; w1; w2; alpha; beta; gamma; delta; deltaOld; }

    step!() : () == {
      free { state; v1; v2; w1; w2; alpha; beta; gamma; delta; deltaOld; }

      t1 := AH w1 - conjugate(alpha) * w1 - conjugate(beta) * w2;
      dispose!(w2); w2 := t1;
      (deltaOld, deltaTemp) := (delta, (v2 * w2));
      delta := deltaTemp / (gamma * gamma);

      (v1, v2) := (v2/gamma, v1);
      (w1, w2) := (w2/conjugate(gamma), w1);

      u := A v1;
      alpha := (u * w1)/delta;
      beta := gamma * delta / deltaOld;
      t2 := u - alpha * v1 - beta * v2;
      dispose!(v2); v2 := t2;
      gamma := norm v2;
```

```
      state := state + 1;

      if valuation(deltaTemp) = 0
      then error "[BiKrySpc]␣Serious␣breakdown␣in␣BiLanczos␣process";
    }


    i = state => return;
    i = 1 => {

      v1 := copy cachedStart;
      w1 := copy cachedBiStart;

      delta := (v1 * w1);

      deltaOld := 0;
      u := A v1;
      alpha := (u * w1)/delta;
      beta := 0;
      v2 := u - alpha * v1;
      gamma := norm v2;

      state := 1;
    }

    i > state => {
      if state = 0
      then goToState!(1);
      for j in 1..i-state repeat step!();
    }

    i < state => { goToState!(1); goToState!(i); }
  }


  vColumnAccess(i : SI) : Vector == {
    goToState!(i);
    return v1;
  }

  tColumnAccess(i : SI) : Record(u:GroundField, d:GroundField, l:GroundField) == {
    goToState!(i);
    return record(beta, alpha, gamma);
  }

  return per record(A, cachedStart, cachedBiStart, [vColumnAccess], [tColumnAccess]);
}
```

```
  basis(k : %) : VDom == rep(k).V;

  coefficients(k : %) : TDom == rep(k).T;

  operator(k : %) : Operator == rep(k).A;
}
```

- As indicated by a comment in the code, the use of the `AH` macro instead of the identifier `A` is to indicate to the reader that the Hermitian transpose of the operator being applied to a (dual) vector. No such direction is necessary for the compiler however, as the operation to use (i.e. apply the original or the transpose) is specified by the type of the object being acted on (i.e. a vector or a dual vector).
- The Krylov space generating algorithm deals explicitly with conjugation of scalars if they are complex, and so the requirement for a `conjugate` export is added to the ground field parameter.
- The constructors necessary for wrapping the recurrences defined by the domain are added as requirements to the domain for the matrix of basis vectors and the tridiagonal matrix of coefficients. This addition of constructor exports occurs for the domains below as well.

```
tridiagDirectQRSolve(Valuation : OrderedField,
                     GroundField : FieldWithValuation(Valuation)
                       with { conjugate: % -> %; sqrt : % -> %; },
                     zDom : IndexedVector(GroundField)
                       with { bracket : (SI -> GroundField) -> %; },
                     TDom : TriDiagonalMatrix(GroundField, zDom),
                     RDom : BandedUpperTriangularMatrix(GroundField, zDom)
                       with { bracket : (SI -> Record(d:GroundField,
                                                      u1:GroundField,
                                                      u2:GroundField)) -> %; }
          ) : DirectQRSolve(GroundField, zDom, TDom, RDom)
== add {

  directQR(T : TDom, y : zDom) : (RDom, zDom, zDom) == {

    local {
      state : SI := 0;
      R : Record(d : GroundField, u1 : GroundField, u2 : GroundField) := [0, 0, 0];
      c : GroundField; s : GroundField;
      cOld : GroundField; sOld : GroundField;
```

```
    z : GroundField; zTemp : GroundField;
  }

  goToState!(i : SI) : () == {

    free { state; R; c; s; cOld; sOld; z; zTemp; }

    step!() :() == {
      free { state; R; c; s; cOld; sOld; z; zTemp; }

      state := state + 1;

      R.u2 := sOld * T(state - 1, state);
      u1Temp := cOld * T(state - 1, state);

      R.u1 := (c * u1Temp) + (s * T(state, state));
      local dTemp := (c * T(state, state)) - (conjugate(s) * u1Temp);
      cOld := c;
      sOld := s;

      local r : GroundField;
      (c, s, r) := givensRotation(dTemp, T(state + 1, state));
      R.d := r;

      z := (c * zTemp) + (s * y(state+1));
      zTemp := (c * y(state+1)) - (conjugate(s) * zTemp);
    }



    i = state => return;

    i = 1 => {

      R.u2 := 0;
      R.u1 := 0;

      (c, s, r) := givensRotation(T(1,1), T(2,1));
      R.d := r;

      y1 := y(1);
      y2 := y(2);
      z := (c * y1) + (s * y2);
      zTemp := (c * y2) - (conjugate(s) * y1);

      state := 1;
    }
```

```
    i = 2 => {

      if state ~= 1 then goToState!(1);

      R.u2 := 0;

      R.u1 := (c * T(1,2)) + (s * T(2,2));
      dTemp := (c * T(2,2)) - (conjugate(s) * T(1,2));
      cOld := c;
      sOld := s;

      (c, s, r) := givensRotation(dTemp, T(3,2));
      R.d := r;

      y3 := y(3);
      z := (c * zTemp) + (s * y3);
      zTemp := (c * y3) - (conjugate(s) * zTemp);

      state := 2;
    }

    i > state => {
      for j in state..i-1 repeat {
        j = 0 => goToState!(1);
        j = 1 => goToState!(1);
        j > 1 => step!();
    }

    i < state => {
      if i < 3 then goToState!(i);
      else goToState!(2); goToState!(i);
    }
  }


  rColumn(i : SI) : Record(d : GroundField, u1 : GroundField, u2 : GroundField) == {
    goToState!(i);
    return R;
  }


  zEntry(i : SI) : GroundField == {
    goToState!(i);
    return z;
  }
```

```
    residualEntry(i : SI) : GroundField == {
      goToState!(i);
      return zTemp;
    }

    return ([rColumn], [zEntry], [residualEntry]);
  }
}
```

- Similarly to the Krylov space algorithm, the *QR* solve explicitly needs the operations of conjugation (for complex scalars) and square root, and these are added to the requirements of the scalar domain parameter. The need for both arises from the calculation of Givens rotations (the code for which has been omitted as it is entirely standard).

- The solve is specialised to tridiagonal matrices and this is reflected in the stricter type requirement for the domain of matrices to be decomposed (the category only requires upper Hessenberg matrices).

```
length2SearchRecurrence(GroundField : Field,
       zDom : IndexedVector(GroundField),
       Vector : LinearSpace(GroundField),
       VDom : Matrix(GroundField, Vector, zDom)
         with { bracket : (SI -> Vector) -> %; },
       RDom : BandedUpperTriangularMatrix(GroundField, zDom)
      ) : SearchVectorRecurrence(GroundField, zDom, Vector, VDom, RDom) == add {

  recurrence(V : VDom, R : RDom) : VDom == {

    local{ p1 : Vector; p2 : Vector; state : SI := 0; }

    goToState!(i : SI) : () == {

      free { p1; p2; state; }

      step!() : () == {
        free { p1; p2; state; }

        state := state+1;

        t1 := 1/R(state,state) * ( V(state) - R(state-1, state) * p1
                                            - R(state-2, state) * p2);
        dispose!(p2); p2 := t1;
        (p1, p2) := (p2, p1);
      }
```

```
      i = state => return;

      i = 1 => {
        p1 := ( 1/R(1,1) ) * V(1);
        state := 1;
      }

      i = 2 => {
        if state ~= 1 then goToState!(1);

        p2 := ( 1/R(2,2) ) * (V(2) - R(1, 2) * p1);
        (p1, p2) := (p2, p1);

        state := 2;
      }

      i > state => {
        for j in state..i-1 repeat {
          j = 0 => goToState!(1);
          j = 1 => goToState!(1);
          j > 1 => step!();
        }
      }

      i < state => {
        if i < 3 then goToState!(i);
        else goToState!(2); goToState!(i);
      }
    }


    pColumn(i : SI) : Vector == {
      i < 1 => error "[pColumn]␣out␣of␣bounds␣access";
      goToState!(i);
      return p1;
    }

    return [pColumn];
  }
}
```

- The search vector recurrence is specialised to upper triangular factors with a band width of two. However, this is not checked for by the type system, as it would require a separate type for every different band width. Although it has not been included, it would be easy enough to add a dynamic check on the band width value.

```
QMRWrapper(Valuation : OrderedField,
      GroundField : FieldWithValuation(Valuation),
      DualVector : NormedLinearSpace(Valuation, GroundField),
      Vector : NormedLinearSpaceWithDual(Valuation, GroundField, DualVector),
      Operator : LinearOperatorWithDual(GroundField, DualVector, Vector),
      zDom : IndexedLinearSpace(GroundField),
      VDom : Matrix(GroundField, Vector, zDom),
      TDom : TriDiagonalMatrix(GroundField, zDom),
      KDom : BiorthogonalBasisKrylovSpace(Valuation, GroundField, DualVector,
                                    Vector, Operator, zDom, VDom, TDom),
      RDom : UpperTriangularMatrix(GroundField, zDom),
      QRDecomp : DirectQRSolve(GroundField, zDom, TDom, RDom),
      SolveWithState : SearchVectorRecurrence(GroundField, zDom, Vector, VDom, RDom)
     ) : with {

       QMR : (Operator, Vector, Vector, Valuation) -> Vector;

} == add {

   QMR(A : Operator, x : Vector, b : Vector, t : GroundField) : Vector == {

     tolerance := valuation(t);

     minimumResidualCorrection(T : TDom, V : VDom, beta : Valuation) :
                              (zDom, VDom, SI -> Boolean) == {

       (R : RDom, z : zDom, res : zDom)
                := directQR(T, canonicalBasisVector(1, beta::GroundField));
       P := recurrence(V, R);

       lastIteration?(i : SI) : Boolean == {
         residual : GroundField := res(i);
         if valuation(residual) < tolerance then true else false;
       }

       return(z, P, lastIteration?);

     }

     solveFunction := iterativeSolve(biorthogonalKrylovBasis(b),
                                          minimumResidualCorrection);

     return solveFunction(A, x, b);
   }

 }
}
```

- Anonymous category extensions have been dropped from the parameter domains in this extract to prevent the code from becoming too cluttered.

Appendix E

Published Papers

This appendix lists the three papers related to this work that were published during the course of the thesis, and provides some notes as to how they relate to the thesis proper. The papers are presented in chronological order. The first was a workshop paper that accompanied a poster, and the second two were refereed conference papers.

E.1 "The Paraldor Project" – 2003

[7] – this early paper was written in conjunction with colleagues from physics, and was primarily intended for an audience from computational physics. The first half of the paper describes the advantages of the Aldor language model as compared to other better-known languages such as C/Fortran/C++/Java and macro systems. The second half of the paper describes a toy code for an initial investigation into performance questions. The benchmark is the standard conjugate gradients algorithm (as opposed to the use of a modular framework) written in terms of domain exports, acting on a fully dense operator (i.e. $n \times n$ matrix). The results compare the performance of this algorithm implemented using various different domains, including a pure Aldor version with or without some degree of manual memory management (i.e. use of `dispose!` functions), and a version that uses foreign function calls to operations written in C (for both matrix-vector multiplication and all vector operations), again with or without manual memory management. The baseline against which the different imple-

mentations are measured is a pure C version of the algorithm. The pure Aldor results are significantly worse (i.e. by a factor of more than 100) than the baseline. The C back-end does much better, but is still worse than the baseline.

The paper differs philosophically from the work in the thesis in that it recommends the use of high-performance libraries (or assembly kernels) to implement low-level domains, such as matrix-vector application and vector operations, regardless of the cost of modularity. This stems in part from the tradition of this approach in computational physics. Also, it makes no direct mention of developing a framework for iterative solvers rather than the recipe used for the benchmark.

Because this was exploratory work, there was no detailed study of the code generated by the Aldor compiler. Hence, there is no information on why the pure Aldor results are so bad even when manual memory management is used, but factors such as failing to inline and emerge any generators (or higher order functions such as `map`) used for loops over the low-level domains are capable of incurring this kind of dramatic penalty. Failure to optimise such constructs can easily happen due to the difficulties that the inliner has with e.g. parameterised domains, but this is often a case of selecting the correct command line option or simple compiler implementation issues. In addition, the use of a dense matrix for the benchmark (rather than a stencil) has certain implications. A C compiler could apply certain blocking optimisations to improve temporal locality, and the cost of the matrix-vector multiplication will vastly outweigh any other operation.

The concluding sections of the paper contain some important points. The introduction of modularity makes manual memory management difficult (a theme which also occurs in Section 5.3.1.2) and, at the same time, certain automatic garbage collection strategies may be sub-optimal for applications like the iterative solvers. Particularly, a mark+sweep tracing collector tends to encourage the creation of many vectors before they are recovered, leading to poor temporal locality of reference to memory, and the process of root finding in the stack and tracing pointers is too expensive to incur regularly enough to maintain temporal locality of vector objects. The proposed solution is that of *memory spaces*, where different classes of object can be assigned different garbage collection strategies. The motivation behind this was to provide a mechanism

to allow vector objects to be collected by reference counting, and also to eliminate the possibility of object pinning which accompanies conservative garbage collectors. Ultimately, this direction was abandoned due to a lack of specificity to the application area, and the potential difficulties of integrating the scheme into Aldor and its current compiler. The future work section mentions implementing multiple iterative solvers, eigenvalue solvers, low-level objects for QCD (gauge fields etc), and the high level algorithmic structures for Monte Carlo algorithms that make use of the iterative solvers. Progress has been made on all of these, except for the Monte Carlo algorithms.

E.2 "A Modular Iterative Solver Package in a Categorical Language" – 2003

[8] – this paper introduces and describes the algorithmic framework for the iterative solvers and its domain level implementation, as detailed in Chapter 5. In addition to this, it presents a continuation of the argument against other popular approaches, as started in the workshop paper, and a move toward examining cross-component optimisations. This includes benchmark results comparing a pure Aldor version of a full QMR solver derived from the framework and transformed by hand against a version that makes calls to binary level 1 ATLAS BLAS routines. The operator in question is a simple 3D operator, and the machine is a 1GHz Pentium 3 Coppermine. The results show a speedup of up to 1.42 for the hand transformed code over the BLAS version.

The arguments against other approaches will be summarised here, as they are not covered in the thesis. The main arguments against traditional third-generation languages such as C and Fortran is that they do not provide adequate support for abstraction, the type systems do not provide much security (in the presence of type casts etc), and certain parts of the language definitions can be a significant obstacle to optimisation (e.g. pointers). The main arguments against object-oriented languages is that class inheritance is the wrong abstraction for representing groups of mathematical objects, and that while object mechanisms allow a certain degree of generalisation they still rely heavily on type casts, in contrast with static dependent typing. The objection to class inheritance is that it represents a subset relationship (i.e. $\subset$) rather than member-

ship (i.e. $\in$) as defined by categories. For example, given a class called `group`, two classes that inherit from it representing different groups are not both subsets of some larger group – elements of the different groups cannot interact. In addition, object systems are weak when it comes to specifying binary operations, as the model is based on sending messages to one object that owns the function being invoked.

A notable addition to the list of rejected alternatives is the use of *expression templates* [89], an example of compile time meta-programming based on the C++ template mechanism. In the paper it is dismissed based on arguments against its implementation – i.e. the inherent disadvantages of using any macro system (lack of type checking, semantic analysis etc), and the inherited problems from the underlying language being manipulated by the macros. In addition to this, the flaws in the general approach of meta-programming ought to be highlighted, namely that it is fundamentally a static approach that does not incorporate feedback from empirical evaluation of transformations, and therefore are no more adaptable to architectural differences than the static approaches to LDG optimisation given in Chapter 8.

The future work section lists formalising the transformations done by hand, and possible extensions to the general framework (a subset of those suggested in Chapter 11 as future work). The formalisation of the transformations has been covered in this thesis – see Chapters 7 and 8.

E.3 "Cross-Component Optimisation in a High Level Category Based Language" – 2004

[9] – this paper is mostly an expansion of the experimental results presented in [8], again investigating a QMR solver derived from the algorithmic framework, with a simple 3D operator, on a 1 GHz Pentium 3 Coppermine. The paper is presented in terms of cross-component optimisation. A more detailed description of the actual (hand) transformations is included, with several different variations that compare different levels of aggressiveness for fusion, and the introduction of the Fortran program QMRpack as another control. The results plot the relative performance of the different versions and controls against data set size, with a speedup for the transformed code of up to 1.5 over

the Fortran control, and 1.43 over the ATLAS BLAS version.

The benefit of the (cross-component) optimisations is described in terms of maximising instruction level parallelism by avoiding the memory bandwidth bottleneck. There is also a brief discussion on the trade-offs between latency and prefetching that the transformations explore. In keeping with the emphasis on cross-component optimisation, the related work section deals mostly with alternative approaches to embedding domain specific components into a host language and then optimising the result, rather than traditional optimisations or LDG transformations. Examples discussed include expression templates, library annotations and the development of customised parsers for what are effectively domain specific extensions to a language, all of which rely on the specification of domain specific optimisation rules by the developer of the component library.

The suggestions for extension of the work include conducting experiments on more complex operators such as the red-black preconditioned Wilson-Dirac operator, incorporating other solvers, and using iterative optimisation to attack possible latency problems in individual loops after fusion and contraction. In this thesis the work has been extended to include the unpreconditioned Wilson-Dirac operator, the loop transformations have been fully formalised, and iterative optimisation has been applied to the task of fusion/contraction itself – further application of iterative optimisation to individual loops resulting from collective loop fusion is an important part of the future work given in Chapter 11.

Bibliography

[1] Haskell website. http://www.haskell.org/.

[2] MLRISC website. http://www.cs.nyu.edu/leunga/www/MLRISC/Doc/html/.

[3] OCaml website. http://caml.inria.fr/ocaml/index.en.html.

[4] SML/NJ website. http://www.smlnj.org.

[5] SPEC website. http://www.spec.org.

[6] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, pages 231–239, New York, NY, USA, 2004. ACM Press.

[7] T. Ashby, D. Galletly, B. Joó, A.D. Kennedy, and G. Lacagnina. The Paraldor project. In *Nuclear Physics B Proceedings Supplements*, volume 119, pages 1006–1008. Elsevier, May 2003.

[8] T.J. Ashby, A.D. Kennedy, and M.F.P. O'Boyle. A modular iterative solver package in a categorical language. In Press: Proceedings of the Third International Workshop on Numerical Analysis and Lattice QCD, November 2003.

[9] T.J. Ashby, A.D. Kennedy, and M.F.P. O'Boyle. Cross component optimisation in a high level category-based language. In *Euro-Par 2004 Parallel Processing: 10th International Euro-Par Conference*, volume 3149 of *LNCS*, page 654, Pisa, Italy, August 2004. Springer-Verlag GmbH.

[10] J. Michael Ashley. The effectiveness of flow analysis for inlining. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 99–111, New York, NY, USA, 1997. ACM Press.

[11] J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems*, 20(4):845–868, 1998.

[12] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[13] B.Franke, M. O'Boyle, J.Thomson, and G.Fursin. Probabilistic source-level optimisation of embedded programs. In *To Appear in: Proceedings of the ACM SIGPLAN/SIGBED 2005 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2005)*, Chicago, Illinois, June 2005. ACM.

[14] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.

[15] William Blume, Rudolf Eigenmann, Jay Hoeflinger, David Padua, Paul Petersen, Lawrence Rauchwerger, and Peng Tu. Automatic detection of parallelism: A grand challenge for high-performance computing. *IEEE Parallel Distrib. Technol.*, 2(3):37–47, 1994.

[16] F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proceedings of the Workshop on Profile Directed Feedback-Compilation, PACT'98*, 1998.

[17] P. A. Boyle, C. Jung, and T. Wettig. The QCDOC supercomputer: Hardware, software, and performance. *ECONF*, C0303241:THIT003, 2003.

[18] Manuel Bronstein. `libaldor` *user guide and reference manual, version 1.0.2.* INRIA, Sophia-Antipolis, April 2004.

[19] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 56–71, London, UK, 2000. Springer-Verlag.

[20] Manuel M. T. Chakravarty and Gabriele Keller. Functional array fusion. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 205–216, New York, NY, USA, 2001. ACM Press.

[21] Tony F. Chan, Lisette de Pillis, and Henk van der Vorst. Transpose-free formulations of Lanczos-type methods for nonsymmetric linear systems. *Numerical Algorithms*, 17(1–2):51–66, 1998.

[22] Stéphane Chauveau and François Bodin. Menhir: An environment for high performance MATLAB. In *LCR '98: Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 27–40, London, UK, 1998. Springer-Verlag.

[23] Dong Chen, Ping Chen, Norman H. Christ, Robert G. Edwards, George Fleming, Alan Gara, Sten Hansen, Chulwoo Jung, Adrian Kahler, Stephen Kasow, Anthony D. Kennedy, Greg Kilcup, Yu Bing Luo, Catalin Malureanu, Robert D. Mawhinney, John Parsons, Jim Sexton, ChengZhong Sui, and Pavlos Vranas. QCDSP: a teraflop scale massively parallel supercomputer. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–17, New York, NY, USA, 1997. ACM Press.

[24] M. Clarkson and V. Vaish. Array optimizations in OCaml. Technical report, Cornell University, 2001. http://www.cs.cornell.edu/Courses/cs612/2001SP/projects/ocaml-arrays/OCaml.pdf.

[25] Régis Cridlig. An optimizing ML to C compiler. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 28–36, San Francisco, California, 1992.

[26] Alain Darte. On the complexity of loop fusion. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, page 149, Washington, DC, USA, 1999. IEEE Computer Society.

[27] Alain Darte and Guillaume Huard. New results on array contraction. In *13th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2002)*, pages 359–370, San Jose, CA, USA, July 2002. IEEE Computer Society.

[28] L. De Rose. *Compiler Techniques for MATLAB*. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA, 1996.

[29] Chen Ding and Ken Kennedy. The memory bandwidth bottleneck and its amelioration by a compiler. In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, page 181, Washington, DC, USA, 2000. IEEE Computer Society.

[30] R. Edwards, A.D. Kennedy, and C. Vohwinkel. SZIN: An object oriented macro-based system for lattice field theory. http://www.jlab.org/~edwards/szin/macros_v2.ps, 1996.

[31] Oliver Ernst. Lanczos-based iterative solution methods. http://www.mathe.tu-freiberg.de/~ernst/Lehre/AKN01/Ernst/lanczos01.pdf, 2001. Lecture Slides.

[32] Antoine Fraboulet, Karen Kodary, and Anne Mignotte. Loop fusion for memory space optimization. In *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, pages 95–100, New York, NY, USA, 2001. ACM Press.

[33] Roland Freund and Noel Nachtigal. QMRpack. http://www.netlib.org/linalg/qmr/.

[34] A. Frommer. Personal communication.

[35] A. Frommer and B.Medeke. Exploiting structure in Krylov subspace methods for the Wilson fermion matrix. In A. Sydow, editor, *The proceedings of the 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, pages 485–490. Wissenschaft & Technik Verlag, Berlin, 1997.

[36] G.G. Fursin, M.F.P. O'Boyle, and P.M.W. Knijnenburg. Evaluating iterative compilation. In *Proc. Languages and Compilers for Parallel Computers (LCPC)*, pages 305–315, 2002.

[37] Guang R. Gao, R. Olsen, Vivek Sarkar, and Radhika Thekkath. Collective loop fusion for array contraction. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 281–295. Springer-Verlag, 1992.

[38] J. Gaudiot, W. Bohm, T. DeBoni, J. Feo, and P. Miller. The SISAL model of functional programming and its implementation. In *Proceedings of the 2nd AIZU International Symposium on Parallel Algorithms / Architecture Synthesis (pAs '97)*, Aizu-Wakamatsu, Japan, March 1997.

[39] S.V. Gheorghita, H. Corporaal, and T. Basten. Iterative compilation for energy reduction. *Journal of Embedded Computing*, To appear in 2005.

[40] Gene H. Golub and Charles F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[41] Anne Greenbaum. *Iterative methods for solving linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[42] Martin H. Gutknecht. Lanczos-type solvers for non-hermitian linear systems. http://www.sam.math.ethz.ch/~mhg/talks/ltsshort.ps, 1999. Lecture Slides.

[43] Martin H. Gutknecht. On Lanczos-type methods for Wilson fermions. In A. Frommer and et al., editors, *Numerical Challenges in Lattice Quantum Chromodynamics: joint interdisciplinary workshop of John von Neumann Institute for Computing, Jülich, and Institute of Applied Computer Science, Wuppertal University*, volume 15 of *LNCSE*, pages 48–65. Springer, August 1999.

[44] Samuel Z. Guyer and Calvin Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[45] C. Hall, S. L. Peyton Jones, and P. M. Sansom. Unboxing using specialisation. In K. Hammond, D. N. Turner, and P. M. Sansom, editors, *Functional Programming*, pages 96–110. Springer, Berlin, Heidelberg, 1994.

[46] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*. ACM Press, June 2002. Berlin, Germany.

[47] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[48] Richard D. Jenks and Robert Sutor. *AXIOM: The scientific computation system.* Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[49] Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. `C--`: A portable assembly language that supports garbage collection. In *PPDP'99: Proceedings of the International Conference on Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 1–28, Paris, France, 1999. Springer.

[50] Gabriele Keller and Manuel M. T. Chakravarty. On the distributed implementation of aggregate data-structures by program transformation. In *Parallel and Distributed Processing, Fourth International Workshop on Highlevel Parallel Programming Models and Supportive Environments (HIPS99)*, number 1586 in Lecture Notes in Computer Science, pages 108–122. Springer-Verlag, 1999.

[51] K. Kennedy and K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report Techreport TR93-208, Rice University Dept. of Computer Science, 1993.

[52] Ken Kennedy. Fast greedy weighted fusion. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 131–140, New York, NY, USA, 2000. ACM Press.

[53] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, page 237, Washington, DC, USA, 2000. IEEE Computer Society.

[54] Xavier Leroy. Unboxed objects and polymorphic typing. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 177–188, New York, NY, USA, 1992. ACM Press.

[55] Xavier Leroy. The effectiveness of type-based unboxing. In *Workshop on Types in Compilation '97*. Technical report BCCS-97-03, Boston College, Computer Science Department, June 1997.

[56] E. Christopher Lewis, Calvin Lin, and Lawrence Snyder. The implementation and evaluation of fusion and contraction in array languages. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 50–59, New York, NY, USA, 1998. ACM Press.

[57] Amy W. Lim, Shih-Wei Liao, and Monica S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 103–112, New York, NY, USA, 2001. ACM Press.

[58] Bret Marsolf. *Techniques for the Interactive Development of Numerical Linear Algebra Libraries for Scientific Computation.* Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1997.

[59] C. McClendon. Optimized lattice QCD kernels for a Pentium 4 cluster. http://www.jlab.org/~edwards/qcdapi/reports/dslash_p4.pdf, Jlab preprint, JLAB-THY-01-29. Adapted code provided by Bálint Joó, personal communication.

[60] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, 1996.

[61] Kathryn S. McKinley and Olivier Temam. Quantifying loop nest locality using SPEC'95 and the Perfect benchmarks. *ACM Trans. Comput. Syst.*, 17(4):288–336, 1999.

[62] Nimrod Megiddo and Vivek Sarkar. Optimal weighted loop fusion for parallel programs. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 282–291, New York, NY, USA, 1997. ACM Press.

[63] Vijay Menon and Keshav Pingali. A case for source-level transformations in MATLAB. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 53–65, New York, NY, USA, 1999. ACM Press.

[64] J. Modersitzki. Conjugate gradient type methods for solving symmetric, indefinite linear systems. Technical Report 868, Dept. of Math., University of Utrecht, 1994.

[65] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[66] A. P. Nisbet. GAPS: Iterative feedback directed parallelisation using genetic algorithms. In *Proceedings of Workshop on Profile and Feedback-Directed Compilation at PACT98, Paris, France.* Springer Verlag, 1998.

[67] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM Journal of Numerical Analysis*, 12:617–629, 1975.

[68] David Parello, Olivier Temam, and Jean-Marie Verdun. On increasing architecture awareness in program optimizations to bridge the gap between peak and sustained processor performance: matrix-multiply revisited. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[69] Geoffrey Roeder Pike. *Reordering and storage optimizations for scientific programs.* PhD thesis, Computer Science Division, University of California, Berkley, 2002. Chair-Paul N. Hilfinger.

[70] William Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, 1992.

[71] Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3D scientific computations. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 32, Washington, DC, USA, 2000. IEEE Computer Society.

[72] M. Rozloznik and R. Weiss. On the stable implementation of the generalized minimal error method. *Journal of Computational and Applied Mathematics*, 98:49–62, 1998.

[73] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, 1996.

[74] Amr Sabry and Matthias Felleisen. Is continuation-passing useful for data flow analysis? In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 1994. ACM Press.

[75] Sven-Bodo Scholz. Single assignment C: efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[76] Manuel Serrano. Control flow analysis: a functional languages compilation paradigm. In *SAC '95: Proceedings of the 1995 ACM symposium on Applied computing*, pages 118–122, New York, NY, USA, 1995. ACM Press.

[77] Manuel Serrano and Marc Feeley. Storage use analysis and its applications. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 50–61, New York, NY, USA, 1996. ACM Press.

[78] O. Shivers. Control flow analysis in scheme. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 164–174, New York, NY, USA, 1988. ACM Press.

[79] Sharad Singhai and Kathryn S. McKinley. A parameterized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, 40(6):340–355, 1997.

[80] Yonghong Song, Rong Xu, Cheng Wang, and Zhiyuan Li. Data locality enhancement by memory reduction. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 50–64, New York, NY, USA, 2001. ACM Press.

[81] P. Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM Journal on Scientific Statistical Computing*, 10:36–52, 1989.

[82] Z. Sroczynski. `DSlash` in C – personal communication.

[83] Z. Sroczynski. HMC for ALiCE. http://www.theorie.physik.uni-wuppertal.de/Computerlabor/Alice/akmt.phtml, 2002.

[84] David Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML.* PhD thesis, Carnegie Mellon University, 1996. Available as CMU CS technical report 97-108.

[85] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: compiling standard ML to C. *ACM Lett. Program. Lang. Syst.*, 1(2):161–177, 1992.

[86] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 281–293, New York, NY, USA, 2000. ACM Press.

[87] Andrew P. Tolmach and Dino Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, 1998.

[88] P. van der Mark, E. Rohou, F. Bodin, Z. Chamski, and C. Eisenbeis. Using iterative compilation for managing software pipeline-unrolling tradeoffs. In *Presented at the 4th workshop on Software and Compilers for Embedded Systems*, St Goar, Germany, Sept 1999.

[89] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[90] S. Verdoolaege, M. Bruynooghe, G. Janssens, and F. Catthoor. Multi-dimensional incremental loop fusion for data locality. In *Proc. of the IEEE Conf. on Application-Specific Systems, Architectures, and Processors*, pages 14–24, June 2003.

[91] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

[92] Philip Wadler. A taste of linear logic. In *Mathematical Foundations of Computing Science*, number 711 in LNCS. Springer–Verlag, August 1993. Gdansk.

[93] S.M. Watt. *Aldor Users Guide.* http://www.aldor.org, 2nd edition, 2002.

[94] S.M. Watt, P.A. Broadbery, P. Iglio, S.C. Morrison, and J.M. Steinbach. FOAM: A first order abstract machine, v 0.35. Research Report RC 19528, IBM, 1994.

[95] Stephen M. Watt, Peter A. Broadbery, Samuel S. Dooley, Pietro Iglio, Scott C. Morrison, Jonathan M. Steinbach, and Robert S. Sutor. A first report on the A# compiler. In *ISSAC '94: Proceedings of the international symposium on Symbolic and algebraic computation*, pages 25–31, New York, NY, USA, 1994. ACM Press.

[96] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[97] Paul R. Wilson. Uniprocessor garbage collection techniques. In *IWMM '92: Proceedings of the International Workshop on Memory Management*, pages 1–42, London, UK, 1992. Springer-Verlag.

[98] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *IWMM '95: Proceedings of the International Workshop on Memory Management*, pages 1–116, London, UK, 1995. Springer-Verlag.

[99] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih Liao, Chau Tseng, Mary Hall, Monica Lam,

and John Hennessy. The SUIF compiler system: a parallelizing and optimizing research compiler. Technical report, Stanford University, Stanford, CA, USA, 1994.

[100] Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, 20(1):166–207, 1998.

[101] Thorsten H.G. Zörner. Numerical analysis and functional programming. In Davie and Clark, editors, *Proceedings of the 10th international workshop on the implementation of functional languages*, number 1595 in Lecture Notes in Computer Science, pages 27–48. Springer–Verlag, 1998.