Datapath and Memory Co-optimization for FPGA-based Computation

by
Kieron Turkington

A Thesis submitted in fulfilment of requirements for the degree of
Doctor of Philosophy of Imperial College London and
Diploma of Imperial College

Circuits and Systems Research Group
Department of Electrical and Electronic Engineering
Imperial College London
2010

Abstract

With the large resource densities available on modern FPGAs it is often the available memory bandwidth that limits the parallelism (and therefore performance) that can be achieved. For this reason the focus of this thesis is the development of an integrated scheduling and memory optimisation methodology to allow high levels of parallelism to be exploited in FPGA based designs.

A manual translation from C to hardware is first investigated as a case study, exposing a number of potential optimisation techniques that have not been exploited in existing work. An existing outer loop pipelining approach, originally developed for VLIW processors, is extended and adapted for application to FPGAs. The outer loop pipelining methodology is first developed to use a fixed memory subsystem design and then extended to automate the optimisation of the memory subsystem. This approach allocates arrays to physical memories and selects the set of data reuse structures to implement to match the available and required memory bandwidths as the pipelining search progresses. The final extension to this work is to include the partitioning of data from a single array across multiple physical memories, increasing the number of memory ports through which data my be accessed. The facility for loop unrolling is also added to increase the potential for parallelism and exploit the additional bandwidth that partitioning can provide.

We describe our approach based on formal methodologies and present the results achieved when these methods are applied to a number of benchmarks. These results show the advantages of both extending pipelining to levels above the innermost loop and the co-optimisation of the datapath and memory subsystem.

Acknowledgment

I would first like to thank Dr George Constantinides, Prof. Konstantinos Masselos and Prof. Peter Cheung for their supervision and support during this thesis, and the Engineering and Physical Sciences Research Council (U.K.) for funding this work. I am most grateful to George who bravely took me under his supervision following Konstantinos' departure. George is incredibly intelligent and perceptive, able to find the holes in my thinking where they existed, and providing reassurance on the occasions when he did not find any. His push for me to formalise my methods was vital in turning some vague ideas into something more coherent. Konstantinos was my primary supervisor throughout the first six months of this work and I am grateful for the ideas and direction be provided early on to get things started, as well as for the continued support he provided beyond this. I would also like to thank Peter, not only for the useful comments regarding papers and my work in general, but also for suggesting that I might consider a PhD.

I would like to thank all the members of the Circuits and Systems research group for enhancing my PhD experience. I am especially grateful to Dr Pete Sedcole, Dr Alastair Smith, Dr Christos Bouganis and Dr Altaf Gaffar for the numerous bits of help they provided, and Dr Qiang Liu for the useful chats on data reuse and behavioural synthesis.

On a more personal note, I would like to thank my parents for all their support and encouragement throughout my academic career, and Emma for putting up with me during my write up. Also worthy of a mention are my my long suffering flatmates Chris and Rachael, and Helen and Oli whose company during numerous lunch and coffee breaks kept me sane. One final final thank you must go to my late grandfather. His long career as a control engineer inspired my own interest in engineering.

Contents

Abstract **2**

Acknowledgment **3**

Contents **4**

List of Figures **10**

List of Tables **13**

Chapter 1. Introduction **15**

1.1 Motivation 15

1.2 Objectives 17

1.3 Overview 17

1.4 Statement of Originality 17

1.5 Publications 18

Chapter 2. Background **20**

2.1 Dependences 21

2.2 Loop Transformations 24

2.3 Scheduling Techniques 28

2.3.1 Loop Pipelining 29

2.3.2 Pipelining Above the Innermost Loop 36
2.3.3 Scheduling Optimisations 38
2.4 Memory Optimisation 42
2.4.1 Loop and Data Restructuring 42
2.4.2 Data Reuse Exploitation 45
2.4.3 Data to Memory Allocation 51
2.5 Research Hardware Compilers 58
2.6 Commercial Design Tools 62
2.7 Optimisation Methods 67
2.7.1 Greedy Algorithms 67
2.7.2 Simulated Annealing 68
2.7.3 Linear Programming 68
2.8 Summary 70

Chapter 3. LINPACK 1000: A Case Study **71**

3.1 Introduction 71
3.2 LINPACK 1000 72
3.3 Target Platform & Partitioning 75
3.4 Translation and Optimisation 77
3.4.1 Data Reuse Exploration 78
3.4.2 Loop Pipelining 80
3.4.3 Loop Level Parallelism 82
3.4.4 Memory Access Details 85
3.4.5 Low Level Optimisation 88
3.5 Results 90

3.6 Summary 94

Chapter 4. Pipelining Above the Innermost Loop 95

4.1 Introduction 95

4.2 Background 97

4.3 Extending the Solution Space 98

4.4 FPGA Resource Constraints 99

4.5 Imperfect Nesting 101

4.6 Scheduling 103

4.6.1 Searching the Solution Space 104

4.6.2 Modulo Scheduling 108

4.6.3 Calculating the Final Initiation Interval 113

4.7 Hardware Implementation 116

4.8 Results 125

4.8.1 Scheduling Results 126

4.8.2 FPGA Implementation Results 129

4.8.3 Comparison with Loop Interchange and Inner Loop Pipelining . . . 132

4.9 Summary 134

Chapter 5. Memory Optimisation 135

5.1 Introduction 135

5.2 Background 136

5.3 Array to Memory Placement 138

5.3.1 Problem Formulation 139

5.3.2 Variable Reductions 140

5.3.3 Bindings 142

5.4 Data Reuse 142
5.4.1 FIFOs 142
5.4.2 Buffers 144
5.4.3 Pre-optimisation of the Reuse Options 145
5.5 Integrating Data Reuse with Array to Memory Placement 147
5.6 Cost Functions 151
5.6.1 Minimising T 152
5.6.2 Minimising S and II 152
5.6.3 Minimising Z_p 153
5.6.4 Minimising S_{tot} 154
5.7 Integration with scheduling 156
5.8 Results 158
5.9 Summary 163

Chapter 6. Array Partitioning and Loop Unrolling **164**

6.1 Introduction 164
6.2 Extensions to Increase Parallelism 165
6.2.1 Top Level of the Scheduling Search 165
6.2.2 Searching the Unroll Options for Multiple Levels 168
6.2.3 Searching the Unroll Options for a Single Level 171
6.3 Scheduling For Fixed Unroll Values 173
6.4 Modulo Scheduling with Loop Unrolling 175
6.5 Array Partitioning 182
6.5.1 Partitioning to Minimise T 184

6.5.2 Partitioning to Maximise Loop Unroll . . . 185
6.5.3 Partitioning for Unroll at Multiple Loop Levels . . . 191
6.6 Data Reuse Through FIFOs . . . 193
6.7 Memory Optimisation . . . 196
6.7.1 Array to Memory Placement . . . 196
6.7.2 Including Data Reuse Through Buffers . . . 203
6.7.3 Including Data Reuse Through FIFOs . . . 206
6.7.4 Cost Functions . . . 211
6.7.5 Pruning the Partitioning Options . . . 217
6.8 Results . . . 223
6.9 Summary . . . 228

Chapter 7. Conclusion 229

7.1 Summary . . . 229
7.2 Future Work . . . 231
7.2.1 Clock Frequency Effects . . . 231
7.2.2 Extensions for SDRAM . . . 232
7.2.3 Extensions for Multiple Nested Loops . . . 233
7.2.4 Investigation of Heuristics . . . 234
7.2.5 Minimising Resource Usage or Power Consumption . . . 234

Appendix A. Schedule Length Derivation For Outer Loop Pipelining 236

Appendix B. Derivations 240

B.1 Derivation 1 . . . 240
B.2 Derivation 2 . . . 241

Appendix C. Original and Transformed C Code for LU Decomposition and Minimum Residual Algorithms **243**

Appendix D. C Code for Benchmark Algorithms **248**

Bibliography **252**

List of Figures

2.1 A sample data flow graph. 23

2.2 An example of loop interchange. 25

2.3 An example of loop tiling. 27

2.4 An example of loop skewing. 28

2.5 An example of loop pipelining. 30

2.6 An example demonstrating the resource efficiency of loop pipelining. 31

2.7 A sample loop and corresponding dependence graph demonstrating cyclic paths. 33

2.8 Scheduling the loop in Figure 2.7(a) to meet resource and dependence constraints. 36

2.9 An example of dovetail pipelining. 37

2.10 An example of speculation and reverse speculation 40

2.11 An example of branch balancing. 41

2.12 An example of loop merging to reduce memory accesses. 43

2.13 Improving data locality. 44

2.14 Typical memory hierarchies. 45

2.15 Inserting explicit data reuse. 49

2.16 Using scalar replacement to reduce latency. 56

2.17 Array Partitioning to maximise performance. 57

3.1 C Code for LINPACK 1000. 74

3.2 Target Platform for LINPACK. 76

3.3 C Code for *dgefa*. 79

3.4 C Code for *idamax*, *dscal* and *daxpy*. 80

3.5 Overview of the dataflow for *dgefa* after data reuse. 81

3.6 Loop level scheduling for *dgefa*. 83

3.7 Simplified overview of the dataflow for *dgefa*, using N outer loop copies, after exploiting loop level parallelism. 86

3.8 Overview of the dataflow for *dgefa* including the external memory controller. 89

4.1 Scheduling imperfectly nested operations. 102

4.2 A simple controller design for a loop pipelined at the innermost level. . . . 117

4.3 A section of an example schedule for a double nested loop pipelined at the outermost level. 118

4.4 A circuit to generate the loop index for each stage at one level in the loop. . 119

4.5 The 'Incrementer' blocks. 120

4.6 The logic to control the enable signal for each pipeline stage. 122

4.7 The 'Enable/Disable' block. 122

4.8 The logic to control the enable signals for an imperfectly nested loop. . . . 123

5.1 An example loop for which a buffer may automatically be inferred. 145

6.1 Dependence graph simplification. 177

6.2 Pictorial representation of the schedule for a double nested loop which has been unrolled by a factor of 3 at the innermost level and pipelined at the outermost level. 179

6.3 Merging 3 unrolled pipeline copies. 181

6.4 Pictorial representation of the overall schedule for the loop in Figure 6.2 using the merged pipeline in Figure 6.3 182

6.5 Example of buffer inference for a matrix-matrix multiply kernel. 204

C.1 Original C code for the LU decomposition kernel with irregular loop nesting. 244

C.2 Transformed C code for the LU decomposition kernel with only regular nesting. 245

C.3 Original C code for the Minimum Residual algorithm with irregular loop nesting. 246

C.4 Transformed C code for the Minimum Residual algorithm with only regular nesting. 247

D.1 C code for the image edge detection kernel. 248

D.2 C code for the 2D median filter kernel. 249

D.3 C code for the motion estimation kernel. 249

D.4 C code for the matrix-matrix multiplication kernel. 250

D.5 C code for the Successive Over Relaxation kernel. 250

D.6 C code for the 2D hydrodynamics kernel. 250

D.7 C code for the Complex FFT kernel. 251

List of Tables

3.1 Summary of results for LINPACK implementation using Altera Stratix II devices 91

4.1 Summary of the notation used to describe the module scheduling ILP formulation. 109

4.2 Scheduling results for the edge detection, motion estimation, fast Fourier transform and median filter kernels using outer loop pipelining. 126

4.3 Scheduling results for the matrix-matrix multiply , hydrodynamics, successive over relaxation, MINRES and LU decomposition kernels using outer loop pipelining. 127

4.4 Hardware implementation results for the test loops. 130

5.1 Summary of the set notation used to describe the memory optimisation ILP formulation. 147

5.2 Summary of the constants and variables used in the memory optimisation ILP formulation. 148

5.3 Combined pipelining and memory optimisation results for seven test loops. 159

5.4 Combined pipelining and memory optimisation results for seven test loops. 160

5.5 Implementation results for the edge detection and matrix multiply kernels. . 161

6.1 Resulting address functions and partition accesses patterns for the sample loop. 190

6.2 Summary of the notation used in the partitioning and array to memory allocation ILP formulation. 198

6.3 Summary of the notation used to include buffer selection in the ILP formulation. 204

6.4 Memory usage for seven array partitioning schemes. 220

6.5 Scheduling results for seven test loops using the combined pipelining, unrolling and memory optimisation methodology. 224

6.6 Scheduling results for seven test loops using the combined pipelining, unrolling and memory optimisation methodology. 227

Chapter 1

Introduction

1.1 Motivation

As integrated circuits have grown according to Moore's Law, so too have the capabilities of devices offering programmable logic. The first Programmable Logic Devices (PLDs) offered functionality up to hundreds of gates [1, 2]. The same vendors responsible for these PLDs now offer Field Programmable Gate Arrays (FPGAs) with logic equivalent to millions of gates [3, 4]. They also feature a range of dedicated heterogeneous blocks such as embedded memories, embedded multipliers and even hardwired microprocessors [3, 4] so entire digital systems can now be implemented on a single FPGA. The increase in the potential performance of a single device has gone hand in hand with an increase in the complexity of a typical digital system. As the processing power of devices has grown, ever more ambitious algorithms for signal, video and image processing, communication and scientific modeling have been developed to take advantage of them. This means that, while it is possible to implement entire systems on a single device, the design effort required to produce such designs has grown dramatically [5].

Most design for FPGA based systems is still done at the Register Transfer Level (RTL) using hardware design languages such as VHDL and Verilog [6]. RTL design requires all of the low level details of a system to be specified explicitly by the designer, down to the clock cycle by clock cycle behaviour of each signal. Designers will typically

start with a working description of the algorithm, written in software languages such as C, C++ or MATLAB. High level implementation decisions are then made, such as the interface scheme to use or how the code may be partitioned across microprocessors and custom logic. The algorithm is then implemented manually in RTL VHDL or Verilog for the given high level decisions. This process is complicated (and often error prone) and accounts for most of the design time. If designers want to compare different high level designs the RTL code must be re-written and tested for each high level change. This can limit the exploration of the design space, potentially forcing designers to fix high level design decisions early, testing only a small number of partitions or architectures. The final system resulting from this process may then be sub-optimal. One possible approach to lower design times and increase productivity is high level or *behavioural synthesis* [7]. In behavioural synthesis a hardware description is generated automatically from a high level, untimed algorithm description, such as a C program. Automating this process removes the need for error prone and time consuming manual conversion, allowing designers to focus on the high level decisions for the design. A number of commercial and academic tools have already been developed along these lines, including Mentor Graphics Catapult C [8] and the ROCC compiler [9].

Behavioural synthesis is a complicated task as a software-like description must be converted into an efficient hardware implementation[1]. This involves the scheduling of operations, the allocation of physical resources to computations, interfacing with external components and the specification of a memory subsystem. An overview of the steps required can be found in [7, 10]. The partitioning of code between hardware and software may also be necessary if the target platform includes one or more microprocessors. Of key importance in behavioural synthesis for FPGAs is maximising the degree of parallelism in the final design since it is through the exploitation of parallel execution that FPGAs allow algorithms to be accelerated. Often it is the available bandwidth for accessing the memory subsystem which becomes the bottleneck in FPGA based systems. The optimisation of the memory structure and the maximisation of parallel execution within the optimised memory constraints are therefore important in achieving maximum performance.

[1] One that ideally uses the minimum resources and/or power for a given speed.

1.2 Objectives

The objective of this thesis is to investigate methods for converting C-like software descriptions into FPGA hardware, with the goal of minimising execution time for a given algorithm on a specified target platform. The specific focus is the integration of scheduling and memory optimisation into a combined methodology so that the memory bandwidth requirements of the datapath match the bandwidth available from the memory subsystem, and vice versa. Loop pipelining and loop unrolling are targeted to maximise the parallelism for a nested loop.

1.3 Overview

The structure of this thesis is as follows. Chapter 2 reviews existing literature related to behavioural synthesis, the scheduling of operations, memory design methodologies and general optimisation techniques. Chapter 3 presents a case study examining the the methods required to convert a software description of an algorithm into efficient FPGA hardware and identifies potential optimisation techniques that have not yet been targeted in existing work. Chapter 4 explores the extension of an existing approach for pipelining nested loops on VLIW machines at levels above the innermost loop and applies it to FPGAs. The methods in Chapter 4 schedule a nested loop with a fixed memory subsystem as an input to the process. Chapter 5 extends the methodology to optimise the memory system as scheduling progresses, allocating array data to physical memories and selecting structures to exploit data reuse. The work in Chapter 6 further extends upon this by introducing methods to partition arrays across multiple physical memories and unroll multiple levels in the loop where possible. Chapter 7 summarises the conclusions of this work.

1.4 Statement of Originality

The original contributions of this work relate to methods for the co-optimisation of FPGA datapaths and memory subsystems. The three components listed below fit together to form

a methodology to produce a matching datapath and memory subsystem to implement a single nested loop in a near-minimal execution time.

- Extensions to an existing loop pipelining approach [11, 12], originally developed for VLIW processors [13], that can pipeline loop iterations at an arbitrary loop level. The extensions include relaxations to the constraints on the pipeline initiation interval to better suit FPGA implementations, a search scheme to find the shortest schedule, an ILP formulation for modulo scheduling and a design for a generalised pipeline controller.

- An ILP formulation which combines the selection of which data reuse options to implement with the allocation of array data to physical memories. A number of cost functions are provided, allowing the memory optimisation function to be called at various points in the pipelining search to modify the memory subsystem to match the current point in the search [11, 14].

- A method to partition array data across multiple physical memories so that the unroll factor for a given loop level can be maximised. This is combined with an extended memory optimisation ILP formualtion to allocate the partitioned data to physical memories and an extended scheduling algorithm to search the possible values of loop unroll at each level, along with the pipelining possibilities.

1.5 Publications

The following publications have been written during the course of this work:

- K. Turkington, K. Masselos, G. A. Constantinides and P. Leong, "FPGA Based Acceleration of the Linpack Benchmark: A High Level Code Transformation Approach," *Proceedings of the 2006 IEEE International Conference on Field Programmable Logic and Appplications*, pages 375 – 381 This publication relates to the work presented in Chapter 3 of this thesis.

- K. Turkington, G. A. Constantinides, K. Masselos and P. Y. K. Cheung, "Pipeline Exploration for Reconfigurable Targets" *Proceedings of the 2007 Workshop on Application Specific Processors*, pages 11 – 18 This publication relates to the work presented in Chapter 4 of this thesis.

- K. Turkington, G. A. Constantinides, K. Masselos and P. Y. K. Cheung,"Outer Loop Pipelining for Application Specific Datapaths in FPGAs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Volume 16, Issue 10, pages 1268 – 1280, 2008. This publication relates to the work presented in Chapter 4 of this thesis.

- K. Turkington, G. A. Constantinides, K. Masselos and P. Y. K. Cheung, "Co-optimisation of Datapath and Memory in Outer Loop Pipelining," *Proceedings of the 2008 IEEE International Conference on Field-Programmable Technology*, pages 1 – 8. This publication relates to the work presented in Chapter 5 of this thesis.

Chapter 2

Background

High level synthesis is a complex task and multiple issues must be addressed when converting a behavioural description of an algorithm into a hardware implementation. This chapter presents a review of existing methods that are relevant to behavioural synthesis, organised into seven sections. Section 2.1 gives a brief overview of *dependences* and how they affect the order in which operations in an algorithm may be executed. Knowledge of the dependences present in an algorithm is vital during the scheduling of algorithm's operations and so dependence checking is an important issue in high level synthesis. In Section 2.2 various loop transformations that are typically used by compilers (both in the software and hardware domains) are described. Loop transformations are commonly used to increase the potential for parallel execution in an algorithm and to improve the performance of the memory subsystem, which are both important factors for FPGA based designs. Section 2.3 reviews existing methods for scheduling operations in algorithms and Section 2.4 reviews existing memory optimisation techniques typically used in the generation of custom hardware design. Sections 2.5 and 2.6 detail the techniques and optimisations utilised in existing research and commercial behavioural compilers respectively, with the goal of defining which aspects of high level synthesis are well advanced and which aspects can yet be improved. Section 2.7 provides a brief overview of some optimisation methods that have been previously been used to solve problems in high level synthesis, and which are used in the methods developed in this thesis.

2.1 Dependences

When algorithms are described in imperative languages such as C or Java there is an implicit ordering to the operations described. The designer writes the code with the assumption that the operations will execute sequentially in the order they appear in the code. This sequential ordering may need to be preserved for some instructions during compilation, but the freedom may exist to reorder (or parallelise) the execution of other instructions. The execution order of two operations is constrained if altering the order would change the output produced by the algorithm. In such cases the two operations are said to have a dependence between them. In fact a dependence will exist between any two operations that read or write the same element of data, though not all of these dependences constrain the ordering of operations [15]. Knowledge of the dependences that exist in an algorithm is required when scheduling the execution of operations to ensure the correct results. Certain types of dependence between operations can also indicate the reuse of data (multiple reads to the same data element or a write followed by a read) and so detailed knowledge of the dependences present in an algorithm is often required when attempting to exploit this reuse to reduce the required memory bandwidth.

Existing work in this area has identified four types of dependence between operations in an algorithm: *flow dependence*, *anti dependence*, *output dependence* and *input dependence* [15]. The properties of each dependence are listed below.

- **Flow Dependence:** Operations A and B are said to be linked by a flow dependence if operation A is executed before operation B in the original specification and a value written to by operation A is read by operation B. Flow dependences must always be honoured in the final implementation of an algorithm, *i.e.* operation A must always execute before operation B.

- **Anti Dependence:** Operations A and B are linked by an anti dependence if operation A is executed before operation B in the original specification and operation A reads a variable that operation B later writes to. An anti dependence implies that operation A must execute before operation B, but anti dependences may be

removed from an algorithm by conversion to *static single assignment* form [16] (see Section 2.3.3 for further details).

- **Output Dependence:** Operations A and B are linked by an output dependence if operation A writes to a variable that operation B later writes to. If there are operations that read from the variable between A and B then operation A must execute before B. However, in this case the read operation will produce flow and anti-dependences that will ensure execution order is maintained and the output dependence can be ignored. If there are no reads between two write operations then the data produced in the first write operation is never used. Hence the first write operation can be removed from the algorithm altogether[1]. Output dependences can therefore be useful in dead code elimination [17].

- **Input Dependence:** Operations A and B are linked by an input dependence if operation A reads the same variable that operation B later reads. If all data is read from memories then input dependences place no restriction on the order of operations[2]. They can, however, be used to locate opportunities for data reuse (see Section 2.4.2).

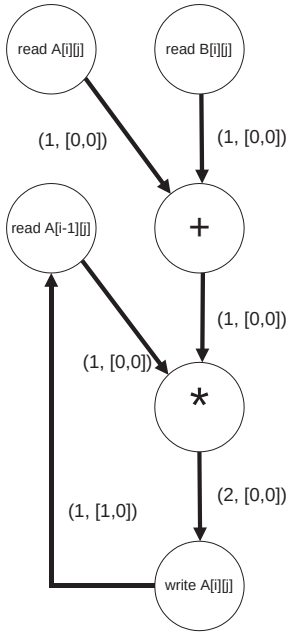
Each dependence in an algorithm will have associated with it a source operation (operation A in the previous descriptions), a sink operation (operation B in the previous descriptions) and a type (flow, anti, output or input). Every dependence will also have an associated latency (the latency of the source operation) and dependences between operations in loops/loop nests each have an associated *iteration vector* [18]. The iteration vector linking two dependent operations denotes the number of loop iterations at each level in the loop nest between the operations. Iteration vectors for a nested loop with N levels will have N elements and are of the form $[i_N, i_{N-1}, ..., i_2, i_1]$, where each i_n is an integer representing the number of iterations separating two operations at level n in the loop nest (N is the outermost loop).

[1] These simplifications are only true if there are no other components external to the algorithm being considered that can read from the algorithm's variables.

[2] This would not be the case for a streaming application as the order of data input would be fixed.

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++){
    temp = B[i][j] + A[i][j];
    A[i][j] = A[i-1][j]*temp;
  }
```

(a)



(b)

Figure 2.1: A sample data flow graph. (a) The input imperative specification (b) The corresponding data flow graph. Each dependence is labeled with a scalar value, representing the latency of the source operation, and an iteration vector.

An algorithm can typically be represented by a *data flow graph* (DFG)[3] where each operation is represented by a node in the graph (a circle in the example graph in Figure 2.1(b)). The nodes are linked by dependences (arrows in Figure 2.1(b)), annotated with the latency of the source operation and the iteration vector. Figure 2.1(b) shows the DFG for the loop nest in Figure 2.1(a). Deriving the DFG for an algorithm from an imperative (software like) description requires methods for dependence checking. Enumerating dependences in blocks of straight line sequential code (no loops) is relatively simple, but enumerating dependences in algorithms with loops and array data is a difficult problem, as described in [20].

Two nodes in a dependence graph can only be linked by a dependence if they accesses the same element of data. For algorithms with loops there are multiple instances of each node to consider (one from each loop iteration), and a dependence will exist between the two nodes if any of these instances access the same data. As a result a node

[3]To cope with control constructs such as 'if' statements a *control data flow graph* may be required, but control dependences can be converted to data dependences [19] so that a DFG is sufficient to represent most algorithms.

can even be dependent on itself if two instances of the node from different loop iterations access the same array element. The goal of dependence checking is first to determine whether any instance of one node is dependent on any instance of another node (or the same node). If this is found to be the case the second goal of dependence analysis is to determine if the number of loop iterations separating the dependent node instances is constant, and if so, how many loop iterations at each loop level separate the dependent operations. This allows us to produce an iteration vector for the constant dependences. The process of enumerating dependences in an algorithm generally produces systems of diophantine equations [21] which must be solved. This is a relatively complex and compute intensive mathematical task, the details of which are beyond the scope of this work. A number of tools have already been developed that allow dependence checking, including Petit [22] and the SUIF compiler [23], and they are relatively mature, open source and can be leveraged in the production of any software or hardware compiler.

2.2 Loop Transformations

Most complex real-life algorithms contain loops and, no matter how an algorithm is implemented, it is usually the loops within it that account for the majority of the execution time. For this reason significant effort has been expended in deriving methods to allow loops to be executed more efficiently (in less time) on various architectures and platforms. Numerous methods for transforming loops (especially nested 'for' loops) have been developed and these have been integrated into most software and hardware compilers. This section gives an overview of some of the loop transformations that are referenced later in this thesis. Some of the common goals (or benefits) of each transformation are briefly mentioned, along with how they may be applied. Various papers and tutorials give more detailed descriptions of all of the loop transformations and their usage, including [24–26].

- **Loop Normalisation:** A normalised loop has an index variable that starts at 0 (or 1) and that increments by 1 with every loop iteration. During normalisation a

```
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        a[i][j] = b[i][j] + c[i][j];
```

(a)

```
for (j = 0; j < M; j++)
    for (i = 0; i < N; i++)
        a[i][j] = b[i][j] + c[i][j];
```

(b)

Figure 2.2: An example of loop interchange. (a) The original loop. (b) The loop after interchanging the outer and inner loops.

loop that does not fit this model is transformed so that it does. All references to the index variable must be adjusted to use the new index range, retaining the behaviour of the original loop. Loop normalisation usually simplifies the process of dependence checking.

- **Loop Interchange:** This can be applied to perfectly nested loops to re-order the execution of the loop levels, as shown in Figure 2.2. Loop interchange may be applied at any level in a loop nest if all dependence vectors remain lexicographically positive after the interchange. Loop interchange can be used to improve data locality and to increase loop level parallelism.

- **Loop Merging:** Also referred to as loop fusion, this is the process by which two separate loops are combined to form a single loop with a larger loop body. This is only usually possible if the two loops to be merged have the same loop bounds and index increment/decrement (See Figure 2.12 in the following section for an example). Loop merging can be used to improve data locality and reduce memory requirements, as described in Section 2.4.1. It can also be useful in increasing the size of loops for pipelining, as mentioned in [27].

- **Loop Distribution:** Also referred to as loop splitting or loop fission, this is the opposite process to loop merging. This can be useful if a particular loop body is

too large to implement in parallel on the available resources, and can improve data locality and reduce cache misses in some cases.

- **Loop Unrolling:** A loop is typically unrolled by a given integer, N. The loop body is replicated N times, with any references to the loop index variable in each copy adjusted accordingly. The index variable increment/decrement is multiplied by N and the number of loop iterations reduced by a factor of N. If there are L loop iterations and N is not a factor L, then $(L \,\mathbf{mod}\, N)$ iterations may be 'peeled' from the start of end of the loop before unrolling. A loop may be completely unrolled, *i.e.* N is equal to the number of loop iterations. In this case the loop header can be removed entirely. Loop unrolling can be used to allow higher levels of instruction level parallelism to be achieved during scheduling and to match the parallelism to the available resources. Loop unrolling is often a necessary step in achieving time optimal pipeline schedules [28].

- **Strip Mining:** Strip mining involves splitting a single loop into a nested loop. The resulting inner loop iterates over a group or strip of the original loop iterations, and the new outer loop iterates over all the strips. Strip mining does not change the order of execution of operations in the loop and so may be applied to any loop. Strip mining can be used to match parallelism in the target algorithm to the available resources for implementation. For instance, say every iteration in a given loop may be run in parallel without breaching dependence constraints, but there are only enough resources to implement three iterations in parallel. In this case the loop may be strip mined, with a strip length of three iterations. All three iterations of the resulting inner loop may then be run in parallel, while the resulting outer loop is run sequentially.

- **Loop Tiling:** Loop tiling is a combination of strip mining and loop interchange, so called because, in the case of a double nested loop, it divides the iteration space into a number of rectangular 'tiles'. Depending on the dependences of the loop, either the iterations within each tile can be executed in parallel with the tiles executed sequentially, or vice versa. Loop tiling is implemented in two steps, as demonstrated

```
for (i = 1:100)
    for (j = 1:100)
        a[i+10][j+10] = a[i][j] + b[i][j];
```

(a)

```
for (m = 1:10)
    for (i = 1:10)
        for (n = 1:10)
            for (j = 1:100)
                a[10*m + i+10][10*n + j+10] =
                    a[10*m + i][10*n + j] + b[10*m + i][10*n + j];
```

(b)

```
for (m = 1:10)
    for (n = 1:10)
        for (i = 1:10)
            for (j = 1:100)
                a[10*m + i+10][10*n + j+10] =
                    a[10*m + i][10*n + j] + b[10*m + i][10*n + j];
```

(c)

Figure 2.3: An example of loop tiling. (a) The original loop. (b) The loop after strip mining. (c) The loop after interchange.

for the example loop in Figure 2.3(a). In the first step the two loop levels are strip mined into 10 strips of 10 iterations, as shown in Figure 2.3(b). The iterations in the loops with 'i' and 'j' as the iterators can now be executed in parallel, while the loops with 'm' and 'n' as iterators must be executed sequentially. In the second step the loops are interchanged so the two sequential loop levels are moved to the outermost levels, as shown in Figure 2.3(c).

- **Loop Skewing:** Loop skewing can be used to reorder the execution of iterations in nested loops with loop carried data dependences at every level so that parallel execution may be exploited. This is demonstrated for the example loop in figure 2.4(a). A data dependence is carried across consecutive iterations at both loop levels preventing either loop level from being executed in parallel. Loop skewing involves adding the outer loop iterator to the lower and upper bounds of the innermost loop and adjusting the array indexing to match the new bounds, as shown in figure 2.4(b). If the two loop levels are now interchanged the new innermost loop can be executed in parallel.

```
for (i = 1:N)
    for (j = 1:N)
        a[i][j] = a[i-1][j] + a[i][j-1];
```

(a)

```
for (i = 1 : N)
    for (j = 1+i : N+i)
        a[i][j-i] = a[i-1][j-i] + a[i][j-1-i];
```

(b)

Figure 2.4: An example of loop skewing. (a) The original loop. (b) The loop after skewing has been applied.

2.3 Scheduling Techniques

In the previous section we examined a number of general loop transformation techniques. While these may alter the ordering of operations in an algorithm, they do not impose a specific fixed schedule for the execution of each operation. A start time must be set for each instruction in the loop, whether the intended platform chosen for implementation is an instruction processor or a custom hardware implementation. In this section we examine a number of methods used in software and/or hardware compilers to produce schedules.

The goal when scheduling an algorithm is generally to minimise the execution time of a loop within a given resource budget, or to minimise the resource usage for a given execution time[4]. One of the keys to achieving these goals is the exploitation of parallel execution in the final schedule. Two forms of parallelism may be exploited during scheduling: instruction level parallelism and loop level parallelism. Instruction level parallelism is generally achieved by scheduling individual instructions to execute in parallel and requires *fine grained* scheduling methods, while coarse grained parallelism is generally considered to be the scheduling of whole loops or loop iterations to execute in parallel and requires *coarse grained* scheduling methods. Both are often required to achieve an efficient implementation of a system. For example, consider a multiprocessor system in which each processor core has multiple functional or arithmetic units that may

[4]With the increase in mobile applications and systems the minimisation of power consumption has also become a concern, but we do not focus on this.

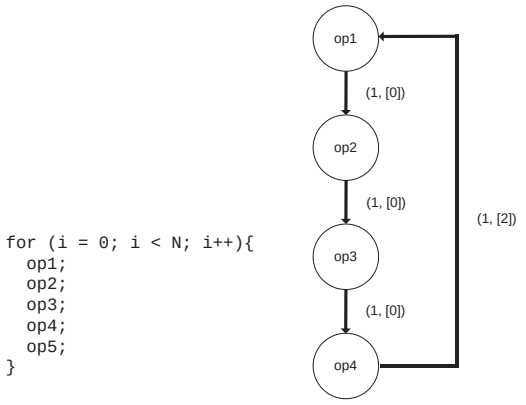
execute in parallel (a common example being an Intel Core i7 multi-core processor [29]). The exploitation of instruction level parallelism would be key to maximising the usage of the resources within each processor core, while coarse grained parallelism would be exploited to utilise as many of the cores as possible.

We focus mainly on fine grained scheduling methods here since they are generally more complex and interesting. While coarse grained scheduling is important, it consists mainly of dependence checking to find whole loops or loop iterations that have no data dependences between them, and then specifying that they execute in parallel. When attempting to schedule loop iterations to execute in parallel, the scheduling process generally amounts to nothing more than applying one or more loop transformations from the previous section to a loop whose iterations are not independent to reorder the iterations such that the data dependences between them are removed [26]. Once this has been achieved the instructions within a single iteration of the modified loop are scheduled using fine grained methods and this schedule is duplicated for each of the parallel iterations. As such, the bulk of the scheduling work is in the fine grained scheduling.

2.3.1 Loop Pipelining

Loop pipelining involves re-organising the execution of a loop so that operations from multiple loop iterations execute in parallel, essentially overlapping the execution of multiple loop iterations. In most pipelining approaches a new loop iteration is started every II clock cycles (where II is termed the initiation interval of the pipeline), and the value of II is less than the number of clock cycles required to execute a single loop iteration. An example of loop pipelining is shown in Figure 2.5.

Loop pipelining is a particularly attractive scheduling tool for generating custom logic as it allows each hardware resource to be utilised for a large percentage of the execution time. Consider the example loop in Figure 2.6(a). The hardware configuration shown in Figure 2.6(b) could be used to execute the loop sequentially, with the operations from each loop iteration executed one at a time in the order shown in the code. In this case each adder or multiplier would only be utilised for 25% of the time, so this would be



(a) (b)

Cycle	i = 1	i = 2	i = 3	i = 4
0	op1			
1	op2			
2	op3	op1		
3	op4	op2		
4		op3	op1	
5		op4	op2	
6			op3	op1
7			op4	op2
8				op3
9				op4

(c)

Figure 2.5: An example of loop pipelining. (a) The original loop. (b) The DFG for the given loop. (c) The pipelined loop schedule for the first four iterations.

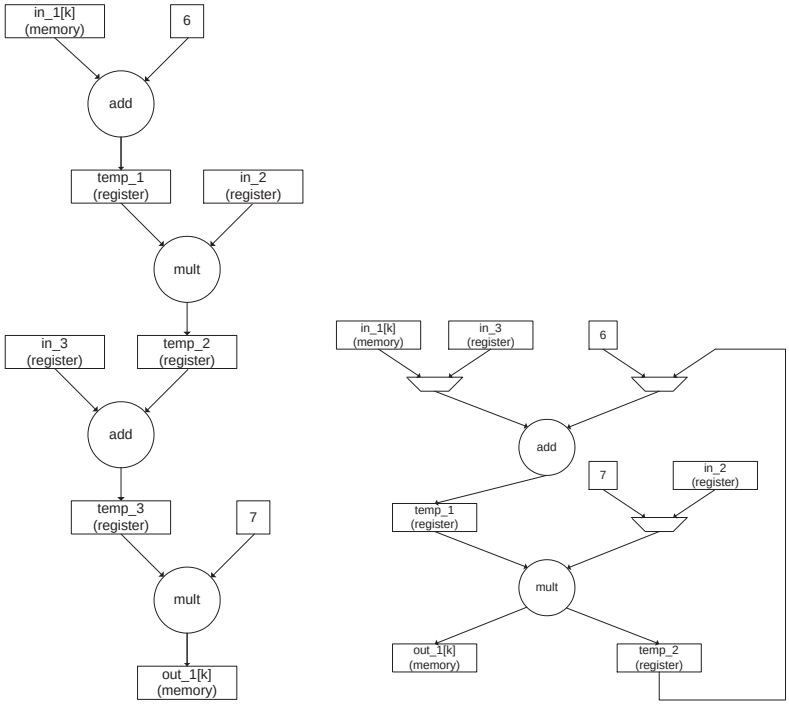
an inefficient (and slow) implementation. A single adder and multiplier could be shared between operations within a loop iteration, as shown in Figure 2.6(c). Instantiating two copies of the hardware shown in Figure 2.6(c) would allow two loop iterations to be executed in parallel[5] using the same number of adders and multipliers as before, offering a speedup of 2x. However, we have incurred a cost of 3 multiplexors and potentially decreased the maximum clock frequency for the design as the multiplexors will add extra delay along a number of paths in the circuit, and each adder and multiplier is still only used for 50% of the time. By retaining the circuit of Figure 2.6(b) and starting a new iteration on every clock cycle (instead of waiting 4 clock cycles for each iteration to complete to begin a new iteration) a speedup of nearly 4x can be achieved compared to sequential execution, along with a resource usage of nearly 100%. The only times when adders or multipliers go unused are during the first three and final three iterations as the pipeline fills and empties. This represents a near 2x improvement over unrolling the loop and sharing the resources, and comes without the multiplexor or potential operating frequency cost (though one extra register is required).

A good overview of loop pipelining methodologies is available in [30]. The earliest pipelining approaches worked by first unrolling the target loop (always the innermost loop if a loop nest is considered) a number of times. The operations in the body (a single

[5]This assumes that there are sufficient memory ports available to access all the data required.

```
void example (int *in_1, int in_2, int in_3, int *out_1)
{
      int k;
      for (k=0; k<100; k++)
         out[k] = (( in_1[k] + 6 ) * in_2 + in_3 ) * 7;
}
```

(a)



(b) (c)

Figure 2.6: An example demonstrating the resource efficiency of loop pipelining. (a) C code for a function to be implemented in hardware. (b) A sequential implementation with no resource sharing. (c) A second implementation with resource sharing.

iteration) of the unrolled loop are then moved up and down through prototype schedules until a repeating pattern of operations emerges. This repeating pattern must contain P copies of every operation present in the original loop body (where P is some integer greater than 0), though the operations in the pattern may be from different iterations of the original loop. A single unit of the repeating pattern formed becomes the body of the pipelined loop, which is executed N/P times, where N is the number of iterations in the loop. If N is not a multiple of P, iterations may be peeled from the beginning or end of the loop as required and executed sequentially. The number of clock cycles

required to execute a single unit of this repeating pattern is referred to as the initiation interval, II, and the goal is generally the minimisation of II. Examples of pipelining methodologies that take this approach can be found in [31, 32]. The main disadvantage with this pipelining approach is that there may be no upper bound on the time taken to for a pattern to emerge [33]. Another approach is to deliberately schedule the loop body so that a pattern is formed. The most common method to follow this approach is *modulo scheduling* [30].

Modulo scheduling works by scheduling the operations of a single loop body so that, if a new iteration is started every II clock cycles, then no dependence or resource constraints are breached. The loop body is scheduled for a candidate II value so that all dependence constraints within the loop body are satisfied and all loop carried dependences are satisfied, assuming the given value of II. Let the first scheduled operation in the loop body start at time $t = 0$, and the last operation end at time $t = L$. The loop body must also be scheduled so that the resources required by all operations assigned to t values with the same $(t \,\mathbf{mod}\, II)$ do not exceed the available resources. The value of II should always be minimised to minimise the length of the complete loop schedule.

Constraints on the Minimum Initiation Interval

Since pipelining approaches generally seek to minimise the value of II for a given loop and target platform, it is often useful to know what this minimum value is. The lower bound value of II is determined by a combination of two factors: the dependence constraints of the loop and the resource constraints of the target platform [34]. Using the notation presented in [34], the data dependences in the loop will restrict the value of II to be greater than or equal to some value, $RecMII$, and the resource constraints will restrict the value of II to be greater than or equal to a second value, $ResMII$.

The value of $RecMII$ for a given loop is determined by any *cyclic paths* that exist in the loop's data dependence graph [34]. A cyclic path exists where a node (operation) in the graph is both the source and sink for a chain of dependences, at least one of which will be carried across multiple loop iterations. For example, consider the simple loop shown

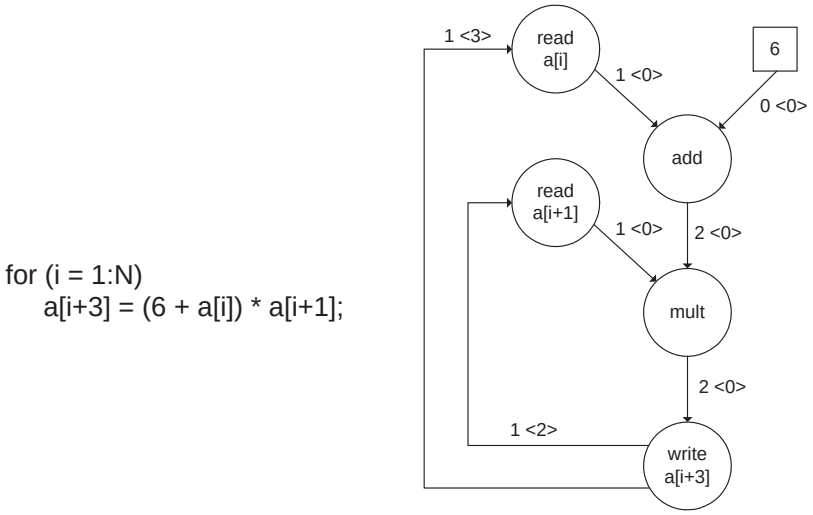


Figure 2.7: (a) An example loop with two loop carried dependences. (b) The corresponding data dependence graph with two cyclic paths. The first cyclic path passes through the 'read a[i]' node to the 'write a[i+3]' node, and the second passes though the 'read a[i+1]' node to the 'write a[i+3]' node. Each dependence is tagged with a number inside angular braces representing the number of loop iterations that each dependence crosses, and a number outside the braces representing the latency of the source operation.

in Figure 2.7(a) and the corresponding data dependence graph in Figure 2.7(b). There are two dependences which cross loop iterations (between the write and the two read operations) and these produce two cyclic paths. The cyclic paths set lower bounds for the value of $RecMII$ determined by the total latency of the operations in a single cycle of the path, L_{cycle}, and the sum of the loop iterations spanned by the dependences in a single cycle of the path, I_{cycle}. Each cyclic path constrains the value of $RecMII$ as shown in Inequality (2.1) [34].

$$RecMII \geq \lceil L_{cycle}/I_{cycle} \rceil \tag{2.1}$$

The two cyclic paths in the dependence graph in Figure 2.7(b) have latencies of 4 and 6 cycles spanning 2 and 3 iterations respectively. This give a $RecMII$ value of 2 cycles.

The value of $RecMII$ can be found by enumerating all cyclic paths in a dependence graph and applying Inequality (2.1) to every path, but the worst case execution time of such a method scales exponentially with the number of nodes in the dependence graph so it is not generally practical. However, the algorithm presented in [35] can be used to

find the value of $RecMII$ and its run time scales polynomially with the number of nodes. The details of this algorithm are beyond the scope of this thesis, but it is useful tool for placing lower bounds on the initiation interval that may be achieved for a given loop.

As stated previously, $ResMII$ is a lower bound placed on the initiation interval by the resource constraints of the target platform. In general the constrained resources can be arithmetic or logic units, look-up tables, shared buses and/or memory ports. The resource constraints limit the minimum achievable II because the iterations with overlapping executions must use the same resources and a new iteration cannot begin until previous iterations have stopped using the shared resources. For example, consider a loop with one multiplication per iteration which must be executed on a target platform with a single multiplier with an unpipelined latency of 3 cycles. In this case the initiation interval could not be less than 3 cycles as each successive loop iteration must wait for the previous iteration to complete the multiply.

The calculation of $ResMII$ can be a complex process in the general case, requiring the use of reservation tables and collision vectors [34]. Such methods are required when considering processor architectures with instructions that require a sequence of resources to be available at fixed intervals (relative to the first step of the instruction). For example, a multiply might require an input bus to be free for a cycle, followed immediately by the multiplier and then the output bus. However, when scheduling to generate custom datapaths for FPGAs, such complex resource requirements are rarely necessary as any number of registers and/or multiplexors could be added between resources, so strictly timed sequences of operations on resources are not required. Furthermore, the high logic densities of modern FPGAs (which can feature tens to hundreds of thousands of look-up tables and thousands of dedicated fixed point multipliers [3, 4]) mean that the arithmetic and logic resources available are often not the limiting factor. Each arithmetic and logic operation in a single iteration of the loop can allocated to a dedicated resource, leaving only the memory ports through which data is accessed as the only constrained resource [36]. The value of $ResMII$ is then determined by the ratio of memory accesses to the number of ports through which the accesses may be made.

Let II_{min} be the minimum initiation interval that may be achieved for a given loop on a given platform. The values of $RecMII$ and $ResMII$ place a lower bound on II_{min}, but this lower bound may not always be achievable. This is because the value of $RecMII$ is calculated in the absence of any resource constraints, and the value of $ResMII$ is calculated in the absence of any dependence constraints. When both sets of constraints are combined (*i.e.* when we actually attempt to schedule for the minimum initiation interval) the resulting minimum initiation interval may be greater than that imposed separately by either set of constraints. For example, let us again consider the loop in Figure 2.7(a), which has a $RecMII$ value of two cycles. If this loop were to be implemented on a target platform with one read port and one write port then the value of $ResMII$ would also be 2 cycles (as there are two reads which must use one read port), meaning the lower bound on the initiation interval is 2 cycles. However, it is not possible to modulo schedule the loop with an initiation interval of 2 cycles without introducing a resource conflict where two pipelined iterations try to use the same port at the same time. To meet the dependence constraints for an initiation interval of 2 cycles (ignoring resource constraints) each iteration of the loop must be scheduled as shown in Figure 2.8(a) in order to minimise the latency between each read operation and the write operation (to minimise the lengths of the two cyclic paths). To achieve an initiation interval of 2 cycles the operations must be split into pipeline stages of 2 cycles, which produces the pattern shown in Figure 2.8(b). Modulo scheduling requires that all of these stages can be executed in parallel (for different loop iterations) without breaching the resource constraints, but we can see that both read operations are scheduled to cycle 0 of their respective stages, meaning that they will both require the read port of the memory at the same time if a new iteration is started every 2 cycles. Moving either read operation forward one cycle will breach the dependences within each loop iteration, and moving either read back one cycle will increase the length of one of the cyclic paths. Hence an initiation interval of 2 cycles cannot be achieved. As shown in Figure 2.8(c), an initiation interval of 3 cycles can be achieved, and this is therefore the minimum initiation interval for the given loop on the given platform.

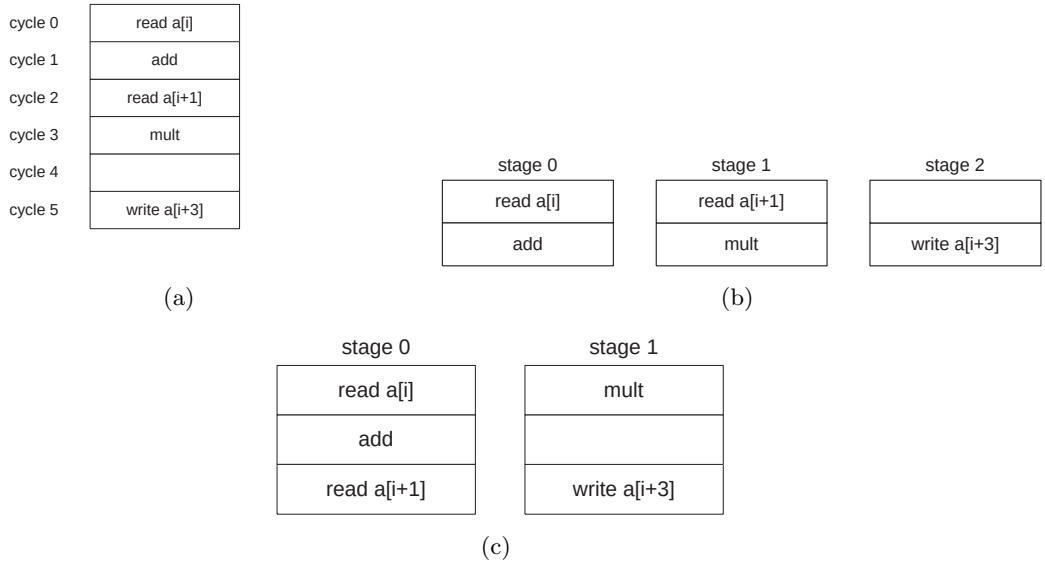


Figure 2.8: Scheduling the loop in Figure 2.7(a) to meet resource and dependence constraints. (a) Scheduling to minimise the cyclic paths in the absence of resource constraints. (b) Splitting the schedule into pipeline stages of 2 clock cycles. The stages cannot be executed in parallel as both read operations will require the read port at the same time. (c) Splitting the schedule into stages of 3 cycles removes the resource conflicts.

Even though the minimum initiation interval estimated using the values of $ResMII$ and $RecMII$ may not be achievable, it is still useful as it allows methods such as *iterative modulo scheduling* [34] to be used. Iterative modulo scheduling attempts to modulo schedule the loop for increasing values of II until it is successful. Because the minimum II predicted by $ResMII$ and $RecMII$ will never be an overestimate, this value serves as a good starting value of II.

2.3.2 Pipelining Above the Innermost Loop

Most pipelining approaches consider only the innermost loop of a nest, but there are some existing approaches for extending pipelining to higher loop levels. In *dovetail pipelining* [37] the innermost loop is first pipelined as normal (using any standard method). For each iteration of the loop level one step above the innermost loop there will be a pipeline fill and a pipeline flush. Dovetail pipelining merges the flush of one iteration with the fill of the next iteration to keep the pipeline full, as shown in Figure 2.9.

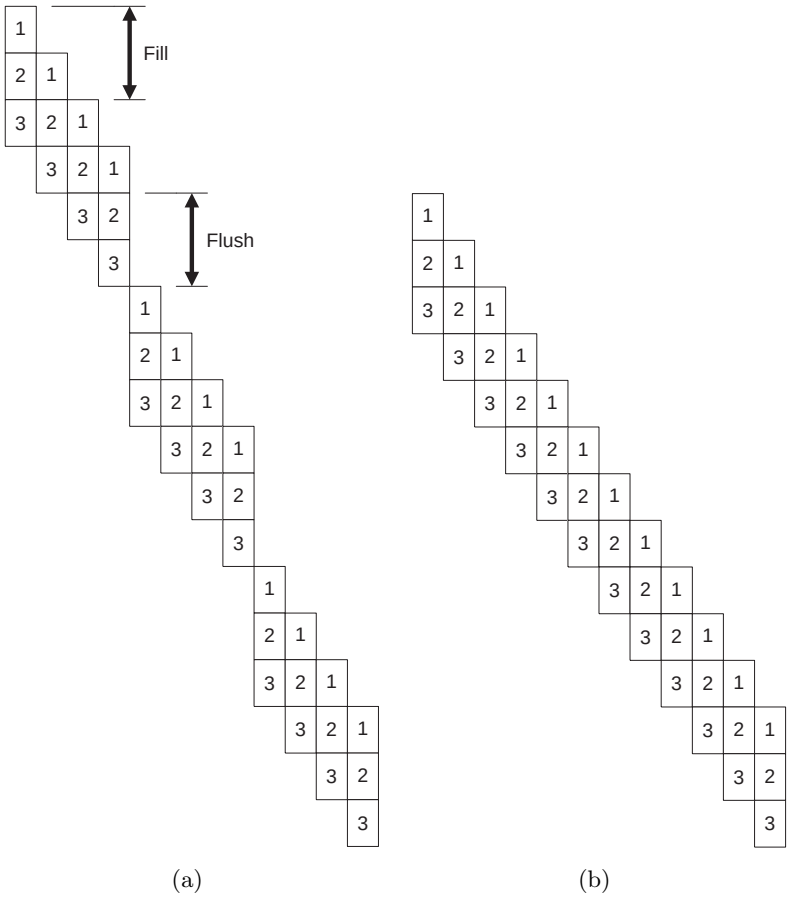


Figure 2.9: An example of dovetail pipelining for a double nested loop with four inner loop iterations and three outer loop iterations. (a) The pipeline scheduling before dovetailing. Each numbered box represents a pipeline stage. There is a separate flush and fill for each outer loop iteration. (b) The schedule after dovetailing has overlapped the flush and fills of consecutive outer loop iterations.

The *single dimension software pipelining* [13], *unroll and squash* [38] and *interlaced loop software pipelining* (ILSP) [39] methods take a different approach to outer loop pipelining. All three methods pipeline a single loop level above the innermost loop, executing the iterations of the innermost loop sequentially, as normal. This ensures that all dependences at the innermost loop are automatically honoured. The iterations of any loops above the pipelined level also run sequentially so dependences at this level are honoured. Unroll and squash is specific to double nested loops, where the outer loop is pipelined and the inner loop is run sequentially. ILSP is designed to pipeline irregularly nested loops, while single dimension software pipelining is focused towards perfectly nested loops (though it can be extended to cover irregular nesting [40]). Single dimension software

pipelining may be applied to a loop nest of any depth and pipelining may be applied at any level in the nest, including the innermost level. As a result, this method can always find a result at least as good as that obtained using standard inner loop methods.

An optimal pipelining approach is presented in [41]. This method assumes no resource constraints and uses a generalised form of the *hyperplane* scheduling technique [42] to derive the minimum initiation interval at each loop level, according to the dependences in the target loop nest. Every loop level in the loop nest is then pipelined to generate the optimal (shortest) pipeline schedule available.

2.3.3 Scheduling Optimisations

The remainder of this section examines a number of existing optimisation methods that can be employed during scheduling to improve results, either allowing shorter schedules to be produced or reducing the hardware resources required.

Single Assignment

As mentioned in Section 2.1, an anti-dependence exists in an algorithm where a variable is read at a given point in the algorithm description and then assigned to at a later point. Such a dependence implies that the read is always executed prior to the assignment to preserve the correct output. However, *single assignment* transformations can allow these dependences to be removed, removing the scheduling restriction. Anti-dependences on scalar (non-array) variables can be removed by converting to *static single assignment* form [43]. Under static single assignment the code is transformed so each scalar variable is assigned to only once during the course of the algorithm. The first assignment to the variable remains unaltered, but a new variable is introduced for every subsequent assignment. The read operations are altered so the correct variable is accessed for each, depending on the number of assignments that have occurred in the code before each read. Since each variable is now assigned to only once, and all reads to each variable occur after the assignment, there will be no anti-dependences. However, conversion to static single assignment form can increase the register requirement for the final system as a single

variable (register) with multiple writes is converted to multiple variables (registers), each with a single write.

Dynamic single assignment [43] is similar to static single assignment, but it may be applied to code with loops and array structures. In this case the algorithm is altered so each array element is assigned to only once. As with static single assignment, this will remove anti-dependences but will increase the memory requirements as whole arrays must now be duplicated if they are written to multiple times.

A related transformation is *dynamic renaming*, which can be viewed as a more selective version of static single assignment. Instead of introducing new variables for every assignment in the algorithm, a scheduling tool might identify a specific operation to move to an earlier start time which is constrained by an anti-dependence. A new variable can then be introduced for this operation alone, rather than the whole algorithm. Dynamic renaming must often be used in conjunction with the speculative methods described in the following section. An example of dynamic renaming can be seen in Figure 2.10, where a new variable, 'x_temp', must be introduced to temporarily store the results of the subtraction operation until the branch condition has been evaluated.

Speculative Methods

Speculative optimisation methods allow operations to be moved into, out of or around conditional branches in an algorithm. *Speculation* takes operations from within a conditional branch and relocates them to a point before the evaluation of the branch condition. This can allow these operations to be executed in parallel with the evaluation of the branch condition, where before a dependence of the conditional operation would force them to be executed sequentially. To ensure the correct result is still produced an additional operation is added to the algorithm to select the correct value for the variable once the condition has been evaluated, but the latency of such an operation would typically be less than that of the arithmetic operation that has been moved. Figure 2.10 shows a simple example of speculation which also requires dynamic renaming to be applied.

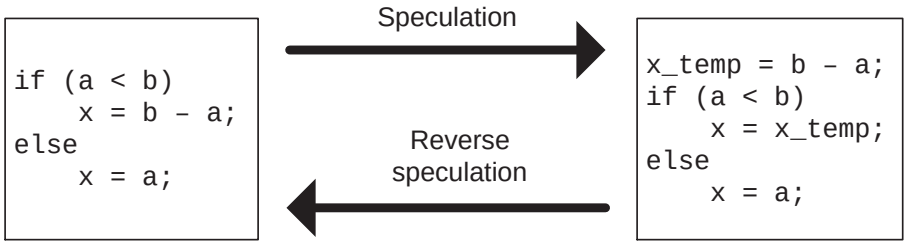


Figure 2.10: Speculation transforms the pseudo code on the left into the pseudo code on the right, while reverse speculation performs the reverse transformation. Dynamic renaming is required during speculation to create the 'x_temp' variable.

Reverse speculation moves operations from before the evaluation of a condition into one or more of the branches arising from the condition. If the variable assigned to by the moved operation is only read by operations in the branches of the condition, the operation must only be moved into the branches where the variable it assigns to is read. This may allow execution time to be reduced if the moved operation cannot be executed in parallel with the operations before the branch, but may be executed in parallel with operations inside the branch. Both speculation and reverse speculation can also be useful in *branch balancing* [44]. During branch balancing operations are moved into and/or out of conditional branches in an attempt to equalise the latencies of all the branches. This can be especially useful when generating custom datapaths in hardware since the operations scheduled after the merging of the conditional branches have to be scheduled for the worst case branch. Hence, if one branch is longer than the other(s) and operations after the branch merge can be moved into one of shorter branches or executed in parallel with operations in the longer branch, then the overall execution time can be reduced. An example of branch balancing to reduce execution time is shown in Figure 2.11.

Common Subexpression Elimination

Common subexpression elimination (CSE) can reduce both the execution time of an algorithm, and the hardware resources required by locating subexpressions (logical and/or arithmetic operations) that appear repeatedly in an algorithm and assigning them to variables. Each use of the subexpression is them replaced by an access to this variable,

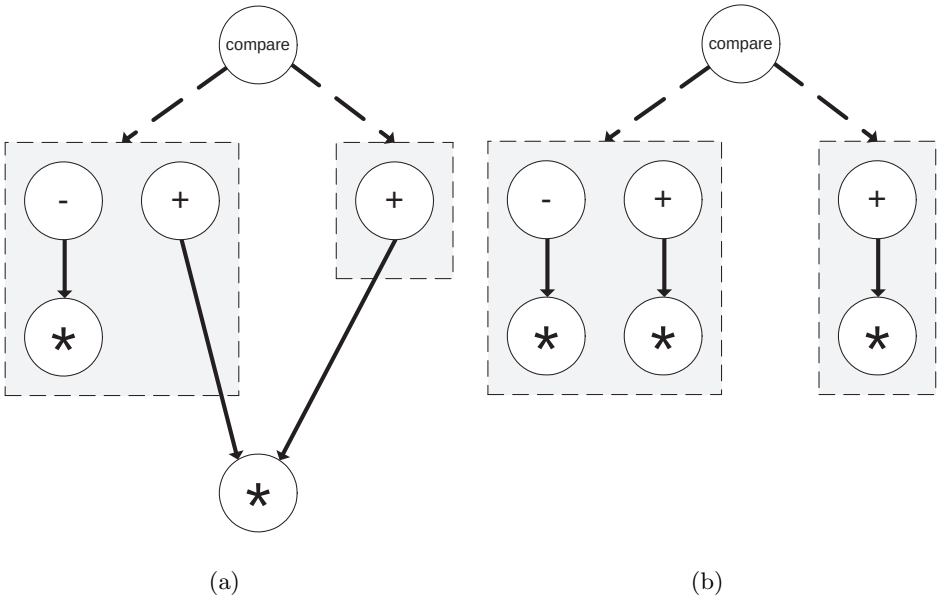


Figure 2.11: An example of branch balancing allowing a shorted schedule. (a) The dataflow graph for an algorithm prior to branch balancing. The final multiply is dependent on the add operation in whichever branch is executed and cannot be scheduled to start until after the branches have merged. (b) Moving the multiply into both branches increases the latency of the right hand branch but not the left hand branch so the worst case delay through the branch remains unchanged.

removing the need to recalculate the same value multiple times [17]. CSE can either be applied before or after scheduling, or combined with scheduling in a process referred to as *dynamic CSE* [17]. CSE can only be applied up to the point where the value of any of the variable in the common subexpression changes. At this point the subexpression must be recalculated. Moving the operations that affect the common subexpression back or forward in time within the schedule will affect the point at which the subexpression must be recalculated, and so combining CSE with the scheduling decisions can increase the number of subexpressions replaced. The work in [17] has shown significant improvements when using dynamic CSE compared to CSE pre or post scheduling.

Retiming

The optimisations described so far are applicable to both hardware and software compilers, but *retiming* is a hardware specific optimisation. A typical custom datapath comprises combinatorial logic elements with registers on some of the paths connecting logic blocks.

The delay of the longest path between two registers determines the maximum frequency at which the datapath may be clocked. Retiming involves moving registers back or forward through a datapath to minimise the length of the longest path in the circuit [45].

2.4 Memory Optimisation

The memory subsystem for an FPGA based design (or any embedded system) can usually be highly customised to allow the best performance, area and/or power characteristics for the given application. The term 'memory optimisation' covers not just transformations and methods for improving the memory structure, but also includes some loop and scheduling transformations that help tailor the target application to a given memory subsystem. No matter how a given optimisation technique operates, it will attempt to improve performance in one or more of the following three areas.

1. Minimisation of memory power consumption.
2. Minimisation of memory usage.
3. Maximisation of system performance (speed).

Various memory optimisations and how they affect each of these metrics are discussed in the subsections that follow.

2.4.1 Loop and Data Restructuring

It may be possible to reduce both the number of memory accesses and the amount of memory required by restructuring the data flow within the target application. The data flow will often be derived from an input specification written in a software language, such as C. This input specification may contain redundant arrays, data transfers and data accesses that can be removed without affecting the correctness of the final system.

Consider the pseudo code in Figure 2.12(a). Assuming that all arrays are stored in a global memory, the first loop will require 2MN memory accesses (MN reads and MN writes) and the second loop will require 3MN accesses (2MN reads and MN writes). The

```
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        b[i][j] = a[i][j] + 6;

for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        c[i][j] = b[i][j] * c[i][j];
```

(a)

```
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++){
        b[i][j] = a[i][j] + 6;
        c[i][j] = b[i][j] * c[i][j];
    }
```

(b)

```
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++){
        b = a[i][j] + 6;
        c[i][j] = b * c[i][j];
    }
```

(c)

Figure 2.12: An example of loop merging to reduce memory accesses. (a) The original loop structure. (b) The resulting loop after merging (c) The resulting loop after scalar replacement.

two loops may be merged, as shown in Figure 2.12(b), since they have the same loop bounds and there are no dependences preventing this. While this does not necessarily reduce the number of memory accesses, merging the two loops does allow the 'b' array to be reduced to a single scalar, as shown in Figure 2.12(c) (This assumes that the 'b[i][j]' values are not referenced outside of these loops). Assuming that the 'b' variable can be held in a local register, this scalar replacement will reduce the number of memory accesses to 3MN, an overall reduction of 40%, and reduces the memory usage by MN words. This example uses loop merging to improve the data access patterns, but other transformations such as loop interchange or loop skewing can also serve to improve memory access patterns [15, 46, 47].

```
for (i = 0; i < N; i++)
     for (j = 0; j < M; j++)
          a[i][j] = b[i][j] + c[j][i];
```

(a)

```
for (i = 0; i < N; i++)
     for (j = 0; j < M; j++)
          a[i][j] = b[i][j] + c[i][j];
```

(b)

Figure 2.13: Improving data locality. (a) The original loop. (b) The loop after swapping rows and columns of *c*.

The layout of the application data can also be modified to data improve both spatial locality (successive accesses address neighbouring memory elements) and temporal locality (successive loop iterations access the same data values). This is demonstrated for the pseudo code in Figure 2.13. Figure 2.13(a) shows a loop nest in which the accesses to the `a` and `b` arrays move along a given row as the inner loop increments, providing good spatial locality. However, accesses to the `c` array move along a given column, offering potentially poor spatial locality if the array is stored in row-major format. By swapping the rows and columns of the `c` array, essentially storing `c` in column major format, the spatial locality is improved. Note that this transformation must be applied globally, across all references to `c` in all program loops, potentially reducing spatial locality elsewhere.

Loop and data transformations along these lines can not only reduce the amount of memory required, but also the power usage of the system since they can reduce the overall number of accesses and reduce the distance between sequential accesses (reduce the switching in the address wires). There is a significant amount of existing work relating to loop and data transformations to improve data locality and reduce storage and transfer redundancy. An overview of existing work in this area can be found in Section 2 of [48], while more detailed descriptions of methods and approaches for applying such transformations are included in [49–51]. The work in [49] in particular seeks to reduce power consumption as well as memory area.

Much of this work seeks to improve data locality in the source algorithm to improve

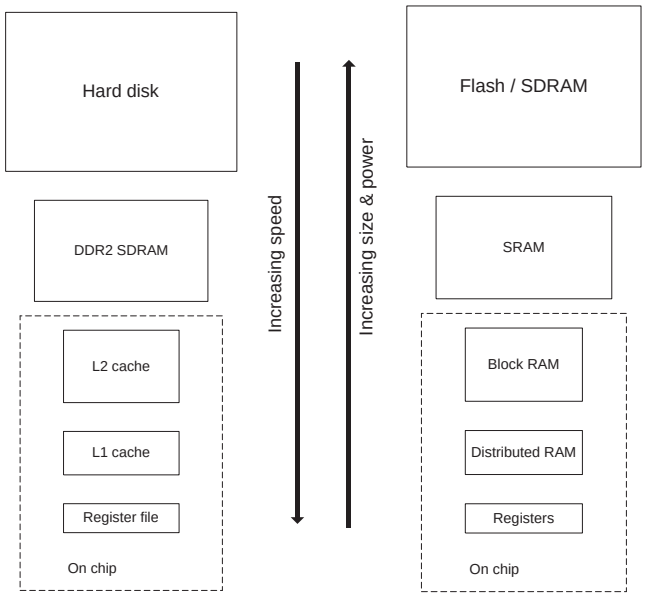


Figure 2.14: (a) Memory hierarchy for a typical desktop computer. (b) Possible memory hierarchy for an embedded FPGA based system.

cache level data reuse [52]. While FPGA based systems do not tend to employ data caches, improving data locality in the source algorithm can still be desirable. This can simplify data reuse exploitation and array to memory mapping optimisations that are discussed in the following sections. Generally, applying the type of loop and data transformations described this section can provide a better starting point from which further memory and scheduling decisions can be made.

2.4.2 Data Reuse Exploitation

Most computing platforms allow/utilise a hierarchical memory structure, with multiple (at least two) types of memory resource making up multiple levels in the hierarchy. For example, even a single FPGA with no off-chip memories has the potential for an internal memory hierarchy, with registers, distributed RAM[6] and block memories forming the different layers. Figure 2.14 shows two example memory hierarchies, one for a typical desktop computer and another for an FPGA based embedded system.

[6]Xilinx and Altera (Stratix III devices onwards [53]) include extra hardware that allow the LUTs to be configured as small RAMs.

Generally, accesses to each layer of the hierarchy may incur a different cost, with accesses to lower levels in the hierarchy having a lower cost than accesses to higher levels. The cost is usually quantified in terms of the latency or power consumed. The available bandwidth at each level is also of interest. Lower levels in the memory hierarchy should allow higher levels of parallel access (more ports). Combined with a potentially lower access latency, the lower levels in the hierarchy will be able to supply data to a processing unit at a higher rate than memories at higher levels. Hence, one might ideally like to store all data in the lower level memories to maximise system speed. However, at the lower levels in the hierarchy, the size of the available memories is usually quite small, while higher levels offer larger memories. Thus, larger data items (*e.g.* multidimensional arrays such as images or large matrices) must be stored at higher, slower levels in the hierarchy.

For algorithms that process large data structures (such as image processing algorithms) the memory bandwidth can quickly become the bottleneck in the system and limit performance [36]. One method to get around this problem is to exploit any data reuse in the target algorithm. This generally involves identifying smaller portions of a data structure, rows of a large array for example, that are read multiple times in a loop, and storing them in lower levels in the hierarchy. This data may be buffered for reuse when it is first read, or when it is generated if some dependence (usually loop carried) means that its value will be required in a future calculation. Existing literature identifies four opportunities where data may be reused [47]:

- Self-temporal reuse: the same element of data is accessed by multiple instances of the same array reference in multiple iterations of a loop. Self-temporal reuse is a form of inter-loop reuse, since data is reused across multiple loop iterations.

- Group-temporal reuse: the same element of data is accessed by different references to the same array. Since the array references accessing the same data can be in the same iteration or different iterations, group-temporal reuse can lead to inter-loop reuse and intra-loop reuse.

- Self-spatial reuse: the elements of data accessed by multiple instances of the same

array reference in successive loop iteration are on the same cache line or in the same level of a memory hierarchy. This is another form inter-loop data reuse.

- Group-spatial reuse: the elements of data accessed by different references to the same array, either in the same loop iteration or different loop iterations, are on the same cache line or in the same level of a memory hierarchy. Group-spatial reuse can lead to inter-loop reuse and/or intra-loop reuse.

There are two main methods for reusing data in computing platforms: either through generic automated caches or through customised scratch pad memories and/or shift registers and FIFOs. These are examined in the following two sub-sections.

Cache Based Data Reuse

Automated data caches are commonly used to exploit data reuse in embedded systems and generic computing platforms. The major advantages of caches are that they are generic so the same components can be used in many systems, and that caches exploit the implicit reuse within the algorithm. While transformations such as those described in Section 2.4.1 can increase the amount of data that is reused (increase cache hits), caches can exploit data reuse in algorithms without applying any transformations and without altering any components of the cache to suit the algorithm. Multiple algorithms can execute on the same platform, and algorithms for embedded systems can be updated easily without having to modify the hardware. The other advantage of caches is that they can exploit data reuse in algorithms where the patterns of data access are data dependent and cannot be determined at compile time [54].

While caches have been successfully employed in generic systems, they are less successful at exploiting reuse in data dominated applications such as image processing and scientific computing [55, 56]. The large data sets involved and frequent accesses can lead to large numbers of cache misses, degrading the overall system performance. A number of methods have been proposed which attempt to predict and reduce cache misses. The methods presented in [57] monitor frequently occurring address conflicts which lead to

cache misses. The average time between accesses to conflicting addresses is calculated and additional hardware used to fetch the new cache line from memory so it can be loaded into the cache just before it is required and after any accesses to the previous set of data occupying the cache line.

Other cache optimisation approaches use partitioning to reduce cache misses. Partitioning involves separating the memory used by the cache into multiple independent sections. Methods differ on how the partitions are filled with data to improve performance. The work in [56] and [58] uses two cache partitions, with one partition used to exploit spatial locality and the other used to exploit temporal locality. The work described in [59] can use any number of partitions and attempts to dynamically target array references in loops to cache partitions so that each reference is assigned to a cache partition with other references that are likely to use the same data. In [54] multiple partitions, referred to as sub-caches, as used to provide multiple ports through which FPGA datapaths may access data in parallel.

Custom Data Reuse Structures

The work in [54] investigates the use of multi-port caches for FPGA datapaths and shows that they can afford performance gains, especially in algorithms with data dependent array addressing. However, algorithms with address functions that are affine functions of loop indices are commonly targeted to FPGA based systems, and in such algorithms it is possible to infer data reuse patterns at compile time. Once these patterns have been uncovered, custom data structures based around scratch pad memories (commonly referred to as buffers) and/or shift registers can be created to reuse data potentially more effectively than generic caches, typically with a lower hardware cost.

The cost to the designer for custom data reuse is the dependence analysis required to infer the reuse patterns, and the selection of the memory and control structures to exploit the data reuse. An example of how a loop nest with implicit data reuse may be modified to explicitly reuse data is shown in Figure 2.15. Note that, while data caches will exploit implicit data reuse though temporal and spatial locality, high level transformations such

```
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        for (k = 0; k < L; k++)
            a[i][j+k] = b[i][k] + c[i][j];
```

(a)

```
for (i = 0; i < N; i++){
    for (p = 0; p < L; p++)
        b_temp[p] = b[i][p];
    for (j = 0; j < M; j++){
        c_temp = c[i][j];
        for (k = 0; k < L; k++)
            a[i][j+k] = b_temp[k] + c_temp;
    }
}
```

(b)

Figure 2.15: Inserting explicit data reuse. (a) The original loop nest. (b) The loop nest transformed for explicit data reuse.

as those shown in Figure 2.15 can still improve cache performance. Explicitly declaring the desired data reuse in this manner can steer later compilation stages to ensure that the algorithm is finally implemented in such a way that this reuse does occur.

High level code transformations which add extra arrays and loops into the source code to make data reuse explicit, such as those shown in Figure 2.15, have been explored by a number of researchers in the field [60–62]. The methods presented generally rely on exposing all possible data reuse possibilities through the creation of a tree structure. Each leaf node in the tree represents the inclusion of a different subset of the possible buffers that can be instantiated in the various levels of the target memory hierarchy. The tree is traversed and the scheme with the lowest value of some cost function, based on speed and/or power usage, is chosen for implementation. A custom memory hierarchy can then be designed onto which the reuse scheme can be effectively mapped [61], or the reuse scheme can be built around a predefined memory hierarchy [62].

Transformations of this nature can generally allow the speed of the final implementation to be increased as there are fewer accesses to the large, slow memories. Furthermore, the power consumption of the system can be significantly reduced, especially if accesses to off-chip memories (which have a high power cost) are replaced with accesses to on-chip memories. The work in [63] details a methodology for optimising accesses to a memory hierarchy with the explicit goal of minimising power consumption within real-time execution constraints. The real-time execution constraints for the target system and algorithm can usually be met reasonably easily, allowing good scope to minimise the power consump-

tion. However, adding extra arrays to an algorithm will always increase the total memory required. When targeting systems with predefined memory architectures this is not such a problem, so long as the total storage required after data reuse exploitation is not greater than the memory available. When the freedom exists to build a custom memory hierarchy around the data in the system, adding extra arrays may increase the memory components required. This extra memory cost must then be traded against the performance gains and/or power reductions as in [63].

Other data reuse methods more specific FPGA based systems have been proposed which focus more on improving speed and less on power consumption. For FPGA based systems it is often the available memory bandwidth that limits pipeline speed, causing the pipelined data-paths to pause (stall) while input data is retrieved. The main goal of these optimisations is to increase pipeline performance by reusing previously read data, thus reducing the number of memory accesses and the number of pipeline stalls. The work in [36] uses shift registers to reuse data where the same array element is accessed in successive iterations of the inner loop. Where larger quantities of data, such as rows of array data, can be reused between iterations at levels above the innermost loop, on-chip RAMs are inferred. This data reuse methodology is used in connection with Pipeline Vectorization [64]. A similar register/FIFO based approach is presented in [65]. This work uses 'input queues' to store data that may be reused. Simple, one dimensional queues (tapped delay lines) are formed to account for data reuse in innermost loops. One dimensional queues can then be linked by further one dimensional queues to form 'window queues' to account for reuse at loop levels above the innermost loop.

This type of shift register and FIFO based reuse will typically be of use in applications that use sliding *data windows*. These are common in signal and image processing algorithms, such as filters, where a window (pattern) of accesses moves across an array of input data as computation progresses. Further work on data reuse with 'data windows' is detailed in [66], where 'smart buffers' are used to store, reuse and supply window data to the data-paths generated in the ROCCC compiler [9]. Rather than using shift registers of the correct length to supply reuse, the 'smart buffers' are actively controlled so that the

number of elements in the buffer may be minimised while ensuring that live data (which may be used again in subsequent windows) is not overwritten. This approach may allow greater levels of data reuse with fewer storage elements than the shift register based methods, but it will require larger and more complex control logic and may lead to a lower clock frequency.

Recent work (carried out at the same time as the work presented in this thesis) targets hierarchical data reuse specifically to FPGAs [67–70]. Data reuse structures may be introduced at any level in a nested loop, implemented in on-chip SRAM blocks or registers. All of the data reuse options are explored and formed into a tree, enumerating all possible selections of reuse options with one or less reuse structures selected at each level of the loop [67]. Methods are introduced to minimise the storage required by each buffer [68], along with methods to select the set of buffers to implement within the available resources such that the access to off-chip memories or power consumption may be minimised [69]. Further methods are explored to instantiate multiple copies of buffers so that loops may be unrolled to maximise the parallelism exploited [70].

2.4.3 Data to Memory Allocation

All of the data used by the target algorithm must ultimately be stored in a storage component (register or memory block) in the memory subsystem. How data is assigned to memory can affect not just the amount of storage required, but also how the algorithm may be scheduled and how much power the system consumes. Early work on behavioural synthesis used a single memory strategy where it is assumed that all data is stored in a single, monolithic memory model [71, 72]. This allows easy support for pointers and dynamic memory constructs, but can severely limit system performance as parallel data access is limited to the number of ports on the memory (which is usually small) so most memory accesses must be serialised. Other compilers, such as the Synopsys Behavioural Compiler, used a one-to-one memory strategy where each array in the algorithm is assumed to map to its own memory block (scalar variables are targeted to registers). This approach obviously offers greater parallel access than the single memory strategy, but it

assumes that all arrays require the same levels of parallel access. The critical arrays may still have too few ports through which they may be accessed, while the ports to other arrays are seldom used. Also, assigning each array its own memory block can be area expensive, especially when targeting FPGAs where memory blocks have predetermined, discrete sizes. Grouping small arrays that require little parallel access together in the same memory block may not affect the lowest schedule length achievable, but may reduce the number of memory blocks used.

More recent studies on behavioural synthesis have investigated methods to allow a better tradeoff between area (memory bits/blocks and address decode logic) and speed (schedule length) to be investigated. The methods used fall into two general categories: *Clustering* and *Partitioning*.

Clustering

Array clustering, first proposed in [73], is the process whereby arrays are grouped together, with each cluster/group being assigned to a memory block with a single address space. The ASSASYN tool described in [74] uses two clustering (or concatenation) approaches. *Vertical concatenation* is the mapping of the data words from multiple arrays to different data words of a grouped array. Vertical concatenation limits scheduling in that the number of simultaneous accesses to all vertically concatenated arrays must not exceed the number of ports on the memory component chosen to store the array group. *Horizontal concatenation* is the mapping of data words from multiple to a single, longer data word of a grouped array. Horizontal concatenation limits scheduling in the same manner as vertical concatenation. The exception to this is when accesses to the concatenated arrays are to the same array index. In this case accesses to all arrays can be performed at the same time through a single port.

Earlier work on which the ASSASYN tool is based also proposes *time multiplexing* [75], where data words from multiple arrays are mapped to the same element of a grouped array. Obviously the lifetimes of the grouped arrays may not overlap, placing constraints on the schedule for when array data may be created or when array data must

be consumed. In [75] the lifetime of each array is determined based on the lexical scope of the array declaration in the behavioural specification. This is quite a crude method and more advanced methods based on data flow analysis are used in [63] to determine the lifetimes of array elements.

The MeSa tool [73] clusters arrays using only vertical concatenation. It starts with a one-to-one array to memory mapping and clusters arrays until there are no arrays left to cluster (only one grouped array) or until a user set scheduling constraint is broken. MeSa targets only one memory technology and assumed all memory to be on-chip. The MemPacker tool [76] supports different memory technologies but must be supplied with a schedule for the target algorithm. It will then minimise the memory cost within this schedule. The ASSASYN tool uses both vertical and horizontal concatenation, though not time multiplexing as suggested in earlier work. For each array grouping it binds the grouped arrays to candidate memory components and schedules the memory accesses. The tool chooses the best array to memory mapping using simulated annealing [77] with a cost function based on the cost of the memory components used, the schedule length and the number of wires necessary to connect the memory to the data path.

An array clustering algorithm to maximise the speed of hardware pipelines in reconfigurable hardware is presented in [78]. The Napa C compiler [79], into which this algorithm is integrated, targets only loops with no loop carried dependencies for pipelining. Furthermore, it assumes that the memory ports for accessing data are the only resource constraints. As a result, the execution time for each pipelined loop in the target algorithm is determined by the number of clock cycles required to execute all the memory accesses in a single iteration of each loop body. The compiler assigns the data to the available memories so that this cost is minimised across all pipelined loops. Non-pipelined loops do not seem to be considered. The size of the memories is not considered and it is assumed that the arrays assigned to a given memory can be accommodated.

The Napa C clustering algorithm proceeds by enumerating all possible data to memory assignments. The compiler uses implicit enumeration to reduce the search space and all equivalent assignments are compressed to a single array to memory assignment.

Equivalent assignments are common in cases where all of the system memories are homogeneous (have the same access properties). For example, consider the case with two arrays, **a** and **b** (which may be of different sizes), and two homogeneous memories, M_1 and M_2. Assigning **a** to M_1 and **b** to M_2 is equivalent to assigning **a** to M_2 and **b** to M_1. Likewise, assigning both arrays to M_1 is equivalent to assigning both arrays to M_2. This gives only two possible array to memory assignments, as opposed to four in the case when the memories are not homogeneous.

A more advanced array clustering algorithm for maximising pipeline speed is presented in [36]. This algorithm is used to optimise results achieved using the Pipeline Vectorization methodology in the SPC tool [27]. The methodology presented advances on the Napa C approach by considering the sizes of the memories resources, array aliasing (where an access in the algorithm may be to one array or another, but not both at once) and flexible memory word lengths. Memory components may be configured to have 8 bit, 16 bit or 32 bit word lengths. The number of words in the memory is adjusted based on the word length selected. Integer Linear Programming (ILP) [80] is used to solve the system of constraints to find the optimal array to memory placement so that the pipeline speed is maximised.

Partitioning

Clustering is useful when there are small arrays (relative to the size of the available memory components) and the accesses to these arrays are not critical to system performance. However, many real applications feature nested loops which manipulate large, multi-dimensional arrays. Often the rate at which this data can be accessed will determine the throughput of the system, especially in FPGAs where there is an abundance of logic resources with which to implement high levels of parallelism. Simply placing an entire array in a single memory block/component will typically limit the bandwidth for that array to one or two accesses per cycles (most block RAMs are single or dual port). In most cases this will limit the parallelism that can be exploited in the system.

Modern FPGAs contain large numbers of embedded, configurable RAM blocks.

For example, a mid range Altera Stratix II FPGA (EP2S60) offers 329 512bit RAMs, 255 4kbit RAMs and 2 512kbit RAMs. FPGA based systems also typically feature more than one bank of off-chip memory. The Celoxica RCHTX board [81], for example, has four banks of SRAM connected to a single FPGA. If arrays that are frequently accessed can be split across these multiple physical memories in such a way that parallel loop iterations, or different accesses within the same loop iteration, will always accesses different memories, then these accesses can be executed in parallel. This can, in turn, allow higher levels of parallel execution and increase system throughput. The splitting of array data across multiple physical memories is referred to as partitioning.

The idea of array partitioning for FPGAs has been explored in some depth at the University of Southern California. In [82] *scalar replacement* is considered to reduce the latencies of critical (longest) paths through a DFG. Scalar replacement involves targeting arrays, or sections of them, to registers for storage. When an array is targeted to registers, all of the elements in the array may be accessed at once, potentially allowing much higher levels of parallelism to be exploited. The accesses latency is also reduced from a minimum of one clock cycle to zero clock cycles. This can reduce the critical path through a DFG, but only if the correct combinations of accesses are moved to registers. Consider the sample DFG shown in Figure 2.16(a). Moving either the a[k] or the b[k][j] accesses to registers will not reduce the critical path, both must be moved. To deal with this, the concepts of a *critical graph* (CG) and a *cut* in the CG are introduced. The critical graph is a subgraph of the DFG such that all of the critical paths in the DFG are included. A *cut* is defined as a minimal subset of array access nodes, such that their removal would bisect all the critical path in the CG. Figure 2.16(b) shows the CG for the DFG in Figure 2.16(a), including all possible cuts.

To find the best set of array elements for scalar replacement, the algorithm presented in [82] iteratively applies scalar replacement to the cut in the CG which requires the fewest registers to implement. The process iterates until all registers are used or no cuts remain. While scalar replacement can reduce the clock cycles required to execute an algorithm, it can require large numbers of registers if the arrays being considered are

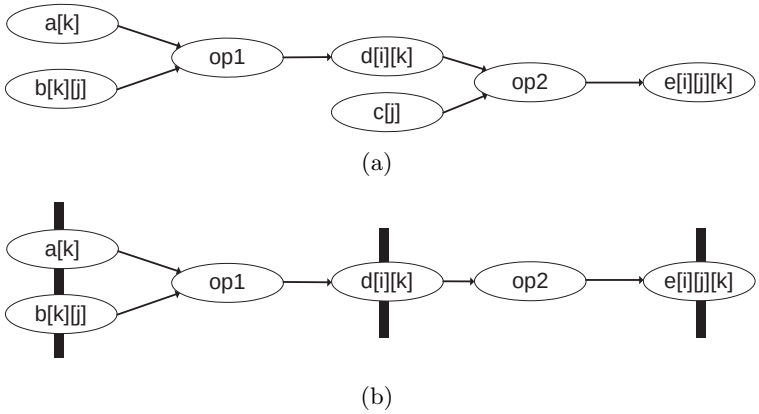


Figure 2.16: (a) A typical DFG. op1 and op2 represent arithmetic operations, all other nodes represent memory accesses. (b) The critical graph for the given DFG. The black vertical lines shows the possible cuts, which are {a,b}, {d} and {e}.

large (in [82] a limit on the number of registers that can be used is set by the user). Also, using registers to store arrays (or sections of arrays) can lead to complex routing, wide multiplexors, long critical timing paths and lower clock frequencies. Any drop in clock frequency must be traded against any reduction in the cycle count for the algorithm.

Scalar replacement for large arrays is often impractical and so coarser grained partitions across multiple physical memories must be considered. The Raw compiler [83] uses *modulo unrolling* to order array data across multiple memory banks. This method is applicable where a multi-dimensional array is accessed in a nested loop. Array elements are partitioned along the dimension which varies most quickly in accesses in the loop nest. Consecutive accesses are placed in neighbouring memory banks as demonstrated for the sample code in Figure 2.17(a). The modulo unrolled placement in Figure 2.17(c) allows 2 accesses per cycle, twice that available using the naive placement of Figure 2.17(b). However, as shown in [84], modulo unrolling does not yield the best array to memory mapping in every case. Figure 2.17(d) shows an array to memory mapping such that 4 accesses may be executed in parallel, the maximum available through the four banks of RAM. In [84] the precise dependencies between the multiple accesses to a given array in a loop are considered. For array accesses which are affine functions of the loop index variables, greatest common divisor (gcd) based tests can be applied to determine whether two array references may ever access the same array element. *Virtual memories* are then created for

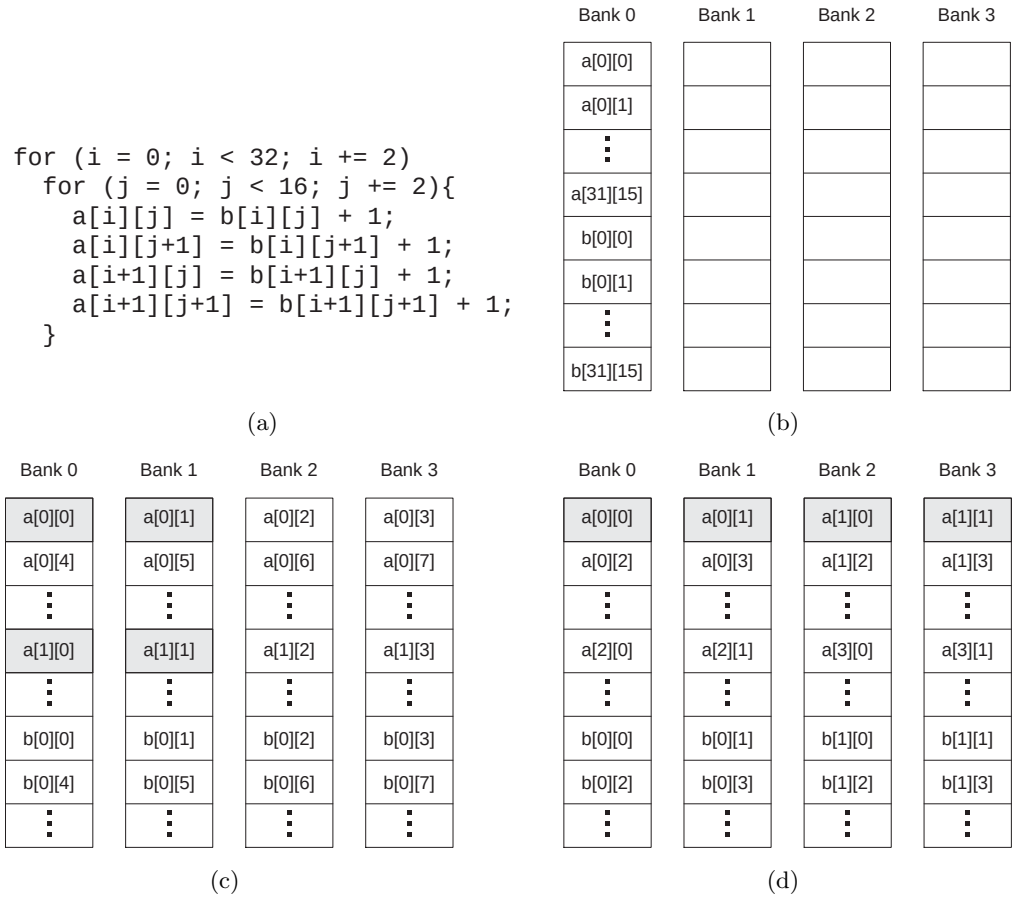


Figure 2.17: Array Partitioning to maximise performance. (a) The target loop (b) A naive data layout allowing one array access per cycle (c) The modulo unrolled layout. The shaded elements must be written in the first loop iteration. Only two of the four writes may be implemented in parallel (d) The optimal layout. All four writes may now be executed in parallel.

each mutually exclusive set of array accesses. These virtual memories are then mapped to the physical memories in the system so that the number of accesses to each memory per loop iteration is minimised. This final stage involves the vertical concatenation of virtual memories, as described in the previous section. Using the methods presented in [84] the optimal array to memory mapping in Figure 2.17(d) can be found.

A simpler method for array partitioning, developed at the University of Santa Barbara, is presented in [85]. This method is based on the premise that a good partitioning can often be found along one of the dimensions of an array *e.g.* a two dimensional array can be partitioned into groups of rows or columns. Each partitioning direction is considered for all possible levels of granularity (number of rows or columns per memory block). While

this method is relatively simple and may achieve good results in practical implementations, with speedups of between 2x and 46x achieved for a small set of DSP and image processing applications tested in [85], it cannot find more complex partitions, such as that used in Figure 2.17(d).

The most recent work at the University of Southern California in this area [86] has combined the scalar replacement methods presented in [82] with the partitioning methods of [84]. Array replication is also considered, where multiple copies of arrays may be generated to supply concurrent accesses to the original array in the target algorithm. An algorithm similar to that used for just scalar replacement in [82] is applied to place the array data in the available memories and registers so that the lengths of all critical paths in the DFG are minimised.

2.5 Research Hardware Compilers

There is a relatively large number of research compilers documented that convert a high level language into hardware. These include the Bach-C [87], Streams C [88], Haydn-C [89], SPC [64], Napa C [79], Sea Cucumber [90], ROCCC [9] and SPARK [17] tools. The most common input language used by these compilers is some form of C (or C++) variant, though [90] uses Java. One of the goals of high level synthesis is to allow an easy transition from existing software (possibly a software model of the system) to hardware. Languages such as C are widely used and well understood by most programmers. Hence, C is most commonly chosen as the input language. Most of the compilers produce either VHDL or Verilog as their output [9, 17, 87, 91, 92], while others synthesize to a final programming bitstream or gate netlist [79, 93–95]. The work in [89] targets Handel-C which is a commercial higher level language (above VHDL and Verilog). When targeting FPGAs there are a number of commercial compilers that can create efficient FPGA hardware (often device specific) from VHDL or Verilog [96, 97], and Handel-C [98]. The FPGA vendors have produced tools to map and place and route the results from these compilers onto their chips [99, 100]. These are all commercial products and significant development effort has been expended to ensure they produce reliable, good final implementations. As

a result, when targeting FPGAs, it is often most efficient to output some form of HDL code (be it VHDL, Verilog or even Handel-C) rather than a final bitstream.

In the existing work there appear to be three general templates for design entry. The first is where the algorithm is simply defined using the input language as if it were standard, sequential software, though there may be some restrictions on recursion, pointers and/or file handling. This represents the highest level of abstraction as no knowledge beyond software is required. This format is used in [9, 64, 92, 101]. The second design entry format is the most complex and also seems to offer the lowest level of abstraction. The standard input language syntax is used but the algorithm must be written to a specific, hardware oriented template. An example of this method can be found in [95] where the C++ class structure is used to define 'machines', which are analogous to VHDL processes or components. This format seems counter productive since one of the goals of high level synthesis is to allow hardware design using standard software programming models. If a new programming model must be used, it may be almost as easy to simply convert to VHDL or Verilog. The third format for high level design entry uses the input language syntax, often with some restrictions (usually on recursion and pointers) and with some extensions [79, 87, 91, 93, 102]. The code is written like normal sequential software, but the user will define parallelism using language extensions. Other hardware issues such as resource sharing, custom integer word lengths and inter thread communication can be defined using these language extensions.

Not all behavioural code is amenable to acceleration in FPGAs. Typically FPGAs can accelerate frequently reused sections of code, such as loops, by exploiting the opportunities for parallel execution within them. Inherently sequential code can often be implemented more efficiently on a microprocessor. For this reason it is common for applications to be partitioned between custom logic and (one or more) microprocessors when both resources are available. Hence, a number of the compilers were written with hardware/software co-design in mind. They allow sections of the complete algorithm to be targeted to hardware while others are targeted to software, often running on embedded processors within the same device as the custom logic [9, 79, 88, 101]. The partitions may

be specified manually by the designer using compiler directives or '#pragma' statements in the source code [88], or determined through profiling as in [9,27]. The conversion from behavioural code to a description that may be synthesised is automated, as is the generation of the interface between the processor and the custom logic. This allows the designer to explore different partitions much more quickly since most of the time and effort in changing a partition normally goes into generating the new HDL code. Synthesis directly from the behavioural source allows the designer to focus on the high level details of the design, exploring numerous possible partitions or interface schemes. The normally time consuming low level details are dealt with automatically.

One common theme through a number of the compilers investigated is the use of channels to allow communication between coarse grained parallel units (threads) of the algorithm [87,88], similar to those presented in the Communicating Sequential Processes (CSP) framework [103]. CSP style channels essentially allow communications between parallel hardware blocks to be described without considering how each process will be scheduled. A process will hold at a communication point until the process it must communicate with is ready to send/receive data. To reduce unnecessary stalling due to processes trying to send data before it can be received, the compilers break from the strict CSP protocol and allow the use of buffered channels. The advantage of using channels, either buffered or unbuffered, is that coarse grained parallelism can be described at a behavioural level, without consideration for where each process may be implemented (in hardware or in software) and how each process may be scheduled. The disadvantage of channels is that they require extra resources to implement the control and buffer structures.

Significant transformation and optimisation is generally required to convert from a behavioural description to a hardware description and in the existing compilers numerous methods are employed. The methods used can be separated into five general families which appear critical for converting software into efficient hardware. The first family of optimizations groups together standard 'software' methods, those typically carried out by good software compilers. These transformations include constant folding, dead code elimination, strength reductions (replacing expensive operations such as multiplies with

shifts or adds) and common subexpression elimination. Various combinations of these transformations are featured in [9, 17, 101, 102, 104]. The second family of optimizations involves hardware specific, lower level optimizations such as micro-pipelining (inserting extra registers to reduce critical paths) [91,92,101], multiplexor reductions [101], splitting wide integer operations into chains of narrower (smaller word length) operations [92] and register retiming [91]. These are all useful for increasing the maximum clock frequency of the final design.

The third family of optimizations features fine grained scheduling operations. These include percolation, trailblazing, list scheduling, speculative code motions and resource sharing/resource binding, combinations of which which are used in most compilers, including [17, 87, 89, 101]. A number of enabling transformations are often applied before scheduling to improve the results. These include conversion to static single assignment form [16] and IF-conversion [19]. The SPARK compiler tool [17] features a comprehensive 'toolbox' of scheduling transformations that may be turned on or off to allow their effects and interactions to be investigated.

The fourth family of optimisation methods is loop transformations. These are important as it is often loops that account for most of the execution time of algorithms. Furthermore, loops are best suited to being accelerated in hardware as they can often be parallelized by some method and see the same resources used repeatedly for the same operations. Perhaps the most important loop transformation implemented is loop pipelining, which is automated in most existing compilers [9, 64, 79, 89, 92, 93, 101, 104]. This transformation (or scheduling operation) is key to the efficient implementation of loops as it allows resource utilisation to remain high without having to multiplex the inputs of the functional units between numerous operations (lines of behavioural code). Other loop transformations are considered in a number of compilers, but these are mainly used to support loop pipelining. For example, the Pipeline Vectorization work in [27] only considerers the innermost loop in a loop nest for pipelining. However, loop unrolling, loop tiling, loop merging and loop interchange are all considered to make the innermost loop better suited to pipelining. Likewise, literature on the ROCCC compiler [9] mentions the

use of loop unrolling, loop merging and strip mining.

The fifth and final family of transformations deals with the optimization of the memory structure, both on and off chip. Two compiler tools automatically exploit data reuse in the target algorithm. The ROCCC compiler [9] considers generation of 'smart buffers' to reuse windowed data, based around the methods described in [66]. The SPC tool includes shift registers and on-chip buffers to reuse input data where possible [36]. A number of the compiler tools also mention automated mapping of array data to RAMs and ROMs, but only two of the tools describe methods of mapping arrays to system memories (on or off chip) to meet a specific goal [36, 78]. In both cases the goal of the array to memory mapping is to distribute the data so that accesses in loop pipelines may be executed in the minimum time, thus maximising pipeline speed.

2.6 Commercial Design Tools

Most commercial design tools center around VHDL and Verilog synthesis, with these languages remaining the standard method for design entry when targeting FPGAs. Both Altera and Xilinx include VHDL and Verilog synthesis tools as standard in their development environments (Quartus II [100] and ISE [99] respectively). The core synthesis tool for most of the leading EDA software providers, including Synplicity [96], Mentor Graphics [97] and Synopsys [105], is still VHDL and Verilog based. However, while these tools do not offer behavioural synthesis, they do feature relatively advanced tools for lower level optimisations and are good at converting RTL level descriptions into efficient FPGA hardware. They can infer various memory structures from VHDL or Verilog code, though it may need to be written according to some style rules [100], remove logic which has no effect on output pins, and perform register retiming (including register duplication) in order to maximise the clock frequency achieved. This is useful when developing higher level tools for behavioural synthesis as the tool need only generate generic VHDL or Verliog as an output and the user can be reasonably confident that this will be optimised by the RTL level synthesis and place and route tools.

While the core industrial tools remain HDL based, there are an increasing number of products available offering new levels of abstraction. Altera, Xilinx and Synplicity have released DSP design tools that allow 'synchronous data flow' applications [106] to be to be developed at the block flow diagram level through Mathworks MATLAB and Simulink environments. Altera offers DSP Builder [107], Xilinx offers System Generator [108] and Synplicity offers Synplify DSP [109]. While these tools allow DSP hardware to be defined in environments more familiar to DSP algorithm designers and do raise the level of abstraction, they rely on large libraries of manually optimised, parameterised libraries, and are only suitable for synchronous data flow applications with streaming data inputs and outputs. Because the order of data input and output is fixed the scheduling options are reduced and the only consideration is maximising throughput.

A number of C based design tools are also available. Agility DK Suite [98] compiles the Handel-C language [110] to produce either VHDL/Verilog or EDIF hardware descriptions. Handel-C is not behavioural as there is a strict, implicit timing model such that every assignment statement executes in a single clock cycle. All instruction parallelism and resource sharing must be explicitly specified using language extensions. Embedded memories must also be specified explicitly, though they may still be referenced as if they were simple arrays (there is no need to explicitly assign address and read/write control signals). Arbitrary integer word lengths may be specified for variables and the widths for some internal signals/variables may sometimes be inferred by the compiler. A channel framework, similar to CSP, can be used to communicate between functions with different clocks (or the same clock if there is uncertainty in the length of the schedule).

Handel-C offers an increase in abstraction above VHDL and Verilog as the control flow for the design is the same as that for software — the instructions execute in the order they appear in the code[7]. State machines are generated automatically to provide the equivalent hardware implementation. This makes it easier to keep track of the order of operations, and the software like programming model and syntax make it easier to port existing software to hardware. However, the code must usually be comprehen-

[7]Except where they are enclosed in braces preceded by a 'par' tag, in which case they execute in parallel.

sively restructured to suit the hardware model in order to obtain efficient hardware. For example, loops should usually be pipelined, unrolled, tiled, or some combination of loop transformations applied to improve resource usage and reduce execution time. Some low level optimisations such as register retiming and common subexpression elimination are also automated to improve results.

Mentor Graphics Catapult-C Synthesis tool [8] takes a different approach to that of DK Suite and Handel-C. It accepts pure, untimed C++ as its input and compiles to optimised VHDL or Verilog. The scheduling and resource sharing are automated, as are RAM/ROM insertion and the implementation of interface specifications. Various loop transformations such as pipelining, unrolling, merging and tiling are also automated. However, the Catapult-C Synthesis tool is not completely automated in that all of the higher level design decisions are still made by the designer through constraint files (altered through a 'push button' GUI). The user selects the interface scheme, makes the hierarchy decisions, selects which transformations to apply to which loops and which arrays to map to memories. The compiler then implements all of the low level details required to produce the corresponding hardware.

Since the design constraints are defined in a separate file, the same source code can be used to produce many different implementations, depending on the constraints. The tool then displays the area/speed/power characteristics for each design, allowing the user to tradeoff the pros and cons of each high level implementation decision. Because the normally time consuming low level implementation details are automated (and therefore accelerated), the designer has more time to focus on the high level design decisions that can ultimately have a much larger effect on the final system performance. A number of tools similar to the Catapult-C Synthesis tool are also available, including including Forte Design Cynthesizer [111], Accelerated Technology Impulse C [112] and Synfora Pico Express [113], but they are not discussed in detail as there is less documentation of their performance available.

A number of commercial tools accept inputs descriptions written in System C. System C uses standard C++ syntax, extended to include timing information using an

extensive class library [114]. Hardware modules are written as classes, with each instance of the class created in the code akin to instantiating a Verilog Module or VHDL Entity. A System C module class will generally feature one or more System C methods or System C threads to implement the functionality of the intended hardware. A System C method is the same as a standard C/C++ function, except it has an associated sensitivity list and the function runs once every time one of the signals in the list changes value. A System C thread is a function that runs continuously, but which contains wait statements that hold its execution a a given point until a change in a signal in the sensitivity list. The instances of each module class execute in parallel, linked via interface classes which describe how the ports of each module communicate (*e.g.* through shared memories, FIFOs, CSP style channels).

The separation of the implementations of the algorithms and interfaces in this manner allows the designer to focus on the two components of the design independently, allowing greater freedom to explore the options in each component without having the rewrite the whole design. System C offers other advantages over standard RTL languages since it can be compiled and simulated by ordinary C++ compilers, provided the open source System C library has been downloaded and included. It also allows custom hardware, software for associated microprocessors and high level testbenches to be written in the same language, enabling the simulation of entire systems with a single tool.

The Agility SC compiler [115] produces RTL VHDL/Verilog or EDIF from System C input descriptions. It can perform optimisations such as pipelining, retiming and branch balancing, and can share arithmetic resources across multiple operations if they are not scheduled to execute on the same clock cycle. It can also infer word minimum word lengths for variables in the design. The Forte Cynthesizer tool [111] also accepts System C input. The user specifies the desired throughput and clock frequency for streaming algorithms and it performs pipelining and loop optimisations such as unrolling to schedule the design to meet the requirements. It can also generate RTL interface descriptions for standard interface protocols.

The Xilinx Accel DSP tool [116] transforms MATLAB M-files into RTL descrip-

tions. MATLAB code is untimed so the tool schedules the operations, pipelining loops and matrix and vector operations where possible. The implementation of loop pipelining and other loop transformations such as unrolling is automated, but the user must specify which transformations are applied. The user may also select points at which extra pipeline stages should be inserted to improve the clock frequency. Accel DSP also automatically converts the standard floating point variables used in MATLAB code into fixed word length integer values using quantization functions to minimise the number of bits required for each variable.

Altera have released a behavioural synthesis tool, called the C2H Compiler [117], which is aimed at supporting hardware accelerators for embedded FPGA designs using the Altera Nios II soft processor core [118]. The C2H compiler can convert a single C function (which may call other functions) within an application into a hardware coprocessor to support the Nios II core. As with all behavioural compilers, it will only produce a good hardware implementation if the target function is well suited to hardware (usually based around loops). The C2H compiler does automatically pipeline loops and will automatically pipeline memory accesses where possible. The generation of the interface with the Nios II processor is also automated – all the user has to do is select the target function and start the compiler. The original C program is automatically modified to use the resulting hardware accelerator.

The C2H Compiler has a fixed set of rules for generating hardware from C code, and the structure of the input C impacts on the efficiency of the resulting hardware. Where the target function references arrays or variables with global scope (beyond the accelerated function), it must access the main system memories through the main system bus. To this end, a bus master port is generated for every such access in the target function. Little resource sharing is considered. On the whole, each operator in the C code compiles to a dedicated resource. However, if a sub-function is called multiple times in the target function then only one instance of the sub-function is generated. This is the same as the approach taken in Handel-C. The designer can therefore force resource sharing by enclosing operations in a sub-function.

As a stand alone tool the C2H Compiler is of limited use as it generates only relatively simple hardware from a single C function. However, combined integrated into the rest of the Nios II tool chain (with SOPC Builder [45] and the Nios II IDE [119]), it allows entire embedded systems to be implemented in FPGAs without the need for any VHDL or Verilog to be written.

2.7 Optimisation Methods

This thesis investigates methods for producing optimised schedules and memory subsystems for FPGA implementations of algorithms. A number of optimisation approaches have been explored in existing work on scheduling and memory design, including *greedy algorithms* [120], *simulated annealing* [121] and *linear programming* [80]. All of three approaches seek to assign values to a set of variables within a given set of constraints such that a given cost function is minimised (or maximised). The user must formulate the variables, constraints and cost function to model the target problem. Greedy algorithms and simulated annealing use heuristics to guide the search for the solution with the lowest cost function[8] and so may not always find the globally optimal solution. Linear programming methods will always find the optimal solution (assuming a feasible solution exists), but these are more computationally expensive than heuristic methods and large problems may not always complete in suitable length of time or on a given set of computing resources. Heuristic methods may be tailored to produce a 'sufficient' solution for a given run time or computing platform.

2.7.1 Greedy Algorithms

Greedy algorithms iterate over points in the available search space, evaluating the cost function at each point visited before moving to the next candidate solution. The selection of the next candidate point is based only on the current local data, *e.g.* the value of the cost function at the neighbouring points. A common approach is to use hill descent,

[8] From this point on we will assume that the goal is to minimise the cost function, but the methods are equally applicable to maximising a cost function.

where the algorithm moves in the direction with the highest negative rate of change in the cost function to find the next candidate point. The algorithm terminates when the maximum negative gradient reaches zero and a minimum is reached. Because only local data is used to determine the next move, greedy algorithms can easily become trapped in a local minimum, unable to move to a more favourable solution elsewhere in the search space as this would require first moving to a point with a higher cost.

2.7.2 Simulated Annealing

Simulated annealing builds upon greedy algorithms by providing a mechanism for breaking out of local minima. While greedy algorithms can only move to points in the search space with a lower cost than the current point, simulated annealing allows the search to 'jump' to a solution with a higher cost. Whether the search jumps to a point with a higher cost or continues moving towards a lower cost is controlled by a probability, which decreases as the search progresses. The higher the probability, the more likely a move to a higher cost point. The rate at which the probability decreases is referred to as the cooling schedule and is set by the user. While simulated annealing does offer the opportunity for the search to break out of local minima, it still does not guarantee that the optimal solution will be found. Generally the longer (slower) the cooling schedule the greater the chance of finding the optimal. However, run time will generally increase as the cooling schedule is length of the cooling schedule is increased.

2.7.3 Linear Programming

Mathematical programming is a branch of mathematics dedicated to solving optimisation problems of the form: *maximise* $f(\mathbf{x})$ *subject to* $g(\mathbf{x}) \leq 0$, where $\mathbf{x}$ is a vector of variables. Linear programming restricts the cost function, f, and the constraints, g, to be linear functions of the $\mathbf{x}$ vector. Rather than providing 'best known' solutions as with heuristic methods, linear programming is a method for formally optimising problems, providing globally optimum solutions. If a problem can be formulated in linear form there are significant advantages in terms of execution time over higher order mathematical pro-

gramming techniques. It is a well known result in linear programming that, if all the variables are real valued, the linear constraints will produce a solution space that is a convex polyhedron. The optimal solution will exist at one of the vertices of the polyhedron, at the intersection of the planes described by each row of the constraint matrix. The well known Simplex Algorithm [122] takes advantage of this property to solve linear programs, determining the optimal solution in a time that grows exponentially in the worst case with the number of variables in the problem, but typically performs much better than this in practice.

While it is desirable to formulate problems in linear form where possible, it is often necessary to introduce integer variables to produce an accurate mathematical model. Mixed integer linear programming (MILP) uses a combination of integer and real variables, while integer linear programming (ILP) uses only integer variables. Throughout the remainder of this thesis both MILP and ILP problems are referred to as ILP for simplicity. Integer based problems are more difficult to solve than standard linear problems because the optimal solution is not guaranteed to lie at the intersection of the constraint planes. As a result the solution of ILP problems requires further methods beyond those used to solve linear programs, and branch and bound algorithms [122] are widely used for this. In branching the problem's solution space is recursively split into smaller subregions to form a search tree. Algorithms are then used to place lower limits on the value of the cost function in each subregion, bounding the search and eventually allowing the optimal solution to be found. Typically integer relaxation is used to during branch and bound, where the values of increasing numbers of the integer variables are fixed while the remaining variables may take non-integer values. The resulting problem may be solved efficiently as a simple linear program, giving lower bounds for the given subregion of the problem. Despite the advanced branch and bound algorithms used in commercial ILP packages such as ILOG CPLEX [123], the worst case time complexity for solving ILP programs grows exponentially with the number of variables. This can potentially render large problems unsolvable with any given computing resources.

2.8 Summary

There is a large body of existing work on compilers and high level synthesis. Lower level issues such as dependence checking and basic compiler optimisations such as common sub-expression elimination, constant folding and the scheduling of sequential code have been examined in detail (in [17] for example) and are well understood. Loop pipelining has also received a great deal of attention for both the software and hardware domains, and is the major focus of a number of hardware compilers [9, 27]. However, while pipelining above the innermost loop level has been considered for software it has not yet been applied in any hardware compilers. In fact, there is little work on implementing parallelism in hardware above the innermost loop. This represents one possible area for investigation, which is taken up in Chapter 4.

A wide range of memory optimisation techniques have been presented in existing work, some of which have not yet been applied to FPGAs. The reuse of data along input dependences is considered in [36, 65], but no methods are presented for selecting the optimum choice. Array to memory mapping techniques for minimising pipeline initiation intervals are presented in [36, 78], and partitioning methods for minimising schedule latencies are presented in [84], but no combined methodology has yet been presented to combine all of these optimisations and link them to scheduling to minimise the execution time of a pipelined loop nest.

Chapter 3

LINPACK 1000: A Case Study

3.1 Introduction

In the previous chapter a range of existing compilation and optimisation techniques for conversion from software descriptions to hardware were reviewed. In this chapter the manual conversion of an algorithm from software to FPGA hardware is undertaken, both to provide some practical insight into the application and effectiveness of the existing techniques, and to expose potential aspects of the conversion process for which no formal methods have been presented.

Recall that the focus of this thesis is the development of methods that maximise the parallel execution (performance) of the FPGA implementation for the target algorithm, rather than the minimisation of the resource usage for a given performance target. The performance of the memory subsystem has been identified in the existing literature as the performance bottleneck in most FPGA based systems [36, 65], and the optimisation of the memory subsystem has been identified as a potential area for extensions beyond the existing work. With these issues in mind, the LINPACK 1000 benchmark has been chosen as the target algorithm for this case study since it requires an implementation with maximum parallelism, and also because it is dominated by accesses to the memory subsystem. Furthermore, there has been a recent surge in interest in using FPGAs to accelerate algorithms in the scientific computing domain [124]. By targeting the LIN-

PACK 1000 benchmark, which is taken to be indicative of typical scientific computing algorithms [125], it is hoped that some optimisation techniques more specific to scientific computing kernels can be identified.

In this case study the Agility Handel-C language [126] is used to describe the FPGA hardware generated. Handel-C was chosen as it can allow FPGA designs to be produced more quickly than with Register Transfer Level (RTL) languages such as VHDL or Verilog since it offers a slightly higher level of abstraction. Also, the Handel-C simulator within Agility's DK Design Suite [98] can allow much faster simulation that can be achieved with RTL simulators such as ModelSim [127]. Furthermore, the C-like syntax used in Handel-C can potentially allow a more source-to-source style of transformation from software to hardware, with the optimisation steps applied in sequence. After each step the design may still be simulated to ensure correctness.

The remainder of this chapter is set out as follows: Section 3.2 provides an overview of the LINPACK 1000 benchmark, followed in Section 3.3 by a description of the target platform used in this work. Sections 3.4 describes the steps taken in the conversion of LINPACK 1000 from a description in C to a hardware description in Handel-C. The results achieved when the hardware description is targeted to an Altera Stratix II FPGA are presented in Section 3.5.

3.2 LINPACK 1000

LINPACK is an established floating point benchmark traditionally used to evaluate the performance of supercomputers [125]. The Top 500 list [128] still ranks supercomputers based on their sustained (average) floating point performance during the execution of the LINPACK benchmark, almost 30 years after the benchmark was introduced. The LINPACK benchmark solves a system of linear equations, $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is an NxN matrix[1], and $\mathbf{b}$ and $\mathbf{x}$ are N element vectors. Double precision floating point representation is used for the numbers in $\mathbf{A}$, $\mathbf{b}$ and $\mathbf{x}$. The values of $\mathbf{A}$ and $\mathbf{b}$ are known and the value of $\mathbf{x}$ is determined during the benchmark.

[1] N is an integer

The LINPACK benchmark is actually made up of three benchmarks, with performance reported for 3 different values of N. The most important benchmark for modern, highly parallel supercomputers is the Highly Parallel Computing version [125], where N may be varied to any value to achieve the best performance. It is the performance for this version of the benchmark that is reported in the Top 500 list. The other two versions of the benchmark use fixed values of N – 100 and 1000 respectively. In this work the focus is the $N = 1000$ version, referred to as LINPACK 1000. The small size of the problem solved in LINPACK 1000 relative to the scale and capabilities of modern, highly parallel supercomputers makes it ill suited to test the top end performance of such machines. However, LINPACK 1000 may still be used to benchmark the performance of single core CPU based systems, and provides sufficient computation density to easily fill all the resources on a single, large FPGA. As such it is reasonably well suited to test the floating point performance of FPGAs. The version of the benchmark with $N = 100$ would also provide sufficient computation density to fully utilise the resources on an FPGA, but it is small enough to store all of the data used in the on-chip memories and, as such, is not necessarily a realistic representation of scientific applications on the whole. The 1000x1000 element double precision floating point matrix used in LINPACK 1000 is sufficiently large to require off-chip storage and so this problem does not arise.

LINPACK 1000 solves a random linear system of order 1000, measures the average floating point performance achieved and reports the residual error produced. The C code for the operation of the LINPACK 1000 benchmark is given in Figure 3.1. The benchmark generates a random 1000x1000 element matrix, $\mathbf{A}$, and 1000 element vector, $\mathbf{b}$. The *dgefa* and *dgesl* subroutines are then used to find a 1000 element vector, $\mathbf{x}$, such that $\mathbf{Ax} = \mathbf{b}$. The *dgefa* subroutine performs LU decomposition by Gaussian elimination with partial pivoting on $\mathbf{A}$. A description of this method can be found in Appendix 4 of [103]. *dgefa* modifies $\mathbf{A}$ and returns a vector, **ipvt**, containing the pivot indices. The *dgesl* subroutine then solves the simplified LU version of the original system. The time taken to complete the *dgefa* and *dgesl* subroutines is measured and the average floating point operations per second (FLOPS) calculated. The rest of the algorithm estimates the normalised residual error in $\mathbf{x}$ due to the finite precision of the floating point number representation.

```
void LINPACK_1000()
{
  double A[1000][1000], b[1000], x[1000];
  double resid, residn, flops, t1, ops;
  int ipvt[1000], i;

  // generate random linear system Ax = b
  matgen(&A, &b);

  // solve system using LU decompositon and record time taken
  t1 = second();
  dgefa(&A, &ipvt);
  dgesl (&A, &b, &ipvt);
  t1 = second() - t1;

  // result for x is returned in b vector by dgesl and must
  // be moved into x
  for  (i = 0; i < 1000; i++)
      x[i] = b[i];

  // calculate average FLOPS
  ops = (2.0/3.0)*pow(1000.0, 3.0) + pow(1000.0, 2.0);
  flops = ops / t1;

  // regenerate original A & b and calculate b = Ax - b
  matgen(&A, &b);
  for  (i = 0; i < 1000; i++)
      b[i] = -b[i];
  dmxpy(&A, &b, &x);

  // find residual and normalise
  resid = 0.0;
  for  (i = 0; i < 1000; i++)
      if (fabs(b[i] > resid)
            resid = b[i];
  residn = normalise(resid);

  //print results
  printf("Time = %f\n", t1);
  printf("FLOPS = %f\n", flops);
  printf("Residual = %f\n", resid);
  printf("Normalised residual = %f\n", residn);
}
```

Figure 3.1: C Code for LINPACK 1000.

The goal of this piece of work is to manually generate an FPGA accelerator for the LINPACK benchmark from the existing C code. To evaluate the success of the methods used in the conversion, the performance of a high end CPU is used as a baseline comparison. A pure ANSI C implementation of the benchmark (with no calls to optimised linear algebra libraries) was compiled and profiled using gcc v3.4.2 (with -O3 and loop unrolling optimisations turned on). When executed on a 3.4GHz Intel Pentium 4 processor (1Mbyte L2 cache) running Linux, the benchmark completes in 1.51 seconds at 499 MFLOPS.

The LINPACK benchmark calls a number of basic linear algebra subroutines (BLAS). Some highly optimised BLAS libraries are available which can dramatically increase the performance of linear algebra algorithms. One such library is ATLAS [129] which, when installed on a given platform, tunes its BLAS library to make best use of the host processor's cache size and instruction set extensions. Modifying the benchmark to use functions from the ATLAS LAPACK library (version 3.60) which are equivalent to *dgefa* and *dgesl* yields greatly improved performance. The benchmark completes in 0.31 seconds, providing a sustained performance figure of 2907 MFLOPS.

3.3 Target Platform & Partitioning

This work targets the general platform shown in Figure 3.2. It consists of a host microprocessor system linked to an FPGA based coprocessor via a PCI interface. The coprocessor features a high density FPGA, a PCI interface/controller and on-board memory resources. In this work the on-board memory is assumed to be SRAM with between 4 and 6 banks giving a total of up to 32Mbytes of storage. The data width of the memories is assumed to be 32 bits. Although this may not match any existing development boards the features specified are realistic and similar to those found on a number of boards including the Celoxica RC300 [130] and RC2000 [131].

This target platform has a key characteristic that affects the implementation of the LINPACK 1000 benchmark, namely that the benchmark must be partitioned between the FPGA and the microprocessor to achieve good performance. However, communication

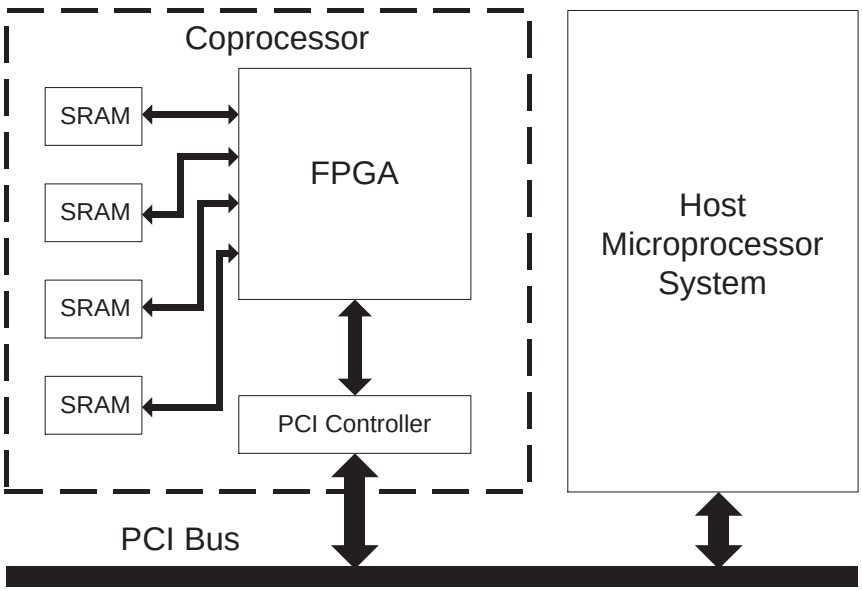


Figure 3.2: Target Platform.

between the microprocessor and the FPGA will be slow compared to accessing the local SRAM. Hence a partition must be chosen that limits the total communication required. The execution of the benchmark on a Pentium 4 processors was profiled and this revealed that 93% of the execution time is consumed by the *dgefa* subroutine. This in turn calls a smaller function, named *daxpy*, 499500 times with the result that *daxpy* accounts for 90% of the execution time.

For simplicity the benchmark is partitioned so that the entire LU decomposition subroutine (called *dgefa*) is implemented on the FPGA co-processor. This partition limits the communication across the PCI bus to two large block transfers of data (totaling 2001000 64-bit transfers), one at the start of the subroutine and the other at the end. It also provides potential opportunities for data reuse and parallelism on the FPGA since *dgefa* contains a nested loop with three levels. Finally, since *dgefa* accounts for 93% of the execution time of the benchmark, there is good potential for acceleration of the complete algorithm.

3.4 Translation and Optimisation

With the benchmark partitioned to implement the *dgefa* subroutine on the FPGA, the next step is to convert the sequential ANSI C code into valid Handel-C syntax. This process is trivial since Handel-C is an extended subset of C, but it is useful to perform this step early on since it allows the algorithm to be tested through the DK Suite simulator [98]. This allows parallel optimisations to be simulated, verified and debugged where necessary.

To convert the *dgefa* subroutine to Handel-C a global clock is declared, all C side effects are replaced by separate assignments and all floating point arithmetic is replaced by double precision floating point macros from Agility's floating point library [132]. Finally, all integer variables are assigned the appropriate word lengths. Integer word length assignment is itself a complex optimisation [133]. However, it has been included as a translation step since, although the Handel-C compiler can infer the widths of some variables, the Handel-C code will often not compile without manual integer width assignment.

Before any aggressive optimisations are considered, two further translation steps are considered. The 'for' loops within the code are converted into 'while' loops and all pointer based data access is replaced by direct variable/array addressing, assuming this is possible. 'While' loops can be implemented more efficiently in Handel-C since the increment of the loop iterator can be explicitly parallelized with the loop body, something not possible with 'for' loops. Converting pointers into direct addressing is useful as it makes it easier to optimise the memory structure in subsequent steps. The resulting code can be compiled as Handel-C and synthesized to hardware. However, the hardware generated will not be very fast or efficient as no parallelism has been exploited and no embedded memories specified. These issues are tackled in the four optimisation stages that aim to:

- Identify and exploit opportunities to store sections of matrix data that are reused in on-chip memories so that parallel access to the matrix data may be increased.
- Pipeline the loops in the algorithm to increase the throughput of the system.
- Exploit opportunities to schedule iterations of the same loop or whole loops to execute in parallel.

- Optimise the low level details of the design using techniques such as common sub-expression elimination [17]

These issues are tackled in the following sub-sections.

3.4.1 Data Reuse Exploration

In this optimisation stage the potential for data reuse exploitation is investigated. The goal is to search for any reuse pattern that could be exploited to reduce external memory accesses or reduce array accesses within loop nests. This could also allow greater parallelism to be exploited in later stages of the implementation by allowing such accesses to be re-written so that they correspond to small, parallel on-chip memories.

The first step is to provisionally assign each array to a layer of the memory hierarchy in the target platform, in this case either off-chip memory, on-chip memory or registers. This allows data reuse to be focused on those arrays stored in memories with low access bandwidth, mainly the off-chip memory in this case. The **A** matrix requires more storage than is available in on-chip memory of any current FPGA and so must be stored in the off-chip SRAM. The **ipvt** vector is small enough to be stored in the on-chip SRAM. Data reuse efforts can now be focused towards the **A** matrix since it is frequently accessed and the available bandwidth through which it can be accessed is relatively low.

The C code for the *dgefa* subroutine is shown in Figure 3.3, along with the C code for the three functions it calls in Figure 3.4. It is possible to reuse (up to) a column of **A** matrix data between iterations of both the outer loop and the middle loop in the *dgefa* subroutine. The *daxpy* function takes two matrix columns as inputs, denoted as **A**[ ][**j**] and **A**[ ][**k**]. The same **A**[ ][**k**] data is used for each iteration of the middle loop in *dgefa* meaning that, if the correct **A**[ ][**k**] data is buffered on the FPGA at the start of the outer loop, it can be reused for every iteration of the middle loop. More importantly, the **A**[ ][**k**] data for iteration $k+1$ of the outer loop is calculated as a result column in iteration k. If this column of data is stored on the FPGA as it is generated then only the **A**[ ][**k**] column for the first iteration of the outer loop will need to be read from the external memory. As a result, inside the loop nest in Figure 3.3, only the **A**[ ][**j**] data must read from the external

```
void dgefa(double *A, int *ipvt)
{
  int k, j, piv;
  double t, temp;

  for (k = 0; k < 999; k++){          //outer loop
    piv = idamax(A, k);
    ipvt[k] = piv;
    if (A[piv][k] != 0){
      temp = A[piv][k];
      A[piv][k] = A[k][k];
      A[k][k] = temp;
      t = -1/(A[k][k]);
      dscal(A, t, k);
      for (j = (k+1); j < 1000; j++){ //middle loop
        temp = A[piv][j];
        A[piv][j] = A[k][j];
        A[k][j] = temp;
        t = A[piv][j];
        daxpy(A, t, k, j);   //inner loop in daxpy
      }
    }
  }
  ipvt[999] =999;
}
```

Figure 3.3: C Code for *dgefa*.

memory instead of the **A**[ ][**k**] and **A**[ ][**j**] data. This effectively halves the number of reads from the external memory and halves the bandwidth required.

This data reuse can be implemented using two new arrays (provisionally assigned to on-chip RAM), each capable of storing one column of matrix data. The first array, referred to as **k_col**, stores the **A**[ ][**k**] data for the current iteration of the outer loop in *dgefa*. The second array, referred to as **k_next**, stores the **A**[ ][**k**] data needed for the next iteration as it is generated. During the *idamax* and *dscal* functions the new **A**[ ][**k**] data stored in **k_next** is modified and transferred across to **k_col**. A block diagram for the FPGA dataflow after data reuse has been implemented is provided in Figure 3.5. Each of the grey boxes represents a memory block, while the white boxes represent functional blocks. The bold arrows represent the flow of the floating point data in the system.

```
int idamax(double *A, int k)
{
  double max = fabs(A[k][k]);
  int piv = k;
  int i;

  for (i = k+1; i < 1000; i++)
    if (fabs(A[i][k]) > max)
    {
      max = fabs(A[i][k]);
      piv = i;
    }
  return piv;
}

void dscal(double *A, double t, int k)
{
  int i;

  for (i = k+1; i < 1000; i++)
    A[i][k] = t*A[i][k];
}

void daxpy(double *A, double t, int k, int j)
{
  int i;

  for (i = k + 1; i < 1000; i++)
    A[i][j] = t*A[i][k] + A[i][j];   //inner loop
}
```

Figure 3.4: C Code for *idamax*, *dscal* and *daxpy*.

3.4.2 Loop Pipelining

Stage 2 of the optimisation process explores the possibility of loop pipelining [30]. The three functions called in the *dgefa* subroutine (*idamax*, *dscal* and *daxpy*) each contain a single loop over the same bounds. The C code for each of these functions is given in Figure 3.4. Each function also contains at least one pipelined floating point core. These cores are pipelined to fixed depths beyond the control of the user. Pipelining the loop iterations of each function into these cores will increase the parallelism and throughput of the system without increasing the resource usage. However, this may increase the

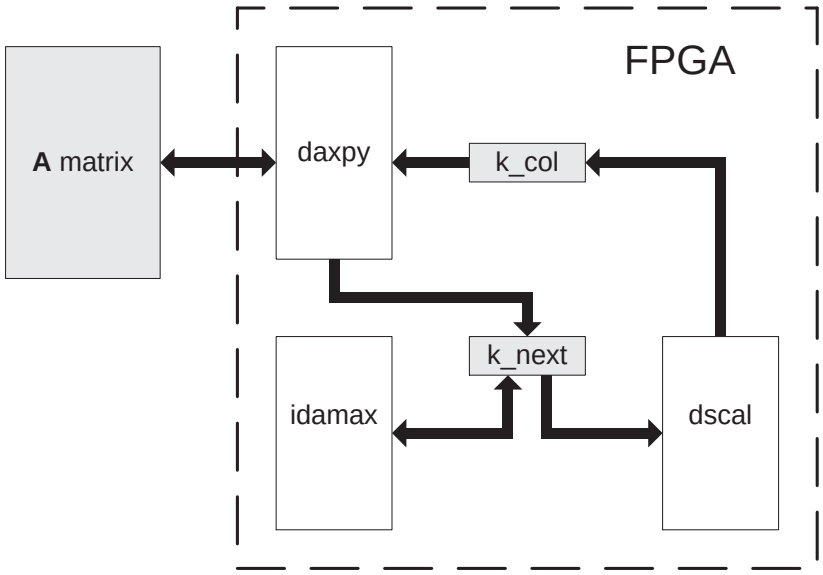


Figure 3.5: Overview of the dataflow for *dgefa* after data reuse. The bold arrows represent the dataflow between the functional blocks (white) and memory blocks (grey).

memory bandwidth required by each function as data will be accessed more frequently. The initiation interval of each pipeline must therefore take into account the bandwidth available for accessing each array on which the pipeline operates.

Each array in the algorithm has been provisionally mapped to a layer of the memory hierarchy. This places a loose, upper bound figure on the bandwidth available to access each array. During pipelining the arrays accessed by the target loops are, where necessary, given more exact array to memory allocations to maximise the pipeline performance. For example the loop in the *daxpy* function, the pseudo code for which is shown in Figure 3.4, iterates over a single instruction with two floating point operations. There is a read and a write to the **A** matrix and a read from the **k_col** buffer. An new iteration of the loop may begin on every clock cycle provided all of these memory operations can be issued once per cycle.

The **k_col** buffer is provisionally assigned to on-chip memory and so the requirement of a single read per cycle can always be met without making this assignment any more strict (by requiring dual port memory for example). The **A** matrix is assigned to the off-chip SRAM and each of the SRAM banks is assumed to be single ported. This means

that the requirement of one read and one write to and from **A** per cycle can only be met if the elements of **A** are partitioned across multiple SRAM banks in such a way that the read and write addresses are never in the same bank. The read and write will always access different rows of **A** and so partitioning the even rows and the odd rows to different banks will allow this requirement to be met, provided an odd row is read when an even row is written and vice versa. An odd row will be read when an even row is written (and vice versa) if there is an odd number of clock cycles between the read and write for the same iteration of the loop in the *daxpy* function. The combined pipeline depth for the floating point adder and multiplier on the datapath between the read and write operation in the *daxpy* function is 10 clock cycles, so an extra cycle delay (register) is added to meet the requirement for an odd number of cycles.

The loop in the *idamax* function requires only a single read from the **k_next** buffer per cycle and so the provisional assignment of on-chip memory is sufficient. The *dscal* function requires a read from **k_next** and a write to **k_col** and both these requirements are again met by the provisional assignments.

3.4.3 Loop Level Parallelism

In this optimisation stage the goal is to schedule loops/loop iterations with maximum parallelism within the memory bandwidth constraints. Resource constraints are ignored for now and will be taken into account later. Examination of the *dgefa* subroutine reveals that there is a loop carried dependency in the outer loop such that all of the input data used by iteration $k + 1$ is generated in iteration k. This means that, if multiple outer loop iterations are run in parallel with their start times offset correctly, then data could effectively flow from the outputs of one iteration to the inputs of the next without going through the external memory. Only the first outer loop iteration would need to read data from the external memory, and only the final outer loop iteration would need to write data[2]. This schedule, detailed in Figure 3.6(a) for the first 4 outer loop iterations, allows

[2]This assertion is not strictly true as each outer loop iteration must still write the result column from the instance of the *dscal* function within it to external memory as this data is not modified further. This issue is dealt with in the following section.

Iteration 0	Iteration 1	Iteration 2	Iteration 3
idamax[0]	wait	wait	wait
dscal[0]	wait	wait	wait
daxpy[0][1]	idamax[1]	wait	wait
daxpy[0][2]	dscal[1]	wait	wait
daxpy[0][3]	daxpy[1][2]	idamax[2]	wait
daxpy[0][4]	daxpy[1][3]	dscal[2]	wait
daxpy[0][5]	daxpy[1][4]	daxpy[2][3]	idamax[3]
daxpy[0][6]	daxpy[1][5]	daxpy[2][4]	dscal[3]
daxpy[0][7]	daxpy[1][6]	daxpy[2][5]	daxpy[3][4]
▮	▮	▮	▮
▮	▮	▮	▮
▮	▮	▮	▮
daxpy[0][999]	daxpy[1][998]	daxpy[2][997]	daxpy[3][996]
	daxpy[1][999]	daxpy[2][998]	daxpy[3][997]
		daxpy[2][999]	daxpy[3][998]
			daxpy[3][999]

(a)

Pipeline 0	Pipeline 1	Pipeline 2	Shared
wait	wait	wait	idamax[0]
wait	wait	wait	dscal[0]
daxpy[0][1]	wait	wait	idamax[1]
daxpy[0][2]	wait	wait	dscal[1]
daxpy[0][3]	daxpy[1][2]	wait	idamax[2]
daxpy[0][4]	daxpy[1][3]	wait	dscal[2]
daxpy[0][5]	daxpy[1][4]	daxpy[2][3]	idamax[3]
daxpy[0][6]	daxpy[1][5]	daxpy[2][4]	wait
▮	▮	▮	▮
▮	▮	▮	▮
▮	▮	▮	▮
daxpy[0][999]	daxpy[1][998]	daxpy[2][997]	dscal[3]
daxpy[3][4]	daxpy[1][999]	daxpy[2][998]	idamax[4]
daxpy[3][5]	wait	daxpy[2][999]	dscal[4]
daxpy[3][6]	daxpy[4][5]	wait	idamax[5]
daxpy[3][7]	daxpy[4][6]	wait	dscal[5]
daxpy[3][8]	daxpy[4][7]	daxpy[5][6]	idamax[6]
daxpy[3][9]	daxpy[4][8]	daxpy[5][7]	wait
▮	▮	▮	▮
▮	▮	▮	▮

(b)

Figure 3.6: Loop level scheduling for *dgefa* (a) Schedule for the first 4 outer loop iterations, ignoring resource constraints. (b) Resource constrained schedule for the first 6 outer loop iterations using 3 loop pipelines. idamax[k] and dscal[k] represent the execution of the *idamax* and *dscal* functions respectively in iteration k of the outer loop in *dgefa*. daxpy[k][j] represents the execution of the *daxpy* function in iteration j of the middle loop in *dgefa* and iteration k of the outer loop.

arbitrary parallelism with only the existing two ports to the external memory specified during loop pipelining. In Figure 3.6(a) `idamax[k]` and `dscal[k]` represent the execution of the *idamax* and *dscal* functions respectively in iteration k of the outer loop of *dgefa*. `daxpy[k][j]` represents the execution of the *daxpy* function in iteration j of the middle loop of *dgefa* and iteration k of the outer loop.

The pattern shown in Figure 3.6(a) can be extended to cover all outer loop iterations. Scheduling iterations of any other loop level in parallel would lead to a memory bandwidth requirement that scales with the parallelism. *idamax*, *dscal* and *daxpy* all contain loops iterating over the same bounds so they essentially take the same number of clock cycles to run, but the pipeline depths of the floating point macros in each function are not the same. As a result the execution times for the three functions would differ by around 2%, so extra delays (registers) are added to the *idamax* and *dscal* functional units to standardise the execution times. This allows instances of the three functions to be scheduled as if they were individual instructions. The other operations in the algorithm are moved into one of these three functions to simplify the coarse grain scheduling of the system (we first schedule instances of each function before considering the low level scheduling of operations within each function).

`idamax[k+1]` operates on the data generated in `daxpy[k][k+1]`. As the output data from `daxpy[k][k+1]` is generated it can be sent directly to `idamax[k+1]`, allowing the two functions to run in parallel. `daxpy[k+1][j]` runs in parallel with `daxpy[k][j+1]` but uses the data that was generated by `daxpy[k][j]` during the previous iteration of the middle loop. To deal with this an additional buffer, capable of storing up to one column of matrix data, must be inserted between consecutive iterations. These buffers act to store the data generated by each iteration until the next iteration in the chain is ready to use it. The buffers, referred to as **j_col** buffers, are provisionally assigned to simple dual port memory (one read port and one write port) on the FPGA. They must be simple dual port since they must be written and read concurrently by two loop iterations to keep the internal pipelines full.

The greedy schedule of Figure 3.6(a) would require at least 500 copies of the outer loop, which is beyond the capacity of current FPGAs. In reality there may be N copies of the outer loop, referred to as loop pipelines from here on, in the final system. The greedy schedule shown in Figure 3.6(a) can be modified to use only N loop pipelines by splitting the outer loop into groups of N consecutive iterations. The first N iterations are scheduled to begin as soon as their dependences allow. Once iteration 0 ends the next N iterations can be scheduled, and so on. Figure 3.6(b) shows how the first 6 outer loop iterations can be scheduled across 3 parallel outer loop pipelines. The pattern shown repeats for all subsequent iterations. Note that in the greedy schedule of Figure 3.6(a) the *idamax* and *dscal* functions are never required by more than one outer loop iteration at once. As a result they can be shared across all N hardware pipelines to reduce resource usage. In this implementation only the first pipeline in the chain (Pipeline 0 in Figure 3.6(b)) will read data from the external memory, and only the last pipeline (Pipeline 2 in Figure 3.6(b)) will write data.

3.4.4 Memory Access Details

While deriving the schedule for parallelising the outer loop iterations of the *dgefa* subroutine in the previous section it was assumed that only the first and last outer loop pipeline instances would accesses the external memory. This would lead to the block diagram for the FPGA dataflow shown in Figure 3.7. However, this simplified view of the system is incorrect. The assumption that only the first pipeline in the chain reads from the external memory and only the last pipeline writes was employed to simplify the description of the loop parallelisation approach, but it is not accurate. This is because the instance of the *dscal* function executed in each outer loop iteration of the *dgefa* subroutine generates (up to) a column of matrix data that will not be modified by any further computations on the FPGA. As a result this data must be written back to the off-chip memory for use by the host CPU system. Furthermore, the matrix column used by the *idamax* function in the first iteration of the outer loop in *dgefa* must be read from the off-chip memory since this data is not the output of any previous outer loop iteration and so is not be buffered

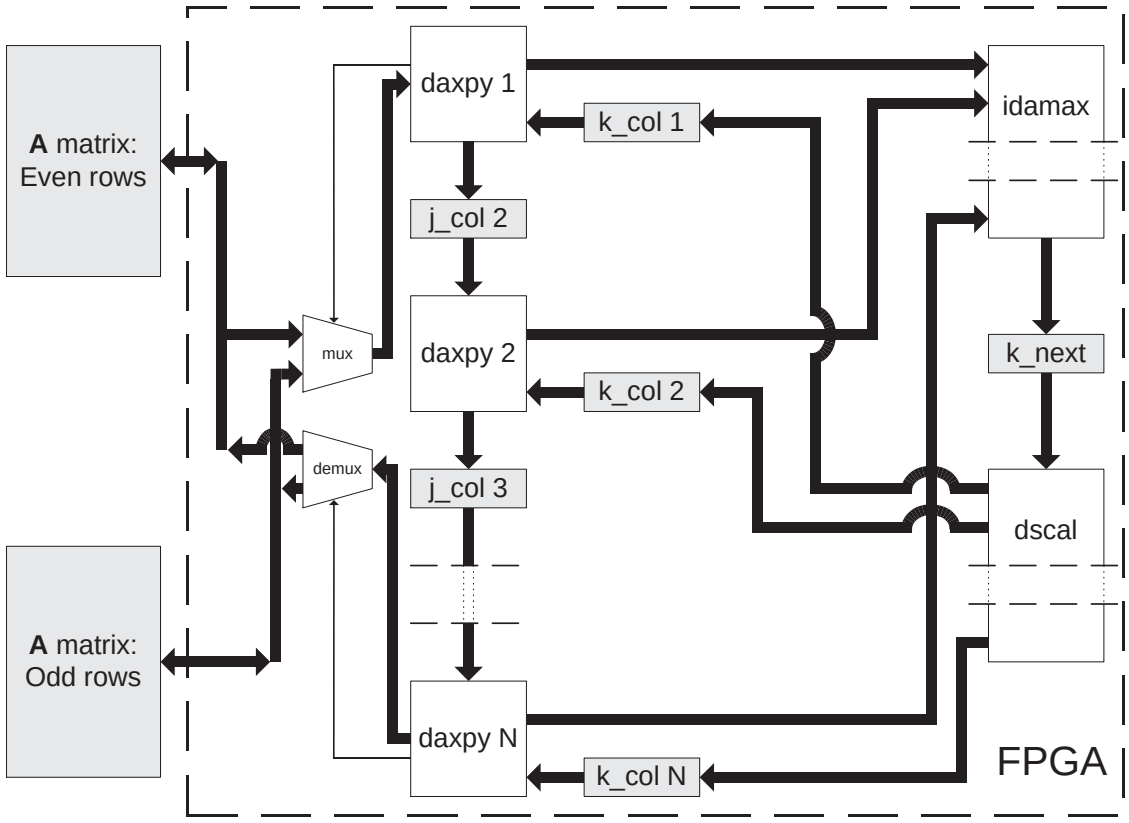


Figure 3.7: Simplified overview of the dataflow for *dgefa*, using N outer loop copies, after exploiting loop level parallelism. The bold arrows represent the dataflow between the functional blocks (white) and memory blocks (grey). The two thin arrows represent signals controlling which input is selected for reading by the multiplexor (mux) and which output is selected for writing by the de-multiplexor (demux).

on-chip (as it is with all subsequent instances of *idamax*).

These issues produce one additional read source (*idamax*) and one additional write source (*dscal*) for the off-chip memory. Fortunately both of these issues can be rectified without any modification to the existing schedule as there are sufficient 'gaps' in the usage of the memory ports by the *daxpy* operations of the outer loop pipeline instances. Figure 3.6(b) shows that the first pipeline in the system (Pipeline 0), which is the only other read source in the system for the off-chip memory, is idle during the execution of the *idamax* instance from the first outer loop iteration in *dgefa* (`idamax[0]` in Figure 3.6(b)). Hence `idamax[0]` can read the data it needs from the off-chip memory without causing any port conflicts.

Writing the result data from the executions of each *dscal* instance is more complicated as these are spread throughout the schedule. However, there are sufficient 'wait' slots in the schedule for the final loop pipeline in the chain (Pipeline 2 in Figure 3.6(b)), which is the only other write source in the system for the off-chip memory, to write all the necessary data. There is also a regular pattern to the timing of these 'wait' slots that can be exploited. It can be seen in Figure 3.6(b) that Pipeline 2 is idle during the executions of the first 3 *dscal* instances. Hence the data produced by the *dscal* functional block for these instances could be written to the off-chip memory as it is generated without creating port conflicts. However, later instances of the *dscal* function, notably `dscal[3]` and `dscal[4]` in Figure 3.6(b), execute in parallel with the *daxpy* operations of Pipeline 2. This means that, in the general case, the *dscal* output data cannot be written to the off-chip memory as it is generated without producing port conflicts. Fortunately this is not a problem since the output data from the *dscal* function is stored for the duration of a *dgefa* outer loop iteration in the **k_col** buffers for the loop pipelines. Hence the data is not lost and there is a window of an entire outer loop iteration in which to write the data to the off-chip memory before it is overwritten.

Between each consecutive outer loop iteration of the *dgefa* subroutine executed on Pipeline 2 in Figure 3.6(b) there are 2 'wait' slots where the pipeline is idle and data can be written to the off-chip memory without conflict[3]. Furthermore, the data produced by the first *daxpy* instance of each outer loop iteration (`daxpy[5][6]` in Figure 3.6(b) for example) is not written to the off-chip memory as it is sent to the *idamax* block and buffered on-chip. This means there are 3 'slots' during which columns of *dscal* data can be written to the off-chip memory during each outer loop iteration, which matches the 3 columns of *dscal* data that must be written during each outer loop iteration. As Pipeline 0 reads the *dscal* data stored in the **k_col** buffer during the third *daxpy* instance of the current outer loop iteration[4], Pipeline 2 will be in a 'wait' state and so this data can be sent to the off-chip memory. Likewise Pipeline 1 can write its **k_col** data during the

[3]These 'wait' slots exist because the number of middle loop iterations, and hence the number of *daxpy* instances, of the *dgefa* subroutine decreases by one with every outer loop iteration.

[4]daxpy[0][3] and daxpy[3][6] in Figure 3.6(b).

second *daxpy* instance of the current outer loop iteration[5]. Pipeline 2 can write the **k_col** data during the execution of the first *daxpy* instance of the current outer loop iteration.

This pattern can be generalised to a system with any number of loop pipelines, N. Let x be the number of each pipeline in the chain (for instance, for Pipeline 0, $x = 0$). Each pipeline will write the **k_col** data to the off-chip memory during *daxpy* instance $N - x$ of the current outer loop iteration. Note that the responsibility for writing out the data produced by the *dscal* functional block has shifted from the *dscal* block itself to the pipeline instance by which the *dscal* data is read. This produces the block diagram for the FPGA dataflow shown in Figure 3.8. Because there are now 2 read sources and N write sources for the off-chip memory, a memory controller block is included to oversee the reading and writing of the off-chip memory data. This block is written in Handel-C and is statically scheduled. For memory writes the other functional blocks merely place data on the input lines to the controller and, depending on the point in the schedule currently being executed, the memory controller selects which of its input lines to take data from and writes the data to the appropriate memory address. For memory reads the controller selects the appropriate address for the given point in the schedule and places the data on its output lines.

3.4.5 Low Level Optimisation

In this optimisation stage a number of steps are undertaken to increase the maximum clock frequency of the design and exploit any remaining instruction level parallelism. Complex calculations and control expressions are first broken up into several smaller operations. Next the code is stepped through and instructions that may be implemented in parallel, without breaching memory bandwidth or dependence constraints, are enclosed within 'par' tags. Each 'par' tag instructs the Handel-C compiler to schedule all the operations in the following set of braces ({}) in parallel.

At this point the arrays in the algorithm can be given final placements in the memory resources. For instance, the on-chip **k_col** and **j_col** buffers and the **k_next**

[5] daxpy[1][2] and daxpy[3][6] in Figure 3.6(b).

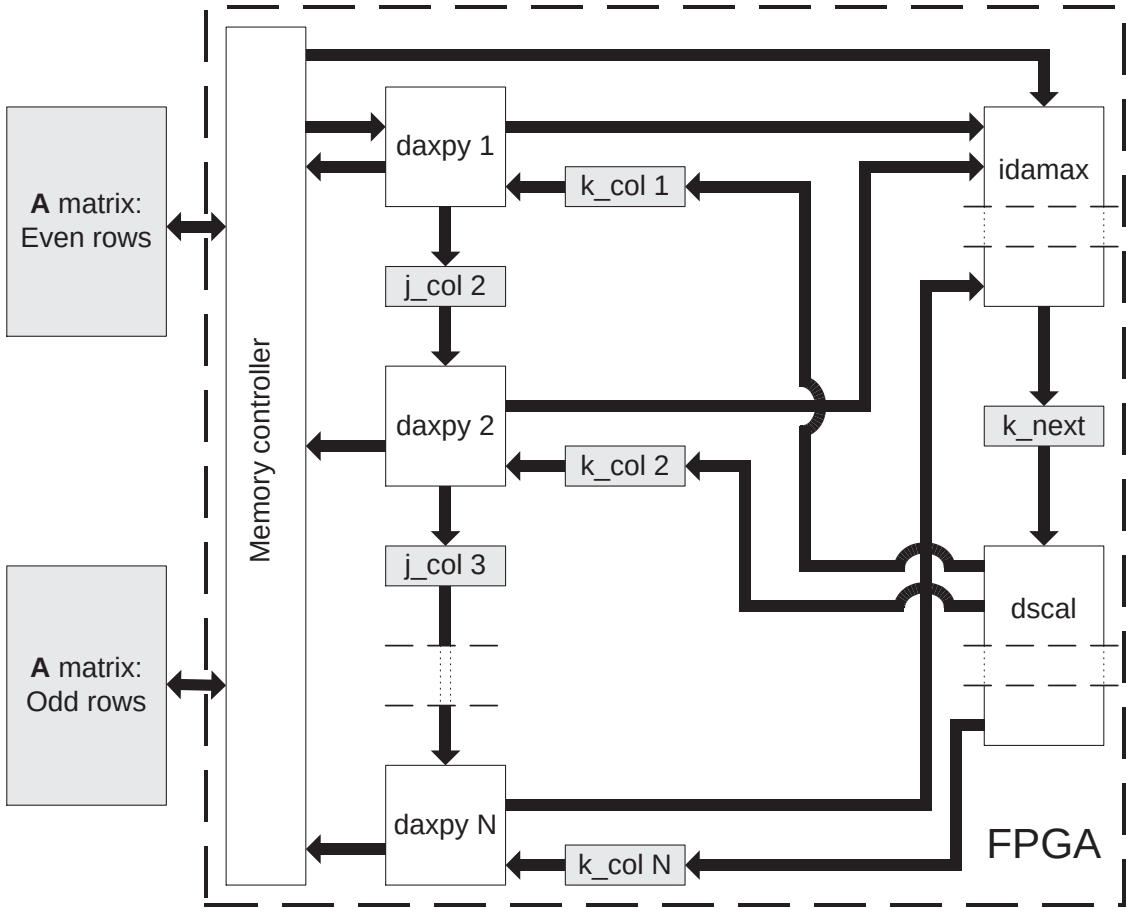


Figure 3.8: Overview of the dataflow for *dgefa* including the external memory controller.

buffer are declared as simple on-chip dual port memories (targeted to 'M4K' blocks for Stratix II devices). Additional registers (variables) are added to the read ports of all the physical memory blocks used, including the off-chip SRAM ports. These registers are written to only by the memory block to which they are connected. This process, referred to as 'memory pipelining' by Celoxica, prevents the Handel-C compiler from clocking the memory blocks on the negative edge of the global clock as it would otherwise do to preserve the one-cycle-per-assignment semantics (see pages 203-215 of [110]), halving the maximum clock speed for the system. The final step is to insert pipeline registers in potentially long routing paths, such as the global variables used to send data to and from the shared *idamax* and *dscal* functions.

3.5 Results

Versions of the *dgefa* coprocessor featuring increasing numbers of loop pipelines were targeted to an Altera Stratix II device (EP2S180). In each case the Handel-C source was compiled to EDIF using Agility (formerly Celoxica) DK Suite v4.0. The EDIF was synthesised using the Design Space Explorer utility of Altera Quartus II v9.1 [100]. Design Space Explorer attempts to place and route the design multiple times with different starting seeds. For this work 8 different seed values were used for each design, with place and route effort set to maximum and all physical synthesis optimisations enabled. The results reported are those for the place and route option found with the highest clock frequency.

The functionality of the *dgefa* coprocessor was verified using the Handel-C simulator in DK Suite. A 4 pipeline version of the *dgefa* coprocessor was downloaded to an Altera Stratix EP1S40 device on a NIOS development board [134] and verified for a system of order 32. It was not practical to implement a larger order system for verification since the results from a DK-Suite simulation of the Handel-C code are required to verify the results achieved in hardware, and the simulation time for even a matrix of order 32 is over an hour.

Table 3.1 details the resource usage and performance of each implementation of the *dgefa* subroutine. The speedup figures are in comparison to the implementation of the benchmark using functions from the optimised ATLAS BLAS library running on a 3.4GHz Pentium 4. In each case the speedup is calculated as the ratio of the Pentium 4 execution time to the FPGA execution time. Two figures are given in each case for the speedup. The figures outside the brackets give speedup for just the execution of the *dgefa* subroutine. *dgefa* executes in 0.29 seconds on the Pentium 4. The figures outside the brackets give estimated times and speedup factors for the entire LINPACK 1000 benchmark. These figures assume that the remaining code is executed on a 3.4GHz Pentium 4 and that data transfers between the processor and the FPGA occur at 75% of the maximum data rate available through a 66MHz PCI bus. With 16 loop pipelines the FPGA implementation executes the *dgefa* subroutine 1.66 times faster than the Pentium 4, but the overhead of transferring the data to and from the FPGA board means that the overall benchmark

Table 3.1: Summary of the implementation results achieved when targeting an Altera Stratix II device. ALUTs are the basic logic elements of the Stratix II device family, while M4K blocks are embedded RAM blocks. The speedup numbers inside brackets indicate the performance of the FPGA relative to an Intel Pentium 4 processor for the complete LINPACK 1000 benchmark. The speedup numbers outside brackets give performance relative to an Intel Pentium 4 processor for the just the *dgefa* subroutine.

No. pipelines	ALUTs	Registers	DSP blocks	M4K blocks	F_{max} (MHz)	Clock cycles	Speedup
2	8040 (4.7%)	7597 (4.4%)	96 (12.5%)	80 (10.4%)	148	171172048	0.23 (0.25)
3	10827 (6.3%)	10187 (5.9%)	128 (16.7%)	112 (15.8%)	146	114285343	0.30 (0.37)
4	13417 (7.8%)	12713 (7.4%)	160 (20.8%)	144 (18.8%)	137	85842160	0.36 (0.46)
8	29354 (17.1%)	23412 (13.6%)	288 (37.5%)	272 (35.4%)	133	43177980	0.62 (0.88)
16	46539 (27.1%)	43424 (25.3%)	544 (70.1%)	528 (68.8%)	126	21847172	0.90 (1.66)
23	66750 (38.9%)	62426 (36.3%)	768 (100%)	752 (98.0%)	122	15356514	1.05 (2.29)

still executes more quickly on the Pentium 4. The maximum number of pipelines that can be implemented on the largest Stratix II device (an EP2S180) is 23. This design consumes all 768 9x9 multipliers in the DSP blocks on the device. With 23 pipelines the FPGA design can execute the *dgefa* subroutine 2.29 times faster than the Pentium 4. With the communication overhead taken into account, this gives virtually identical performance for the FPGA and Pentium 4 implementations of the complete LINPACK 1000 benchmark. During the execution of the *dgefa* subroutine the FPGA implementation achieves a sustained double precision floating point performance of 5GFLOPS.

Due to the heuristic nature of the place and route algorithms used in all commercial FPGA design tools it cannot be guaranteed that the clock frequencies reported for the *dgefa* coprocessor designs are the true maxima that may be achieved. Some work has been done in this area to estimate the level of error in the results produced by tools such as Quartus II by placing and routing synthetic designs with known minimum critical paths for specific FPGA devices [135]. The results in [135] suggest that Quartus II can find solutions within 5% of the best achievable implementation. For the designs implemented

in this work the worst case variance of the clock frequency with different starting seeds was 9%. Given that Quartus II presented a result that is 9% worse than a known alternative solution as optimal for one seed it could be argued that the best solution found could still be 9% worse than the solution for some other starting seed. Hence we estimate that there is potentially 5% to 10% error in the clock frequencies presented here. A 10% improvement in clock frequency would allow the FPGA to implement the *dgefa* subroutine 2.53 times faster than the Pentium 4, but this result was not found by the tool and cannot be presented as such.

The performance of the FPGA design produced relative to the Pentium 4 is promising. The results in [136] suggest that sustained FPGA and CPU double precision floating point performance should be comparable for devices of the era of Stratix II and the Intel Pentium 4. In [136] a sustained double precision performance of 10GFLOPS is suggested as reasonable for Stratix II devices, which is double what was achieved in this work. However, it should be noted that the maximum clock frequency achieved for the designs produced here varied between 146MHz for 2 pipelines and 120MHz for 23 pipelines. In every case the critical path for the design lay within one of the floating point cores from the Celoxica library, and not the components of the design developed during the conversion from software. The maximum operating frequency for a Stratix II device is 500MHz, but such figures would be difficult to achieve for large floating point designs. For double precision designs, clock frequencies of the order of 250MHz are achievable with deeply pipelined libraries [136]. If such figures had been achieved then the floating point performance achieved would have been around the 10GFLOPS predicted. This suggests that the approaches used in this work have achieved reasonable levels of parallelism in the FPGA design, and that the process has been reasonably successful. Obviously, with faster floating point units the critical path in the design may have moved into the logic developed here, but extra registers could have been inserted on any long paths to improve performance.

While there are no other FPGA implementation of the complete LINPACK 1000 benchmark to compare against, there are other implementations of LU decomposition [137, 138], which is the core of what has been implemented here. Of these the best performance

was achieved in [138], with a figure of just under 4GFLOPS reported. The implementation in [138] is simpler than the one presented here as it assumes no pivoting of matrix elements (partial pivoting is included in the *dgefa* subroutine), but our implementation is still able to outperform it by 25%. The implementation reported in [138] uses a Xilinx Virtex II Pro device (XC2VP100) [139], which is a generation behind the Stratix II device used here, but the clock frequency reported in [138] is 120MHz and the number of multipliers available on the device (888), which formed the critical resource, is comparable to the number on the largest Stratix II device (768). These results suggest that the implementation of the *dgefa* subroutine developed here is of a reasonable standard and that the optimisations employed have been successful.

The main goal of this piece of work has been to investigate the usefulness of existing hardware compilation techniques and to identify potential opportunities for new optimisation methods. Firstly this work has demonstrated first hand the gains that can be achieved when pipelining loops. In this example the combined pipeline depths of the floating point adder and multiplier within the *daxpy* function amounted to 10 clock cycles, giving 10 levels of parallelism if loop pipelining is exploited for virtually no additional hardware costs (other than possibly a slight increase in the control logic required). However, the results showed that pipelining alone did not yield sufficient performance to match the Pentium 4. Multiple pipelines had to be instantiated to achieve the necessary parallelism. However, this was only made possible through concurrent optimisation of the memory subsystem. The integration of data reuse, array partitioning and array to memory assignment with scheduling decisions allowed large amount of parallelism to be exploited. Of particular importance in allowing parallelism to scale independently of the required bandwidth was the exploitation of flow dependences across multiple loop iterations to implement data reuse buffers between pipelines. In the LINPACK case this approach allowed an implementation of an algorithm that could be limited by memory bandwidth to become limited by the physical resources available on the largest Stratix II device. The integration of data reuse, array partitioning and array to memory assignment with both low level scheduling decisions (loop pipelining) and loop level parallelism (loop unrolling) has not yet been tackled in existing literature. Furthermore, existing hardware compilers have typically avoided

loop levels with flow dependences spanning multiple iterations [27], rather than exploiting them. These are interesting areas for extending and adapting the existing optimisation approaches, and these will be taken up in Chapters 4 to 6.

3.6 Summary

A Handel-C implementation of a coprocessor for the LINPACK 1000 benchmark has been produced through a series of optimisations that identify opportunities to exploit data reuse, loop pipelining, coarse grained parallelism and parallel memory assignment. When targeted to Altera Stratix II FPGAs this coprocessor can execute the matrix LU decomposition subroutine up to 2.3 times faster than a highly optimised 3.4GHz Pentium 4 solution. The performance is also comparable to an optimised implementation of a slightly simpler LU decomposition routine, suggesting that the steps taken to produce the hardware solution in this work have been successful.

The combined optimisation of the memory subsystem and schedule for a target loop has been identified as an area where the existing work can be extended upon, especially with regard to the exploitation of data reuse using loop carried flow dependences in the target algorithm. By closely linking the various memory optimisation decisions (data reuse, array partitioning and array to memory placement) with both low level scheduling (loop pipelining) and the exploitation loop level parallelism (loop unrolling) it may be possible to improve and extend upon existing work, which has linked some of these factors, but not all of them.

Chapter 4

Pipelining Above the Innermost Loop

4.1 Introduction

In the previous chapter the co-optimisation of the datapath (schedule) and the supporting memory subsystem was highlighted as a key issue in the generation of FPGA co-processors that can exploit maximum levels of parallelism for a given loop on a given target platform. In this chapter the goal is to take the first step towards producing such a co-optimisation approach, by implementing a loop pipelining scheme into which further memory and loop unrolling optimisations may later be integrated. Loop pipelining techniques [30] are critical in achieving efficient parallelism and are included in most hardware compilers. Traditionally loop pipelining is applied at the innermost loop level in a nested loop [8,9,27,30]. This can lead to inefficient solutions in cases where there are dependences that cross multiple iterations at the innermost loop, or if the innermost loop has few iterations. A number of methods have been developed in the software domain that allow nested loops to be pipelined above the innermost level [13, 37, 39] and these can allow shorter schedules to be achieved. The Single Dimension Software Pipelining (SSP) approach [13] in particular has been shown to allow shorter schedules with higher levels of parallelism than can be achieved with inner loop methods, even when loop transformations such as interchange

and unroll are also considered. When applied to nested loops the existing hardware compilers target only the innermost loop level for pipelining. Outer loop pipelining has not yet been considered for hardware because it is assumed that the increase in the control complexity will reduce the maximum clock rate that can be achieved to such an extent that any scheduling gains will be outweighed [27].

In this chapter the existing SSP approach is extended to better suit the generation of schedules for hardware, specifically FPGAs. A search scheme is introduced to find the shortest schedule available within the pipelining framework to maximise the gains in pipelining above the innermost loop. A generic pipeline controller is also developed which is capable of implementing schedules pipelined above the innermost loop without significantly reducing the maximum clock speed of the final system below that which can be achieved for an inner loop only solution. The scheduling approach is applied to nine loop kernels to generate hardware coprocessors which are targeted to an Altera Stratix II FPGA. The results show that the fastest solution for each loop occurs when pipelining is applied above the innermost loop. When compared to inner loop pipelining a maximum speedup of 7 times is achieved, with an average speedup across the nine loops of 2.9 times.

The remainder of the chapter is split into seven further sections. In Section 2 a brief description of Modulo Scheduling and the existing SSP methodology is provided. Sections 3 and 5 describe the extension and adaptation of this existing methodology to improve results when targeting FPGAs, and Section 4 describes the FPGA resource constraints. In Section 6 details of the scheme to search the solution space are presented, along with details of the modulo scheduling formulation. Section 7 provides an overview of the hardware structures required to control datapaths for loops pipelined above the innermost level, and Section 8 details the results achieved for nine test loops. Section 9 summarises the conclusions of this work.

4.2 Background

Perhaps the most widely used loop pipelining methods are based around modulo scheduling [30]. In modulo scheduling the operations from a single iteration of the loop body are scheduled into S stages, with each stage requiring T clock cycles to execute. Each operation in the loop body is assigned to start on a single cycle in a single stage. The S stages run sequentially to execute a single iteration of the innermost loop, but may all run in parallel for different loop iterations without breaching the resource constraints of the target platform. A new iteration of the innermost loop is initiated every T clock cycles with the result that the executions of S loop iterations are overlapped.

Standard modulo scheduling based methods are limited to pipelining (overlapping) the iterations of the innermost loop in a loop nest [30]. Single-dimension Software Pipelining (SSP) [13] extends innermost loop pipelining methods, such as modulo scheduling, allowing them to be applied at any level in a rectangular loop nest. Under this methodology a single loop level is selected for pipelining based upon metrics such as the expected initiation interval for each level and/or the potential for data reuse. The data dependence graph for the loop nest is then simplified according to the method presented in [13]. By assuming that the iterations from loop levels above and below the pipelined level execute sequentially, all dependence distance vectors [140] are reduced to equivalent scalar values. This allows standard modulo scheduling techniques to be applied to the nested loop, regardless of which level is being pipelined. The final schedule is then constructed from the modulo schedule. A new iteration of the pipelined loop level is initiated every T clock cycles, but an extra delay must be added after each group of S consecutive iterations. The delay added between each group is the same and its value is defined in [13]. The extra delays are necessary to ensure that no more than S iterations are overlapped into a pipeline with S stages as this would cause resource conflicts. The existing SSP methodology has been developed for software targeted to processor based platforms that offer instruction level parallelism.

4.3 Extending the Solution Space

Standard modulo scheduling restricts the initiation interval (II) and the number of clock cycles per stage (T) to be the same. This restriction is maintained in the existing Single Dimension Software Pipelining (SSP) work [13]. In this section it will be shown that, when pipelining above the innermost loop level, this restriction may increase the length of the final schedule.

Let $ResMII$ and $RecMII$ be the lower bounds on II set by the system's resource and dependence constraints respectively, as described in Section 2.3.1. The determination of $ResMII$ is discussed in Section 4.4 while the calculation of $RecMII$ is described in [34]. The stage length, T, must be greater than or equal to $ResMII$, while the minimum II is bounded by both $ResMII$ and $RecMII$. Thus, in cases where the resource constraints are less restrictive than the dependence constraints ($RecMII > ResMII$), the minimum T will be less than the minimum II. Forcing T and II to take the same value may therefore increase the minimum stage length that can be achieved and reduce the available solution space.

Equation (4.1) defines the number of clock cycles required to execute a rectangular, perfectly nested loop with L levels of nesting[1] which has been pipelined at some arbitrary loop level, p. N_i represents the number of loop iterations at level i in the nest[2].

$$cycles = \Big(\prod_{k=0}^{p-1} N_k\Big) \cdot \Bigg((\lceil N_p/S \rceil - 1) \cdot \max\Big(S \cdot T \cdot \big(\prod_{i=p+1}^{L+1} N_i\big), S \cdot II\Big) + S \cdot T \cdot \big(\prod_{i=p+1}^{L+1} N_i\big) + \big((N_p - 1) \bmod S\big) \cdot II \Bigg) \quad (4.1)$$

The derivation of Equation (4.1) is provided in Appendix A. When a loop nest with large loop counts is pipelined above the innermost loop the length of the schedule is dominated by the value of T ($cycles \approx T \cdot \prod_{i=1}^{L} N_i$). Hence finding the shortest schedule may require T to be minimised at the expense of a larger value of II.

[1] Level L is the innermost loop and level one the outermost.
[2] N_{L+1} and N_0 are defined to be one for uniformity.

The original SSP work targets platforms with limited functional units (relative to platforms such as FPGAs where logic resources are abundant) such as VLIW processors. One would expect that the limited number of functional units on these platforms might produce relatively tight resource constraints such that $ResMII$ will typically be greater than $RecMII$. Hence the authors of the original work saw no need to separate the values of T and II as they will most likely both be constrained to the same value by $ResMII$. However, the abundant logic resources on modern FPGAs (discussed in detail in Section 4.4) may allow smaller values of $ResMII$, such that $RecMII$ dominates. Thus it was deemed worthwhile to remove the constraint for II and T to have the same value. This extension could be applied to other architectures, such as VLIW processors, but it may simply increase the scheduling effort for no significant scheduling gains.

Allowing the values of II and T to take different values should enable greater acceleration over pipelining at the innermost loop level in cases when $RecMII$ is greater than $ResMII$. However, no assumptions are made about the relative values of $RecMII$ and $ResMII$ and the methodology presented in the remainder of this chapter is applicable in all cases. The separation of these values merely extends the available solution space over that offered by the original SSP work. If $ResMII$ is greater than $RecMII$ then the solution our approach finds may offer less significant (if any) gains over the original SSP approach.

4.4 FPGA Resource Constraints

The goal of this work is to identify a compute intensive loop nest in a target application and compile this loop to a hardware coprocessor on an FPGA. Modern FPGAs have high resource densities with the largest devices offering in excess of 10^5 look up tables (LUTs) and hundreds of embedded multipliers [141]. It may therefore be reasonable to assume that each instruction in the target loop can be implemented on a dedicated resource, ultimately producing no resource constraints (*i.e.* $ResMII = 1$). However, while logic resources are abundant, the data operated on by the FPGA datapath is typically stored in off chip memories, and the number of ports through which this data may be accessed will

be limited. Often it is not the physical resources on the device that limit performance, but the off chip memory bandwidth available to access the arrays used by the application [36]. The limited number of off chip memory ports acts as a constrained resource that must be scheduled around during pipelining, much as the limited number of adders or multipliers must be scheduled around when targeting architectures with limited arithmetic resources. In this work it is assumed that it is the number of available memory ports that will limit the parallelism that may be exploited, rather than the available computational resources. It is also assumed that all functional units (such as multipliers) with latencies greater than a single clock cycle are internally pipelined, which is reasonable given the architectures of modern FPGAs [141].

With these assumptions and an array to physical memory map (which is must be supplied by the designer in this work)[3] the value of $ResMII$ for a given loop and target platform may be calculated based on the ratios of accesses to ports for each memory. The minimum number of clock cycles (cyc_m) required to execute the memory accesses to memory m in a single iteration of the innermost loop (ignoring dependence constraints) can be computed using equation (4.2). R_m represents the number of reads and W_m the number of writes in a single iteration of the innermost loop for all of the arrays assigned to memory m. $ports_m$ is the number of ports[4] to m, and Iss_{rm} and Iss_{wm} are the issue intervals (the minimum number of cycles between successive operations) of the read and write operations respectively. The final value of $ResMII$ for the given target platform and target loop is then defined by equation (4.3).

$$cyc_m = \left\lceil \frac{Iss_{rm} \cdot R_m + Iss_{wm} \cdot W_m}{ports_m} \right\rceil \tag{4.2}$$

$$ResMII = \max_m(cyc_m) \tag{4.3}$$

[3]The only requirement of the array to memory map is that each memory is sufficiently large to accommodate all the arrays mapped to it.

[4]In this description only read/write ports are considered, but the methodology is easily extended to deal with dedicated read and write ports.

4.5 Imperfect Nesting

In practice few loops are perfectly nested and so the Single-dimension Software Pipelining methodology is extended to deal with imperfectly nested instructions [40]. This methodology requires the use of multiple stage lengths to achieve maximum efficiency. To avoid such complications a simpler method for scheduling imperfectly nested loops has been developed. Extra imperfect stages are added to the pipeline which execute only at the beginning and/or end of the loop level where they are required. To prevent resource conflicts in the schedule, the imperfect stages are added in multiples of S, where S is the number of perfect stages. The reasoning behind this is demonstrated using the simple example in Figure 4.1(a).

In this example it is assumed that the loop is first pipelined ignoring the imperfectly nested instructions and then modified to include them. The outermost loop is pipelined and the perfect operations are scheduled into five stages, with an initiation interval of two stages[5]. Once the pipeline has been filled the perfect schedule follows the pattern shown in Figure 4.1(b). The stepped bold line in Figure 4.1(b) marks on each outer loop iteration the end of one iteration of the middle loop level and the start of the next. To implement the full imperfect loop the execution of imperfect operations must be inserted along this line.

Assuming that all of the imperfect operations can be executed in a single pipeline stage the obvious schedule would be that shown in Figure 4.1(c). However this schedule causes multiple iterations to require the same stage at the same time, which is not allowed. A possible solution to this problem is to include extra (possibly empty) imperfect stages so that the number of imperfect stages matches the length of the initiation interval (two in this case). This does remove the stage conflicts, as shown in Figure 4.1(d), but it also creates a new problem. Both the imperfect stages (X0 and X1) must run in parallel with each of the perfect stages at some point in the schedule. This may make it impossible to assign a constrained resource to any cycle in the imperfect stages without conflicting with a perfect stage. This problem will persist for any number of imperfect stages fewer than S

[5]These values are chosen arbitrarily for the example and are not critical.

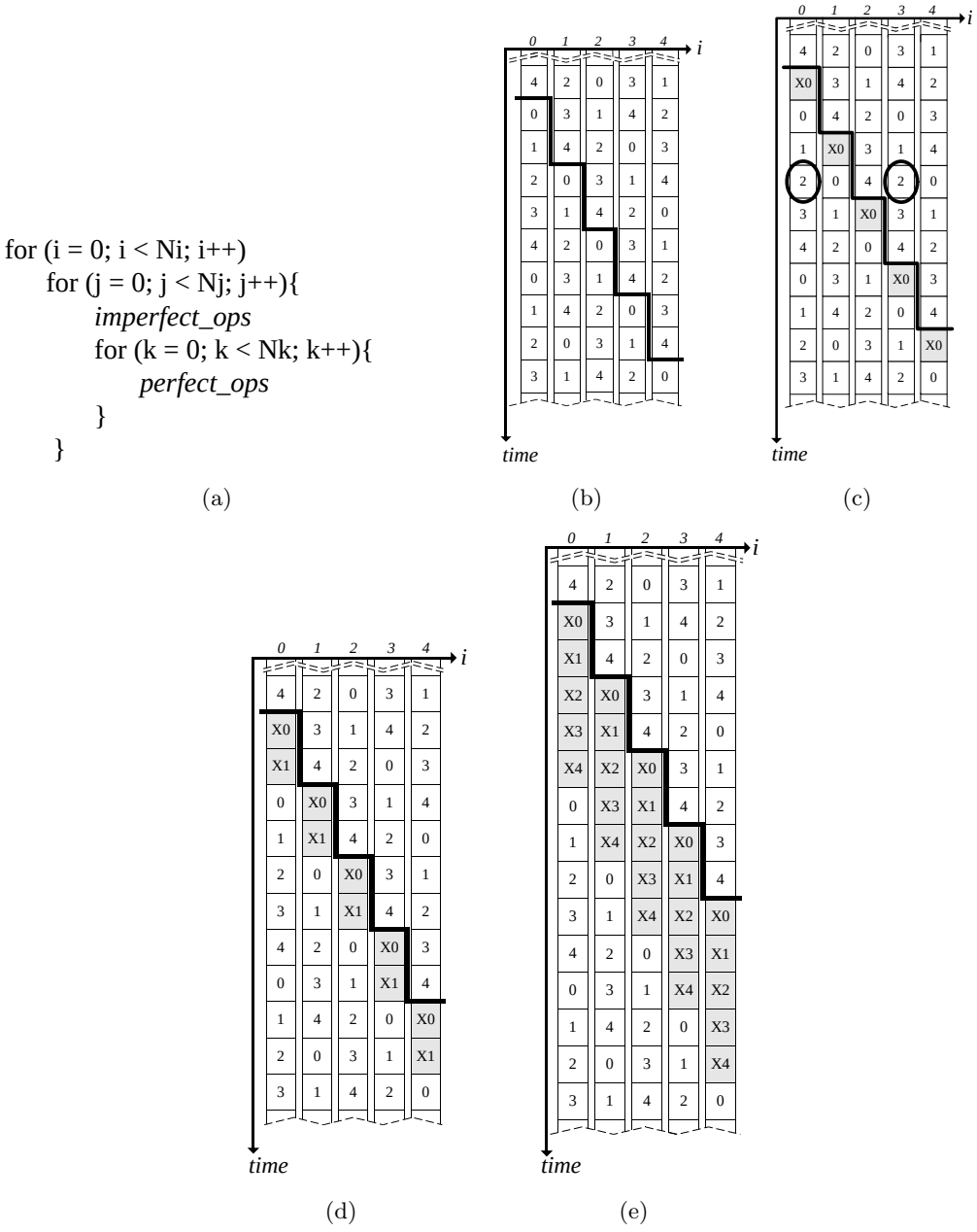


Figure 4.1: Scheduling imperfectly nested operations. Each numbered box represents an execution of the software pipelining stage denoted by the enclosed number. Each vertical column of stages represents part of the execution of one of the first five outer loop iterations. (a) A sample loop nest (b) A section of the perfectly nested schedule. (c) Extending the perfectly nested schedule to include one imperfect stage. The shaded grey boxes represent imperfectly nested stages. The black circles mark out an example of a resource conflict in the schedule (d) Extending the perfectly nested schedule to include two imperfect stages (e) Extending the perfectly nested schedule to include five imperfect stages.

(five in this case). Figure 4.1(e) shows that there are no stage conflicts when five imperfect stages are used. Furthermore, each imperfect stage always 'replaces' one particular perfect stage in the schedule. For example, stage X0 always executes in time slots that would be allocated to perfect stage 0 in the original perfect schedule. Hence, provided the resource usage pattern of the imperfect stage does not exceed that of the perfect stage it replaces on any cycle, there will never be resource conflicts in the imperfect schedule.

Let I_i be the set of imperfectly nested operations that must execute at the start/end of an iteration at level i in the loop nest. The start (or end) of an iteration at level i always entails the start (or end) of an iteration at level $i+1$. The set of imperfect operations executed for level $i+1$ is therefore a subset of the operations executed for level i. As one moves outwards from the innermost loop the number of imperfectly nested instructions may increase. Hence, to improve the efficiency of our approach, an increasing number of sets of stages may be included for each level in the loop above the innermost loop. Let us define Z_i to be the number of sets of stages included in the schedule to accommodate all imperfectly nested operations up to and including loop level i. Only the loop levels up to and including the pipelined level, p, are implemented on the FPGA. Hence the total number of sets of imperfect stages included in the schedule for the FPGA hardware is Z_p.

4.6 Scheduling

In this work the goal is to find the optimum (shortest) schedule available within the restrictions of our framework and the memory port constraints of the target platform. The main restriction of this work is that the target loop must be regularly nested, *i.e.* at each level in the loop there may be multiple instructions and one nested loop, but not multiple loops. For example, the C code for the *dgefa* subroutine in Figure 3.3 represents an irregularly nested loop since the *idamax* and *dscal* functions are called within a loop and both contain a loop. A further restriction is that the approach proposed here will only find optimal solutions for loops with fixed loop bounds since the number of iterations for each loop level must be known in advance. This approach can be applied to loops with non-fixed loop bounds, with the average number of loop iterations at each level used in

place of fixed iterations counts. However in this case it can not be guaranteed that the pipelining search will return the fastest solution.

The relatively large number of variables present, even for small loops, makes optimal scheduling within resource constraints a difficult problem. For this reason our scheduling approach is split into two parts. The first part is a search of the possible values for the stage length, T, the number of perfect stages, S, and the number of sets of imperfect stages, Z_p. The second part is an Integer Linear Programming (ILP) [80] based modulo scheduling formulation that assigns start times to the loop operations for fixed values of T, S and Z_p so that either the initiation interval, II, or the total number of stages executed, S_{tot}, is minimised.

The two inputs to the scheduling process are the data dependence graph for the nested loop and an array to physical memory map. The data dependence graph, $G(V, E)$, comprises a set of nodes, V, and a set of edges E. Each node, v_n, represents an operation in the loop nest. Each edge, (v_{n1}, v_{n2}), represents a dependence between nodes v_{n1} and v_{n2}, with v_{n2} dependent on v_{n1}. Edge (v_{n1}, v_{n2}) is tagged with a dependence distance vector, $\mathbf{d_{n1n2}}$, denoting the number of loop iterations separating the dependent instances of v_{n1} and v_{n2} at each loop level. The data dependence graph is simplified for each loop level. Each $\mathbf{d_{n1n2}}$ is reduced to single scalar value, d_{n1n2}, which denotes the number of loop iterations separating the dependent operations at the pipelined loop level. Further details concerning the form of the data dependence graph and the dependence simplification can be found in [13]. Further to the dependence simplification, all operations (nodes) nested above the pipelined level are removed from the graph as only operations from the pipelined loop level and below are implemented in the hardware coprocessor on the FPGA.

4.6.1 Searching the Solution Space

When pipelining an imperfectly nested loop at an arbitrary level, p, the length of the schedule is defined by equations (4.4) and (4.5). S_{tot} represents the total number of stages executed during a single iteration of the loop at the pipelined level and each Z_i represents

the number of sets of imperfect stages included for level i in the loop[6]. Equation (4.4) is simply a weighted sum of the number of sets of stages nested at each level in the loop and equation (4.5) is simply the imperfect generalisation of equation (4.1).

$$S_{tot} = S \cdot \left(\sum_{i=p+2}^{L} \left(Z_i \cdot (N_i - 1) \cdot \prod_{j=p+1}^{i-1} N_j \right) + (N_{p+1} - 1) \cdot Z_{p+1} + Z_p + \prod_{i=p+1}^{L+1} N_i \right) \quad (4.4)$$

$$cycles = \left(\prod_{k=0}^{p-1} N_k \right) \cdot \Bigg(\left(\lceil N_p / S \rceil - 1 \right) \cdot \max \left(T \cdot S_{tot}, S \cdot II \right) + (T \cdot S_{tot}) + \left((N_p - 1) \bmod S \right) \cdot II + (S \cdot Z_p \cdot T) \Bigg) \quad (4.5)$$

From these expressions it is not immediately clear how the scheduling parameters (T, S, II and the Z_i values) should be traded to find the schedule with the shortest execution time. To this end the search scheme presented in Algorithm 1 has been developed. Algorithm 1 must be executed for each level in the target loop.

The search begins by calculating the value of $ResMII$ based on the port constraints for the target platform and loop, as described in Section 4.4. The scheduling options are explored for increasing values of T until a bounding condition is satisfied. For any given value of T the minimum schedule length according to equation (4.5) is $\left(T \cdot \prod_{i=1}^{L} N_i\right)$ cycles[7]. The search terminates when a value of T is reached such that this lower bound schedule length is greater than the length of the best schedule already found. We choose to first fix the value of T and search the scheduling options for each candidate value for two reasons. Firstly, we expect the schedule length to increase more rapidly with the value of T than with any other parameter since the number of cycles is roughly equal to $\left(T \cdot \prod_{i=1}^{L} N_i\right)$ when the number of loop iterations at each level is large. Secondly, the Integer Linear Programming (ILP) formulation used for bounding values of S, Z_p and II, and for scheduling requires a fixed value of T (see Section 4.6.2 for further details).

For each value of T the minimum value of S is found. The scheduling options

[6] Z_i is defined to be zero for all $i \geq L$.
[7] N_i is the number of loop iterations at loop level i.

Algorithm 1 : Searching the pipelining solution space for each loop level. Lat_{imp} represents the sum of the latencies of all imperfect operations nested up to the pipelined level

```
 1: best = ∞;
 2: T = find_ResMII();
 3: while bound1(T) < best do
 4:    S = find_S_min(T);
 5:    II_est = estimate_II_min(T);
 6:    S_tot_est = estimate_S_tot_min(T, S);
 7:    while bound2(T, S, II_est, S_tot_est) < best do
 8:       Zp = find_Zp_min(T, S);
 9:       while Zp ≤ Lat_imp do
10:          II = find_II_min(T, S, Zp);
11:          S_tot_min = find_S_tot_min(T, S, Zp);
12:          done = true;
13:          while bound3(T, S, S_tot_min, II) < best do
14:             cycles = schedule(T, S, Zp, II);
15:             best = min(cycles, best);
16:             II++;
17:          end while
18:          Zp++;
19:       end while
20:       S++;
21:       S_tot_est = estimate_S_tot(T, S);
22:    end while
23:    T++;
24: end while
```

are then explored for increasing values of S until a bounding condition is breached. For each T the minimum II and S_{tot} values are estimated as these are required to bound the search. For each candidate S the lower bound schedule length is calculated assuming that the estimated minimum II and S_{tot} values may be achieved. It is also assumed that S is a factor of N_p as this removes the ceiling and modulus functions from equation (4.5), making the schedule length a monotonic increasing function of S. When an S value is reached such that this lower bound is greater than the length of the current best schedule, the search is able to progress onto the next value of T. The value of S is searched second because the ILP formulation for bounding values of Z_p and II, and for scheduling requires a fixed value of S.

For each value of S the minimum value of Z_p is found and the scheduling options are explored for increasing values of Z_p. Let Lat_{imp} be the sum of the latencies for all imperfect operations up to and including the pipelined level. Including Lat_{imp} sets of imperfect stages in the schedule allows every ordering for the imperfect operations within

the dependence and resource constraints to be reached. The optimum II and S_{tot} values can therefore always be achieved within Lat_{imp} sets of imperfect stages. Increasing the value of Z_p above Lat_{imp} will always increase the schedule length and so the search may progress onto the next S value once a Z_p value of Lat_{imp} is reached. A similar bounding approach is also employed for the range of S values. The value of S is never increased above Lat_{tot}, where Lat_{tot} is the sum of the latencies for all of the operations in the loop up to and including the pipelined level.

For each Z_p the minimum values of II and S_{tot} are found (not estimated). Modulo scheduling of the loop is then performed for the current values of S, T and Z_p with increasing values of II. The goal during modulo scheduling is the minimisation of S_{tot} as this yields the minimum schedule length for the given values of T, S, Z_p and II. The lower bound schedule length for each value of II is calculated using equation (4.5) by assuming that the minimum S_{tot} may be achieved. Once an II value is reached such that the lower bound exceeds the length of the best schedule already found, the search progresses to the next Z_p value. As with T and S, the decision to order the search so that the values of Z_p are explored third is driven by the requirements of the ILP formulation used for scheduling.

The *schedule*, *find_S_tot_min* and *find_II_min* functions all make use of the ILP modulo scheduling formulation described described in Section 4.6.2. The functions set different variable bounds and minimise different cost functions which are discussed at the end of Section 4.6.2. The *find_Z_p_min* function also uses the scheduling formulation to find the minimum Z_p, attempting to schedule for increasing values of Z_p until a feasible solution is found. The *find_S_min* and *estimate_II_min* functions utilise a simplified version of the scheduling formulation that does not model the imperfectly nested resource constraints. While *find_S_min* is able to return the true minimum S for the given T (because S is only dependent on how the perfectly nested operations are scheduled), *estimate_II_min* returns an estimate of the minimum II for the given T. However, it will never return a value greater than the true minimum and so this value can still be used for generating lower bound schedule lengths.

estimate_S_tot_min is a much simpler function that does not attempt any scheduling. Instead each Z_i value is estimated based on the number of clock cycles required to complete all of the memory accesses nested perfectly or imperfectly up to and including loop level i, ignoring all dependences. The estimated Z_i values, along with the given value of S, allow an estimate of the minimum S_{tot} to be found using equation (4.4). The estimated value for the minimum S_{tot} will never exceed the true minimum and so can be used to generate lower bound schedule lengths.

The number of iterations of the innermost loop in the search will vary from target loop to target loop. However, the worst case size of the search can be calculated based on the worst case ranges of the T, S, Z_p and II variables. It can be shown that the worst case range of values for each variable is approximately Lat_{tot}, where Lat_{tot} is the sum of the latencies for all of the operations in the simplified data dependence graph. Hence the number of inner loop iterations will be of the order of Lat_{tot}^4. While this represents a potentially large number of iterations, for the test cases presented in Section 4.8 the search was found to converge after far fewer iterations.

4.6.2 Modulo Scheduling

Modulo scheduling of the simplified dependence graph is implemented using ILP as this provides a simple method for obtaining schedules of minimal cost. The cost functions used are discussed at the end of this section. We introduce a number of symbols to represent constants and variables in the ILP formulation and these are summarised in Table 4.1 for the reader's reference.

In the ILP formulation the start time of each node, v_n, in the simplified dependence graph, $G_s(V_s, E_s)$, is defined as an integer variable x_n. The latency of each operation is defined as an integer constant l_n. Each edge in E_s produces a linear constraint in the scheduling formulation, as described by inequality (4.6). d_{n1n2} again represents the number of loop iterations at the pipelined loop level between the dependent instances of nodes v_{n1} and v_{n2}. MII is an integer variable representing the minimum initiation interval that may be achieved after scheduling such that all dependence constraints will be met.

Table 4.1: Summary of the notation used to describe the module scheduling ILP formulation.

Notation	Type	Meaning
x_{n1}	Integer variable	Scheduled start time for node $n1$ in the first loop iteration
l_{n1}	Constant	Latency of node $n1$
d_{n1n2}	Constant	Number of loop iterations at the pipelined level between dependent instances of nodes $n1$ and $n2$
MII	Integer variable	Minimum initiation interval for the schedule produced
T	Constant	Number of cycles per pipeline stage
S	Constant	Number of pipeline stages
Z_p	Constant	Number of sets of imperfectly nested stages for all loop levels up to the pipelined level
Z_i	Integer variable	Number of sets of imperfectly nested stages for loop level i
st_n	Integer variable	The pipeline stage to which node n is assigned
d_{nt}	Binary variable	1 if node n is assigned to cycle t of any pipeline stage, and 0 otherwise
$ports_m$	Constant	Number of ports to memory m
d_{nst}	Binary variable	1 if node n is assigned to cycle t of pipeline stage s, and 0 otherwise
b_{an}	Binary variable	1 if node n is scheduled to an imperfect stage which executes after the perfectly nested stages, and 0 otherwise
$d_{n\rho st}$	Binary variable	1 if node n is assigned to cycle t of pipeline stage s in the set of stages denoted by ρ, and 0 otherwise (ρ is 0 for the perfectly nested stages and 1 for the imperfectly nested stages)
$slack_{mst}$	Integer variable	Number of unused ports to memory m on cycle t or pipeline stage s

The final initiation interval is calculated post modulo scheduling, based on the value of MII returned, using the methods described in the following section.

$$\forall(v_{n1}, v_{n2}) \in E_s, \quad x_{n2} + d_{n1n2} \cdot MII \geq x_{n1} + l_{n1} \tag{4.6}$$

Recall that within each individual modulo scheduling formulation the values of T, S and Z_p are constants. The perfectly nested operations may only be scheduled to start in the S perfect stages and every perfect operation must complete its execution before the end of the final perfect stage. These requirements lead to the constraints in inequalities (4.7) and (4.8), which assume the first perfect stage begins its execution at time $t = 0$. P is the set of all perfectly nested operations in the dependence graph.

$$\forall v_n \in P, \quad x_n + l_n \leq S \cdot T \tag{4.7}$$

$$\forall v_n \in P, \quad x_n \geq 0 \tag{4.8}$$

The imperfectly nested operations may be scheduled into any of the perfect or imperfect stages, but must all complete before the end of the final imperfect stage. The $Z_p \cdot S$ imperfect stages may be considered to execute both before and after the perfect stages and so the constraints in inequalities (4.9) and (4.10) must be met during scheduling. I is the set of all imperfectly nested operations in the dependence graph.

$$\forall v_n \in I, \quad x_n + l_n \leq Z_p \cdot S \cdot T \tag{4.9}$$

$$\forall v_n \in I, \quad x_n \geq -Z_p \cdot S \cdot T \tag{4.10}$$

For each level i above the innermost loop and below the pipelined loop, the value of Z_i (which is defined as an integer variable) must be determined during scheduling. Since the imperfectly nested operations may be scheduled to execute either before or after the perfect stages the Z_i values are constrained by both inequalities (4.11) and (4.12), where I_i is the set of operations nested imperfectly up to and including level i.

$$\forall v_n \in I_i, \quad Z_i \cdot S \cdot T \geq -x_n \tag{4.11}$$

$$\forall v_n \in I_i, \quad Z_i \cdot S \cdot T \geq x_n + l_n - S \cdot T \tag{4.12}$$

The remainder of the variables and constraints in our modulo scheduling formulation are required to model the resource constraints of the target system. For each physical memory, m, the value of cyc_m is re-calculated according to equation (4.2), this time including both the perfect and imperfect memory operations in the values of W_m and R_m. Let imp_m be the number of imperfectly nested accesses to memory m in the simplified data dependence graph. The resource constraints for memory m will take one of three forms depending on the values of cyc_m and imp_m.

1. $cyc_m \leq 1$: In this case it is possible to execute all accesses (both perfectly and imperfectly nested) to memory m in parallel. As such no resource constraints are required for the access operations to this memory.

2. $cyc_m > 1$ and $imp_m = 0$: In this case there are insufficient ports to execute every memory access in parallel, but it must still be possible to execute all S perfect stages in parallel. For each memory access operation, v_n, assigned to memory m a new integer variable, st_n, and T binary variables, d_{nt}[8], are defined. st_n defines the stage to which operation is assigned while the d_{nt} binary variables determine the cycle within the stage. d_{nt} is one if operation v_n is scheduled to begin on cycle t and zero otherwise. The start time of operation v_n is then constrained by equations (4.13) and (4.14). The constraints defined by inequality (4.15) ensure that, when all S perfect stages execute in parallel, no more accesses are scheduled to a single cycle than can be supported by the available ports, $ports_m$. P_m is the set of perfect memory accesses to memory m.

$$\forall v_n \in P_m, \quad x_n = st_n \cdot T + \sum_{t=0}^{T-1} t \cdot d_{nt} \tag{4.13}$$

$$\forall v_n \in P_m, \quad \sum_{t=0}^{T-1} d_{nt} = 1 \tag{4.14}$$

$$\forall 0 \leq t \leq T, \quad \sum_{v_n \in P_m} d_{nt} \leq ports_m \tag{4.15}$$

3. $cyc_m > 1$ and $imp_m > 0$: In this case the constraints must ensure that all of the perfect stages may execute in parallel without breaching the port constraints. They must also ensure that each imperfect stage uses no more ports on each cycle than the corresponding perfect stage. For each perfectly nested memory operation, v_n, assigned to memory m, an extra $S \cdot T$ binary variables, d_{nst}[9], are defined. d_{nst} is defined to be one if operation v_n is scheduled to begin on cycle t of perfect stage s and zero otherwise. The scheduled start time of v_n, x_n, is then constrained by

[8] $(0 \leq t < T)$.
[9] $(0 \leq s < S)$ and $(0 \leq t < T)$.

equations (4.16) and (4.17). P_m again represents the set of perfect memory accesses to memory m.

$$\forall v_n \in P_m, \quad x_n = \sum_{s=0}^{S-1} \sum_{t=0}^{T-1} (s \cdot T + t) \cdot d_{nst} \tag{4.16}$$

$$\forall v_n \in P_m, \quad \sum_{s=0}^{S-1} \sum_{t=0}^{T-1} d_{nst} = 1 \tag{4.17}$$

For each imperfectly nested memory operation, v_n, assigned to memory m, a further $S \cdot T \cdot (Z_p + 1)$ binary variables, $d_{n\rho st}$[10], are defined . $d_{n\rho st}$ is defined to be one if operation v_n is scheduled to cycle t of stage s. If ρ is zero the imperfect operation is scheduled to a perfect stage, otherwise it is scheduled to an imperfect stage. A further binary variable, ba_n, is also defined. ba_n is one if v_n is scheduled to start after the execution of the perfect stages and zero otherwise. The start time of v_n, x_n, is then constrained by equations (4.18) and (4.19). I_m represents the set of imperfect access operations assigned to memory m.

$$\forall v_n \in I_m,$$

$$x_n = \sum_{\rho=0}^{1} \sum_{s=0}^{S-1} \sum_{t=0}^{T-1} \Big((-\rho \cdot S \cdot T + s \cdot T + t) \cdot d_{n\rho st} \Big) + 2 \cdot ba_n \cdot S \cdot T \tag{4.18}$$

$$\forall v_n \in I_m, \quad \sum_{\rho=0}^{1} \sum_{s=0}^{S-1} \sum_{t=0}^{T-1} d_{n\rho st} = 1 \tag{4.19}$$

The resource constraints for memory m in the imperfect system are defined by inequalities (4.20) and (4.21). Inequality (4.20) defines the resource constraints for the perfect stages while Inequality (4.21) deals with the imperfect stages. Each imperfect stage is constrained to use no more memory ports than the corresponding perfect stage. In cases where not all of the memory ports are utilised on every cycle in the perfect stages, the $slack_{mst}$ integer variables allow the imperfect stages to make use of these 'spare' access slots.

[10] $(0 \leq \rho < Z_p)$, $(0 \leq s < S)$ and $(0 \leq t < T)$.

$$\forall\ 0 \leq t < T,\ \sum_{s=0}^{S-1} \left(\sum_{v_n \in P_m} d_{nst} + \sum_{v_n \in I_m} d_{n0st} + slack_{mst} \right) \leq ports_m \qquad (4.20)$$

$$\forall\ 0 \leq s < S,\ \forall\ 0 \leq t < T,$$

$$\sum_{v_n \in I_m} d_{n1st} \leq \sum_{v_n \in P_m} d_{nst} + \sum_{v_n \in I_m} d_{n0st} + slack_{mst} \qquad (4.21)$$

The modulo scheduling routine is called numerous times by the search routine with different values of T, S, and Z_p as inputs. The cost function which must be minimised varies depending on which function in the search which makes the scheduling call. The *schedule* and *find_S_tot_min* functions require the minimisation of S_{tot}, which is defined as an integer variable and whose value is determined by equation (4.4). The *schedule* function also places an upper bound on the value of MII. The *find_II_min* function uses MII as the cost function.

4.6.3 Calculating the Final Initiation Interval

Assume for now that the target loop has been modulo scheduled to produce a pipelined implementation with S stages for the perfectly nested operations, each of length T clock cycles. At this point some value for the initiation interval (II) may be selected, but this value must satisfy the following three conditions if the dependence and resource constraints of the system are to be honoured.

1. The modulo scheduling process returns a value of MII that represents the minimum initiation interval that satisfies all of the dependence constraints in the scheduled loop. The final initiation interval value selected must be greater than or equal to this value.

2. During modulo scheduling the operations of the loop body will be scheduled in stages of length T cycles. The process of modulo scheduling ensures that the stages are scheduled such that they may run in parallel for different loop iterations without breaching the resource constraints of the target platform. However, this only holds

true if all stages with overlapping executions have the same start time *i.e.* the start and end times of any stages executed in parallel must line up exactly. For this constraint to be met the value of II must be an integer multiple of T, as in equation (4.22) (where ii is an integer greater than 0).

$$II = ii.T \qquad (ii \geq 1) \tag{4.22}$$

3. The value of II must ensure that no two (or more) iterations are ever scheduled to use the same stage at the same time. Up to S iterations may be overlapped in the pipeline at once. All iterations will be separated by an integer number of stages (since $II = ii.T$). There are S stages in the pipeline, labeled 0 to $(S-1)$ in the order they execute to complete a single iteration of the loop body for simplicity. Let us define the *relative stage number* of each iteration k as the number of the stage being used by iteration 0 when iteration k begins[11]. The total number of stages that iteration 0 has executed when iteration k begins is $k.ii$ and so the *relative stage number* for iteration k will be $(k.ii \,\mathbf{mod}\, S)$. If all (up to) S iterations overlapped in the pipeline at the same time have a unique *relative stage number* then they will never require the same stage at the same time. Furthermore, iteration $\alpha.S+k$ (where α is any integer) will have the same *relative stage number* as iteration k. Therefore no iterations will ever require the same stage at the same time if the first S iterations have unique *relative stage numbers*, *i.e.*

$$\forall_{1 \leq k < S}, \forall_{0 \leq j < k} \qquad (k \cdot ii \,\mathbf{mod}\, S) \neq (j \cdot ii \,\mathbf{mod}\, S) \tag{4.23}$$

Condition (4.23) may be relaxed slightly. Depending on the value of II, it may occur that S iterations are never overlapped in the pipeline due to the time for a single iteration being less than S initiation intervals. Say, for instance, that iteration j ends before iteration k begins. In this case it would not matter if iteration j and iteration k share the same *relative stage number* since their executions will never overlap. However, due to the nature of the **mod** function, iteration j will have the

[11]The *relative stage number* of iteration 0 is defined as 0.

same *relative stage number* as iteration $j+S$. Iteration k can therefore only share the same *relative stage number* as iteration j if the execution of iteration j ends before the start of iteration k, and iteration k ends before the start of iteration $j+S$, *i.e.*

$$\left(k \cdot ii \geq j \cdot ii + S \cdot \prod_{i=0}^{p-1} N_i\right) AND \left(k \cdot ii + S \cdot \prod_{i=0}^{p-1} N_i \leq j \cdot ii + S \cdot ii\right) \tag{4.24}$$

Let *gcd* be the greatest common divisor of S and ii. Conditions (4.23) and (4.24) can be combined and simplified to give the following condition:

$$(gcd = 1)\ OR\ \left(\left(\frac{ii}{gcd} \geq \prod_{i=0}^{p-1} N_i\right)\ AND\ \left(\frac{(gcd-1) \cdot ii}{gcd} + \prod_{i=0}^{p-1} N_i\ \leq\ ii\right)\right) \tag{4.25}$$

The first term in the 'or' statement in (4.25) is a simplified version of (4.23). The second term is equivalent to (4.24), with the worst case values of k (assuming $gcd \neq 1$ since the second 'or' term only need be evaluated if the first term in the 'or' statement fails) used in each case. The derivation of the first and second terms in the 'or' statement of condition (4.25) from (4.23) and (4.24) respectively is included in Appendix B. Any value of ii such that condition (4.25) is satisfied will produce an initiation interval ($II = ii.T$) such that no two iterations will ever require the same stage at the same time.

From these conditions a simple method for finding the minimum II for a given modulo schedule, which will have associated with it fixed values of S, T and MII, can be derived.

- Find the smallest integer value of ii greater than zero such that $ii.T \geq MII$.
- Check condition (4.25) for the candidate ii. If the current ii fails then increment ii and recheck condition (4.25). Repeat this step until an ii value is found such that condition (4.25) is met.

It is worth noting that this process will always complete and find a valid initiation interval as the conditions in the two steps will be met by the first prime value of ii greater than or equal to MII, if not before.

4.7 Hardware Implementation

Extending loop pipelining above the innermost loop does not add any additional complexity to the generation of the resulting datapaths. The automatic generation of VHDL datapaths from pipelined schedules has received considerable attention in existing work [9,27] and no real extensions beyond current methods are required when moving above the innermost loop. However, the hardware control structures for loops pipelined above the innermost level have not previously been considered and this section proposes a novel method for their implementation.

The hardware controller for a pipelined loop must supply the following signals to the datapath:

1. A signal to indicate which of the T clock cycles in each stage is being executed. The stages run in lock step so the same signal supplies every stage.

2. A signal to indicate which of the stages in the pipeline are enabled. During the pipeline fill and flush (and if there are imperfect stages in the pipeline) the correct set of stages must be disabled or enabled to ensure correct operation.

3. A set of signals to indicate the current loop iteration being executed by each stage in the pipeline.

When pipelining at the innermost loop level these signals can be supplied by relatively simple circuits, such as those shown in Figures 4.2(a) and 4.2(b). A counter and shift register supply the correct loop index to each of the S stages while a comparator and a second shift register determine the enable signals. The circular shift register in Figure 4.2(a) tracks the current cycle in each stage. Execution of the pipeline is triggered by setting the reset input high for a single cycle. The simplicity of such a control scheme will generally create a relatively short critical path through the control logic which should be comparable to the critical path in the datapath logic. Hence a high maximum clock rate can generally be achieved. For any outer loop pipelining scheme to be worthwhile it must be possible to implement the controller in a comparably simple manner to ensure that any

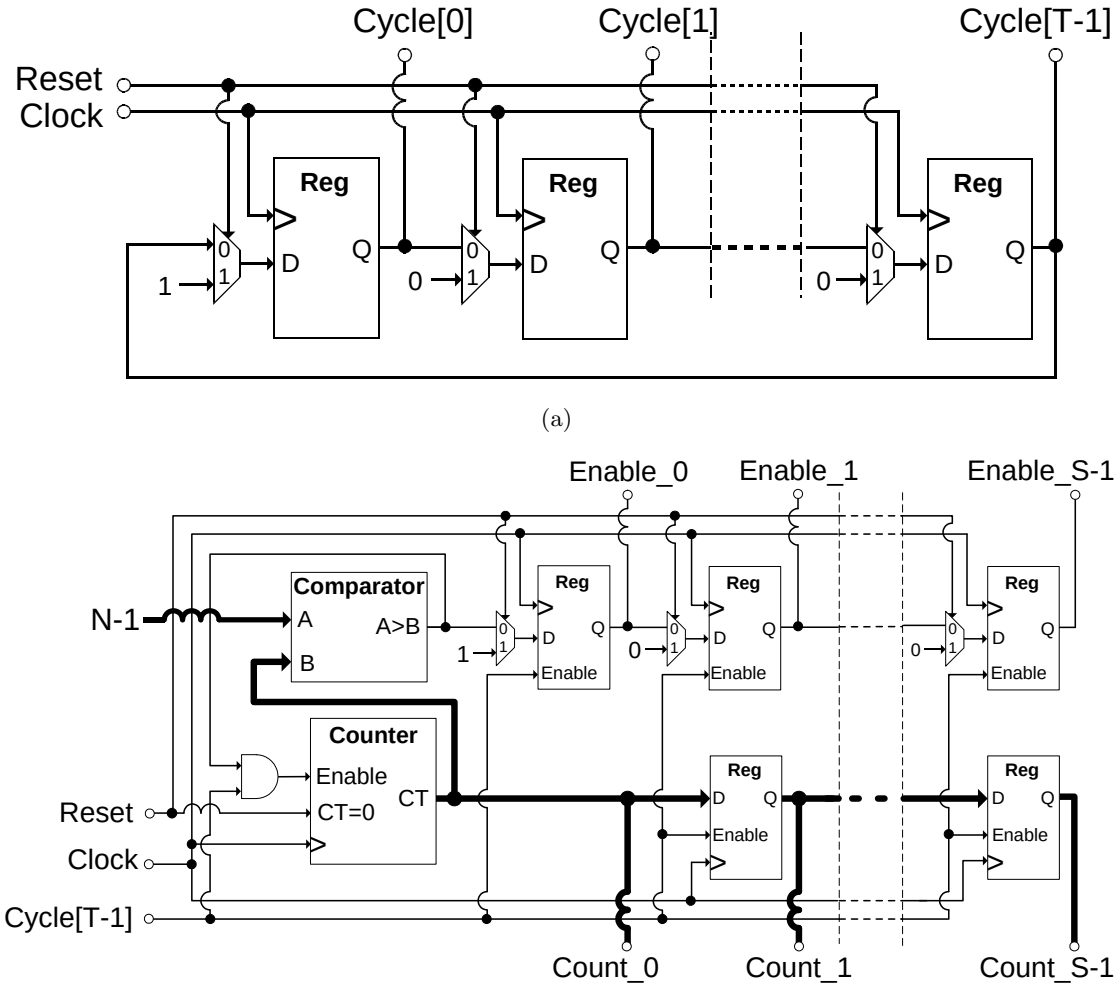


Figure 4.2: A simple controller design for a loop pipelined at the innermost level (a) A cyclical shift register to indicate the current cycle in each stage (b) Control logic to generate the loop index and enable signal for each stage. The bold bus lines are of width $log_2 N$, where N is the number of loop iterations.

drop in clock frequency is minimised. If this is not the case then any potential decrease in the schedule length will be canceled by the drop in clock rate.

In deriving a control scheme for outer loop pipelining let us first consider the generation of the loop index vectors for each stage. A counter is sufficient for this purpose in the innermost loop case because each stage executes the loop iterations in sequence (*i.e.* 0, 1, 2...). This is not the case when pipelining above the innermost loop. Consider the example schedule segment shown in Figure 4.3(a). Starting from the highlighted stage, with time progressing vertically downwards through the stages outlined in bold, stage 0

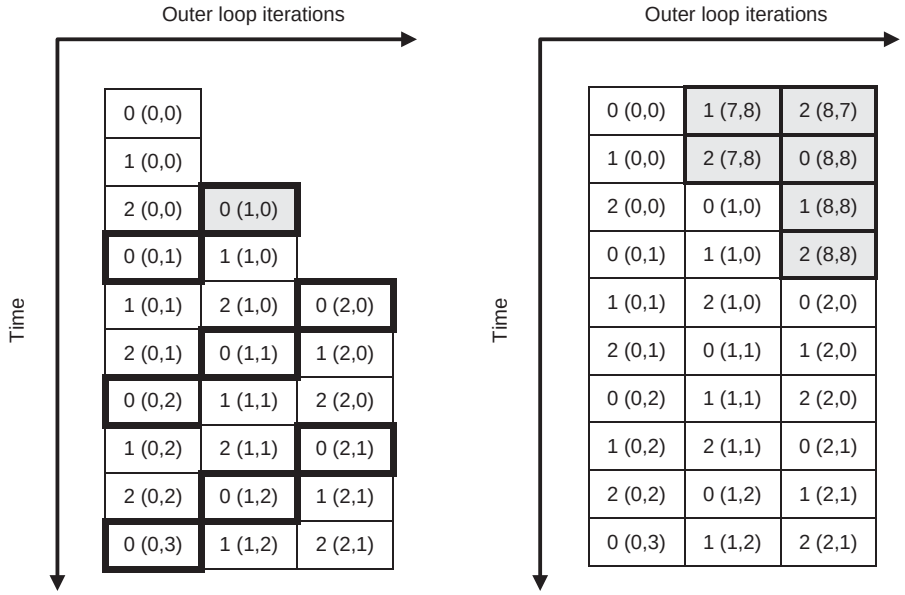


Figure 4.3: A section of an example schedule for a double nested loop pipelined at the outermost level. It is assumed the loop has 9 iterations at each level and it has been scheduled into a pipeline with 3 stages and an initiation interval of 2 stages. These numbers are chosen arbitrarily. Each box represents the execution of one pipeline stage. The number outside the brackets in each box is the stage number while the numbers in brackets represent the index vector for the loop iteration executed by each stage.

executes the loop iterations in the order *(1,0), (0,1), (2,0), (1,1), (0,2), (2,1)* and so on. This is a more complex pattern than a simple increment of the innermost loop index from one execution of a stage to the next. However, looking down the columns of the schedule it is apparent that stages 1 and 2 execute the same iteration as the immediately preceding instance of stage 0. Thus a shift register can still be used to supply all but stage 0 with the correct index vector, reusing the index vector from stage 0 as with inner loop pipelining. Furthermore, it is noted that the index vector for each instance of stage 0 is merely the index vector from the preceding instance of stage 2 with the inner loop index incremented[12].

The circuit in Figure 4.4 is therefore sufficient to provide the loop index to each stage for a single level in the loop. The circuit is duplicated for each loop level up to and

[12]When the inner loop index reaches the total iteration count for innermost loop it is reset to zero and the index for the next loop level is incremented.

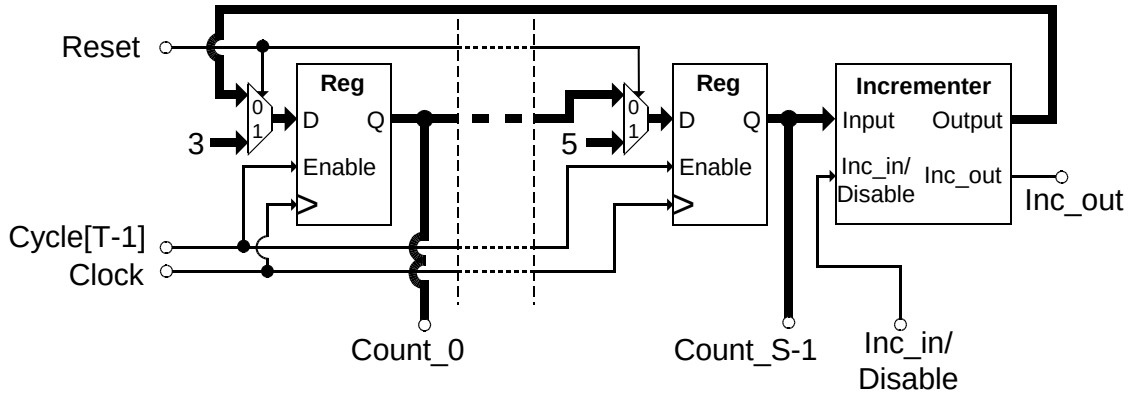
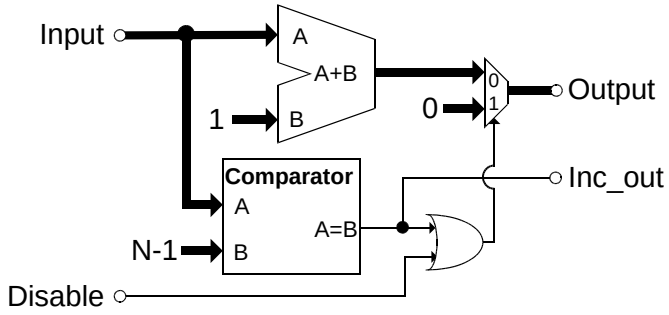


Figure 4.4: A circuit to generate the loop index for each stage at one level in the loop. The bold bus lines are of width $log_2 N$, where N is the number of loop iterations at the given loop level. The 'Incrementer' block is detailed in Figure 4.5. The input values on the '1' input of the multiplexors (3 and 5 in this Figure) represent the initial reset values for the loop indices. The values shown were chosen arbitrarily and are not significant.

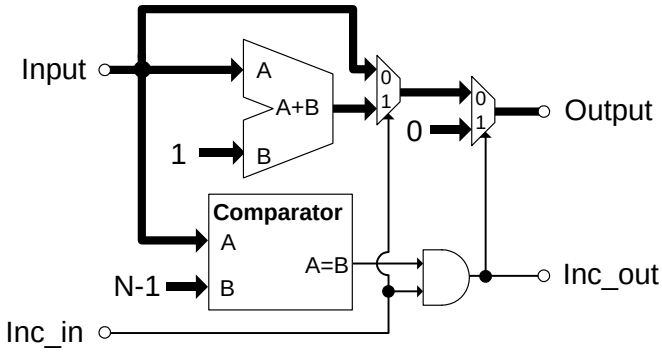
including the pipelined level to provide the complete index vector for each stage. The contents of the 'incrementer' block depend on the level in the loop. Figure 4.5 details the 'incrementer' design for the innermost loop level, the design for the pipelined loop level and the design for all other levels in between.

Due to the feedback loop present in the circuit in Figure 4.4 the index vectors for each stage must be initialised. These initial values may be derived simply from the proposed schedule and set as constants within the circuit. To derive the initial index vector for each stage the schedule is extended back to the start of the loop execution as shown in Figure 4.3(b). The index vectors in the highlighted stages are the values that would be present if the loop were executed repeatedly with no break between one execution and the next *i.e.* they are the index vectors for the end of the loop execution. As these index vectors pass through the 'incrementer' blocks at each level they will overflow back to the start of the loop execution, resulting in the desired initialisation. To ensure the correct operation of the loop the highlighted stages in Figure 4.3(b) will not be enabled and so no datapath operations will be executed for them.

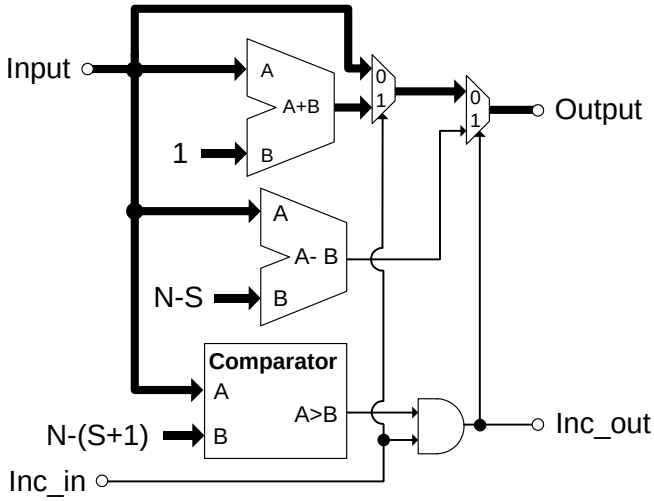
The 'Inc_out' signal generated by the 'incrementer' block at the pipelined level, referred to as 'Inc_top' from here on, may seem superfluous as there are no higher loop



(a)



(b)



(c)

Figure 4.5: The 'Incrementer' blocks used by the circuit in Figure 4.4 to update the loop indices. The bold bus lines are of width $log_2 N$, where N is the number of loop iterations at the given loop level (a) The 'Incrementer' for the innermost loop level (b) The 'Incrementer' for all loop levels between the innermost level and the pipelined level (c) The 'Incrementer' for the pipelined loop level.

levels for it to feed into. However, it is useful in the control of the generation of the correct enable signal for each stage. The logic controlling the enable signals for a pipeline with only perfectly nested stages is shown in Figure 4.6 and Figure 4.7. At the pipeline reset the enable signal for stage 0 is set high to begin the execution of the first iteration of the loop. The '1' is shifted through the register enabling each stage in turn. There is a feedback path from stage $S-1$ to stage 0 as each iteration at the pipelined loop level entails more than one iteration of the innermost loop and so a single stage must remain enabled. The initial values for the index vectors will cause the 'Inc_top' signal to go high $S-1$ times as iterations 2 through $S-1$ (at the pipelined level) begin their executions. This switches the multiplexer in Figure 4.6 so that an additional '1' is input into the shift register for each new iteration[13], eventually enabling all S stages. The 'Enable/Disable' block, detailed in Figure 4.7, counts the number of 'Inc_top' inputs received and, once all S stages have been enabled, it switches its output to '0'. The next occasion when 'Inc_top' goes high occurs when the pipeline flush begins at the end of the loop execution. As the final S iterations at the pipelined level end they again cause the index vectors to overflow as they pass through the 'incrementer' blocks, sending 'Inc_top' high a further S times. This again switches the multiplexer in Figure 4.6, but the output from the 'Enable/Disable' block is now '0' so increasing numbers of stages are disabled. When the last iteration terminates the last '1' in the shift register will be replaced with a '0' and all datapath operations will terminate.

Extending the pipeline controller to deal with imperfectly nested stages is relatively simple. The circuit shown in Figure 4.8 shows the circuit required to control the enable signal for each pipeline stage in the case where there is one set of imperfectly nested stages, and where the lowest loop level that requires their execution is $L-2$ (two levels above the innermost loop). This circuit can be adapted to a larger number of sets of imperfect stages with the addition of two further 'Shift Reg' blocks for each set of imperfect stages. The input to each shift register is a simple combination of the output from the the previous set of stages and the 'Inc_out' outputs from the incrementer blocks (Figure 4.5) for each loop level. In Figure 4.8 the 'enables[S-1..0]' signals control the enabling of the perfectly

[13]The output of the 'Enable/Disable' block is initially '1' after the reset.

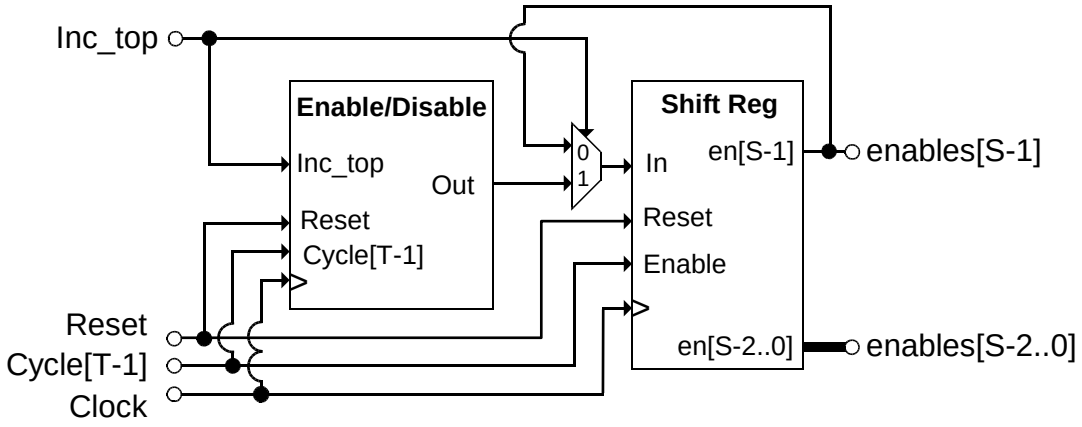


Figure 4.6: The logic to control the enable signal for each pipeline stage. The 'Enable/Disable' block is detailed in Figure 4.7. The 'Shift Reg' block is a shift register with S bits. 'en[0]' is the output from the first register in the chain. 'en[S-1]' is the output from the last register in the chain. At the reset 'en[0]' is set to '1' while all other bits are set to '0'. The 'Inc_top'input is connected to the 'Inc_out' from the 'Incrementer' block at the pipelined loop level.

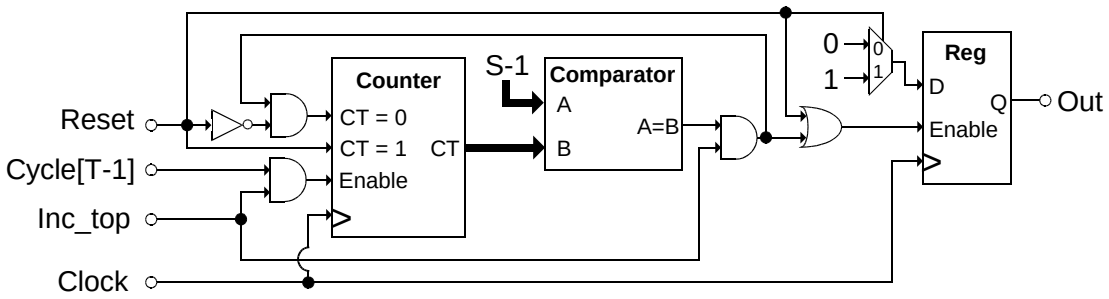


Figure 4.7: The 'Enable/Disable' block. The 'Inc_top'input is connected to the 'Inc_out' from the 'Incrementer' block at the pipelined loop level.

stages while 'enables_before[S-1..0]' and 'enables_after[S-1..0]' control the enabling of the imperfectly nested stages. Although there is only a single set of imperfectly nested stages in the pipeline, into which both the operations nested imperfectly before and after the innermost loop level are scheduled, the operations nested before and after the innermost level are enabled separately using the two sets of signals. This has been done for simplicity. Only the operations nested before the innermost loop should be enabled on the first iteration of the loop, while only the operations nested after the innermost loop should be enabled one the final loop iteration. Between consecutive iterations at levels requiring imperfectly nested operations to execute both the before and after operations are enabled.

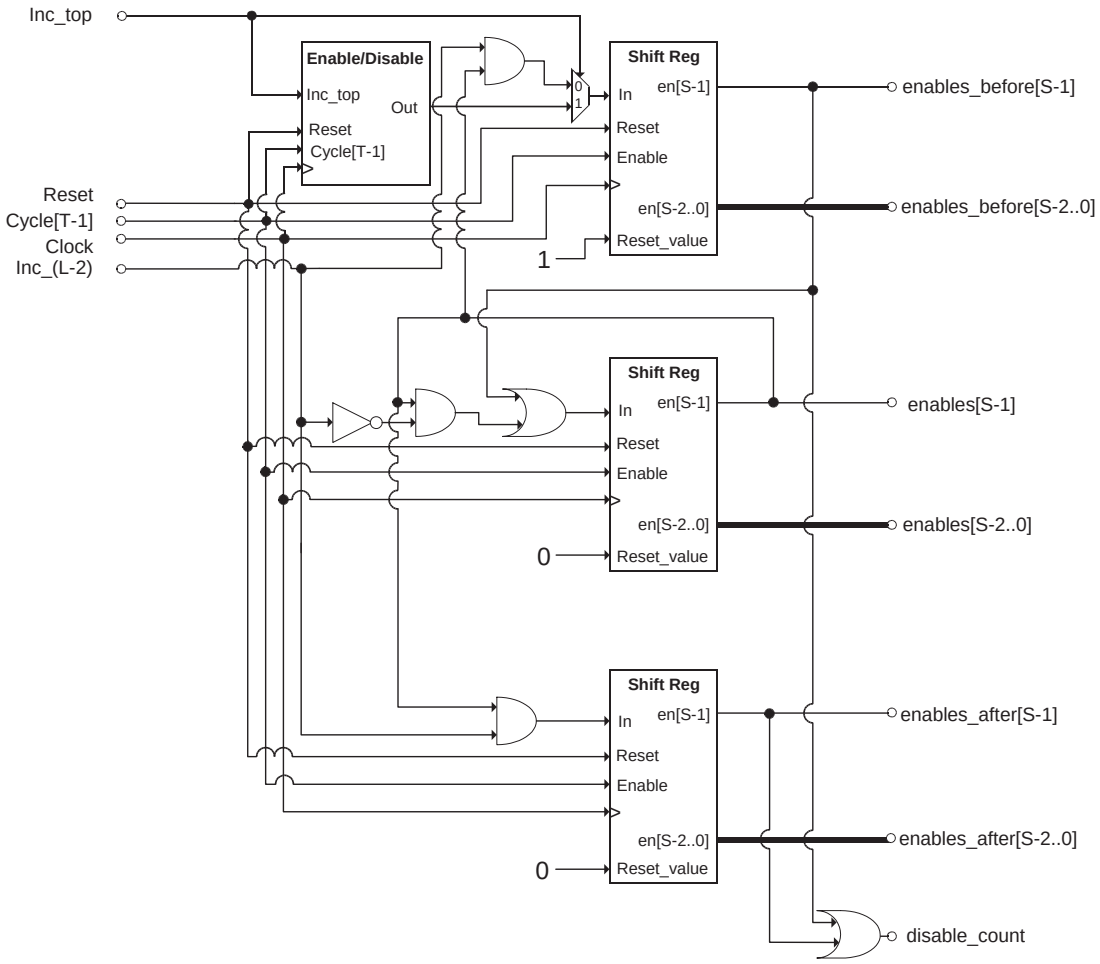


Figure 4.8: The logic to control the enable signals for an imperfectly nested loop with one set of imperfectly nested stages. The 'Enable/Disable' block is detailed in Figure 4.7. The 'Shift Reg' block is a shift register with S bits. 'en[0]' is the output from the first register in the chain. 'en[S-1]' is the output from the last register in the chain. At the reset 'en[0]' is set to 'Reset_value' while all other bits are set to '0'. The 'Inc_top'input is connected to the 'Inc_out' from the 'Incrementer' block at the pipelined loop level. The 'Inc_(L-2)' input is connected to the 'Inc_out' from the 'Incrementer' block two loop levels above the innermost loop.

The operation of the imperfectly nested controller in Figure 4.8 is similar to that of the perfectly nested controller in Figure 4.6. At the pipeline reset the enable signal for the first stage of the imperfect operations (nested before the innermost loop) is set high. This '1' is shifted through the register block, enabling each imperfect stage in turn, before enabling the perfect stages. The '1' cycles around the enable signals for the perfect stages, enabling each in turn before feeding back to perfect stage 0. At the end of any iteration at

the lowest loop level that requires the execution of the imperfect stages[14], the '1' is diverted from the perfect stages and enables imperfect stage 0 (before and after) instead. Once the imperfect stages have executed the '1' is fed back into perfect stage 0 and the process repeats. The 'Enable/Disable' block is unaltered from the perfectly nested case. It feeds an additional '1' into the first imperfect stage after each initiation interval has elapsed until all S parallel loop iterations have been enabled. At the end of the loop execution it replaces each '1' with a '0' until all the stages are disabled. The 'disable_count' output feeds the 'disable' input of the 'incrementer' block for the innermost loop level (Figure 4.5). This serves to disable the increment of the loop counts for the correct pipeline stage while the imperfectly nested operations execute. The same circuitry can then be used to supply the loop counts in the perfect and imperfect cases. The only addition required is the inclusion of an extra shift register (with S stages) to supply the loop counts for the imperfect operations nested imperfectly after the innermost loop. Before the execution of these operations the loop counts for the perfect stages will increment, but the non-incremented values are required. The additional shift register stores these values until the imperfect stages complete. The operations nested imperfectly before the innermost loop use the same loop counts as the perfect stages.

The hardware structures described will serve to control most loops pipelined above the innermost loop level, but there are special cases where variations on the blocks presented must be used. Examples of this are when the number of loop iterations at the pipelined level is less than the number of perfectly nested stages and when the initiation interval is greater than the number of innermost loop iterations in a single iteration at the pipelined level. Although the details for these cases have not been included, they have been considered and a small library of VHDL modules written to cover every possible combination of scheduling parameters (T, S, II and Z_p). The blocks are all parameterised and a simple tool has been developed to instantiate the correct blocks with the correct generic values to automatically generate a pipeline controller for the given values of T, S, II and Z_p.

[14]In Figure 4.8 this is the loop level nested two levels above the innermost loop, but could be any loop level in the general case.

4.8 Results

Our extended Single Dimension Software Pipelining algorithm has been used to pipeline each level in nine nested loops. The pipelined data path for each loop level is implemented manually in VHDL based on the schedule produced by our tool. The VHDL for the pipeline controller for each case is generated automatically by our scheduling tool from the set of parameterised component blocks described in the previous section. Four of the loops use a fixed point number representation in their datapaths. These are an image edge detection kernel, a full search motion estimation kernel, a complex fast Fourier transform (FFT) and a two dimensional median filter. The remaining five loops have floating point datapaths. These are a matrix-matrix multiply kernel, a 2D hydrodynamics fragment taken from the Livermore Loops [142], a successive over relaxation kernel (SOR) [143], the Minimum Residual (MINRES) algorithm [144] and an LU decomposition kernel [145]. The edge detection, motion estimation and median filter algorithms act upon 256x256 pixel images (8 bit fixed point). The search window for the motion estimator is +/-2 pixels and the median filter operates over a 5x5 pixel window. The matrix multiply, hydrodynamics, successive over relaxation, MINRES and LU decomposition kernels operate on 1000x1000 element (single precision) floating point matrices. The outermost loop (level 1) in the MINRES and SOR kernels is not pipelined as it is a while loop in both cases and the number of iterations is not fixed. For each case it is assumed that all of the image or matrix data accessed by the loop is stored in one bank of single port off-chip SRAM. An exception is made for the hydrodynamics kernel where it is assumed that five large matrices used are split across two banks of SRAM.

As stated in Section 4.6, the approach proposed in this work only finds optimal solutions for target loops with fixed loop bounds, but the LU decomposition kernel has variable loop bounds at two loop levels. In this case the average number of loop iterations was used in the place of fixed iteration counts and so the solutions produced may not be optimal. Furthermore, as was also stated in Section 4.6, this approach may only be applied to regularly nested loops and the LU decomposition kernel is irregularly nested. However, it is possible to transform the loops in the LU decomposition (the *dgefa* sub-

Table 4.2: Scheduling results for the edge detection (ED), motion estimation (ME), fast Fourier transform (FFT) and median filter (MED) kernels. The speedup figure is relative to the schedule length obtained when the innermost loop is pipelined. The average parallelism is the ratio of the completely sequential schedule length to the pipelined schedule length. The bandwidth utilisation column lists the percentage of the available off-chip memory bandwidth used.

Loop	Level	S	T	II	Z_p	Cycles	Average Parallelism	Speedup	Bandwidth Utilisation
ED	4	3	2	2	0	1,935,480	2.40	1.000	60.31%
ED	3	3	2	2	0	1,419,352	3.27	1.364	82.16%
ED	2	3	2	2	1	1,297,432	3.68	1.492	94.54%
ED	1	3	2	2	1	1,295,408	3.69	1.494	94.60%
ME	6	3	2	2	0	3,686,400	3.55	1.000	88.92%
ME	5	4	2	2	0	3,315,200	3.95	1.112	98.84%
ME	4	5	2	2	0	3,287,040	3.99	1.121	99.73%
ME	3	5	2	2	0	3,278,848	3.99	1.124	99.96%
ME	2	4	2	2	1	3,277,536	3.99	1.125	99.99%
ME	1	4	2	2	1	3,277,326	3.99	1.125	99.99%
FFT	3	2	8	8	0	41,040	4.14	1.000	99.84%
FFT	2	2	8	4120	0	45,272	3.75	0.906	90.45%
MED	5	3	2	2	0	49,152,000	3.03	1.000	0.0%
MED	4	5	1	6	1	35,717,120	5.63	1.376	4.58%
MED	3	5	1	476	1	164,429,824	1.01	0.299	1.01%
MED	2	8	1	1	1	21,368,576	7.75	2.300	7.66%
MED	1	8	1	1	1	21,364,751	7.75	2.301	7.67%

routine in Figure 3.3) into a regular form that can be tackled using the tool developed in this work. The C codes for both the untransformed LU decomposition, similar to that in Figure 3.3 but with the function calls expanded for clarity, and the transformed version are provided in Appendix C. Similar transformations are also applied to the Minimum Residual algorithm to merge distinct loops into a single loop, as shown in Appendix C. The C code for the remaining benchmark algorithms is given for the readers reference in Appendix D.

4.8.1 Scheduling Results

The scheduling results for each level in the four fixed point test loops are detailed in Table 4.2 and the results for the five floating point loops are detailed in Table 4.3. In every case, apart from the FFT example, pipelining above the innermost loop level does yield a

Table 4.3: Scheduling results for the matrix-matrix multiply (MMM), hydrodynamics (HD), successive over relaxation (SOR), MINRES (MIN) and LU decomposition (LU) kernels. The speedup figure is relative to the schedule length obtained when the innermost loop is pipelined. The average parallelism is the ratio of the completely sequential schedule length to the pipelined schedule length. The bandwidth utilisation column lists the percentage of the available off-chip memory bandwidth used.

Loop	Level	S	T	II	Z_p	Cycles	Average Parallelism	Speedup	Bandwidth Utilisation
MMM	3	3	7	2	0	7,014,000,000	2.42	1.000	28.52%
MMM	2	8	2	2	1	2,002,030,000	8.49	3.503	99.94%
MMM	1	8	2	2	1	2,002,000,030	8.49	3.503	99.94%
HD	2	8	6	42	0	42,006,000	1.95	1.000	13.09%
HD	1	8	6	42	0	6,000,294	13.67	7.001	91.61%
SOR	3	3	7	7	0	7,014,000	2.71	1.000	28.51%
SOR	2	20	2	2	1	2,002,078	9.51	3.503	99.99%
MIN	3	4	7	7	0	7,021,000	2.42	1.000	28.48%
MIN	2	25	2	8	1	2,002,242	8.50	3.507	99.99%
LU	3	6	3	3	0	1,010,524,464	5.95	1.000	98.83%
LU	2	6	3	3	1	1,008,291,933	5.97	1.002	99.12%
LU	1	9	2	36050	1	678,002,026	8.88	1.49	98.3%

shorter schedule. The FFT kernel has a long loop carried dependence at the outer loop level and so there is no advantage in pipelining above the innermost loop. However, since the pipelining methodology presented also considers the innermost loop, no performance is lost compared to an inner-loop-only methodology. While this is of interest, the original Single Dimension Software Pipelining work [13] has already demonstrated the benefits of extending pipelining above the innermost loop. However, the results also demonstrate that our extensions to the SSP methodology can offer gains over the existing work. For example, when pipelining loop level 1 of the hydrodynamics kernel the optimum stage length is found to be 6 while the initiation interval is 42. If T were forced to take the same value as II the minimum stage length would be 42, leading to a schedule that is seven times longer than that presented here. The results for the motion estimation kernel demonstrate the potential benefit of searching the available solution space. For loop level 5 the minimum number of stages when T is minimised is 3. With 3 stages in the pipeline the minimum II of 2 cycles may also be achieved, so 3 stages appears to be optimal. However, the scheduling search increases S to 4 stages as there are 16 iterations at the

pipelined level and making S a factor of N_p minimises the schedule length. When levels 3 and 4 are pipelined the number of perfect stages is first increased to 4 to accommodate extra imperfectly nested instructions (allowing Z_p to be zero), and then increased to five so that it is again a factor of N_p (which is 5 in both cases).

This work assumes that off-chip memory bandwidth is the limiting factor in the performance of FPGA based designs, so for each loop the utilisation of the off-chip memory bandwidth should ideally be 100%. If this is not the case either there is another limiting factor or the pipelining methods have failed to find an efficient solution. The memory bandwidth utilisation figures for each loop are presented in Tables 4.2 and 4.3 and no design achieves the desired 100% utilisation. However, most designs do achieve over 98% efficiency. The lost 2% (or less) is down to the flush and fill of the pipeline at the end of each loop iteration above the pipelined level and incomplete usage of the memory ports by imperfectly nested loop operations. Some loops do achieve noticeably lower bandwidth utilisation than 98%, but this can be explained in each case. The Edge Detection kernel has two small loops (only 3 iterations) at its innermost levels and the frequent flush and fills when pipelining at these levels cause the significantly reduced bandwidth utilisation. This also leads to reduced bandwidth utilisation when pipelining at the two outer loop levels because the loop requires one set of imperfectly nested pipeline stages to be executed after every 9 inner loop iterations. During the imperfectly nested operations the memory port is only used every other cycle because the imperfect operations require only one memory access when the perfectly nested loop operations would require two memory accesses. This wastes 5% of the memory bandwidth.

The solutions for the Matrix-Matrix Multiply, Hydrodynamics, SOR and MINRES kernels pipelined at the innermost loop level have very poor memory bandwidth utilisations, but this is due to the large initiation intervals caused by the loop carried dependences in each algorithm at the innermost level. Pipelining at a higher loop level with no dependences allows much higher bandwidth utilisation. The Hydrodynamics kernel loses 8.3% of memory bandwidth due to an uneven distribution of memory accesses across the two off-chip SRAMs available. There are 13 accesses at the innermost loop to be divided

across 2 ports. This leaves 6 accesses on one port and 5 on another and means that one port goes unused for one cycle in every 6. The most noticeable failure to efficiently utilise the off-chip memory bandwidth occurs with the Median Filter kernel. This is because the three innermost loop levels of the kernel do not have any accesses to the off-chip memory. The two outer loops iterate over the image pixels and the 3 inner loops iterate over the filtering window and sort the pixels into ascending order of magnitude, requiring no further pixel data to be read. As a result the algorithm is bound by the number of operations that may be scheduled in parallel and not memory bandwidth. Because the algorithm presented here can only parallelise the operations from a single iteration of the innermost loop the parallelism is limited by the number of operations at the innermost loop level, which is very small in this kernel. To achieve improved parallelism the ability to unroll the loop at one of more loops levels would be required.

4.8.2 FPGA Implementation Results

Table 4.4 details the performance results for eight of the test loops when the pipelined hardware accelerator for each loop level is targeted to an Altera Stratix II FPGA, specifically an EP2S15 part of the fastest speed grade (C3). Each design was synthesised, placed and routed using the Design Space Explorer utility of the Altera Quartus II (v9.1) tool. For each case Design Space Explorer was set to search the place and route options for 8 different starting seed values with effort set to maximum and all physical synthesis optimisations enabled. The hardware accelerator has not been implemented for either loop level of the FFT as there are no scheduling gains in moving to the outer loop. Likewise, the hardware accelerator for pipelining at level 3 in the median filter has not been implemented as it offers no gains over the inner loop. Only the control and datapath operations from the levels up to and including the pipelined loop level are targeted to the FPGA, with the remaining loop levels executed on a host microprocessor. The design of the pipeline controller is such that two clock cycles are required to initialise the pipeline each time it is called by the host system. The additional cycles have been added to the schedule lengths in Table 4.2 and Table 4.3 to produce those shown in Table 4.4. For each loop

Table 4.4: Hardware implementation results for the test loops. The speedup figure is relative to the innermost loop solution. ALUTs (Adaptive Lookup Tables) are the basic configurable elements in the Stratix II family of FPGAs.

Loop	Level	ALUTs	Registers	Fmax (MHz)	Cycles	Time (s)	Speedup
ED	4	126	109	500	2,129,028	0.0043	1.000
ED	3	130	111	476	1,548,384	0.0033	1.303
ED	2	271	183	482	1,297,940	0.0027	1.593
ED	1	292	235	475	1,295,410	0.0027	1.593
ME	6	95	145	500	3,891,200	0.0077	1.000
ME	5	116	165	500	3,328,000	0.0067	1.169
ME	4	194	273	349	3,289,600	0.0094	0.827
ME	3	206	284	378	3,279,360	0.0087	0.894
ME	2	307	276	473	3,277,568	0.0069	1.127
ME	1	329	285	476	3,277,328	0.0069	1.128
MED	5	51	58	500	52,428,800	0.1049	1.000
MED	4	163	216	500	36,372,480	0.0727	1.443
MED	2	243	474	500	21,369,088	0.0427	2.454
MED	1	317	539	500	21,364,753	0.0427	2.454
MMM	3	834	847	250	7,016,000,000	28.06	1.000
MMM	2	1067	1307	247	2,002,032,000	8.10	3.464
MMM	1	1175	1360	251	2,002,000,032	7.97	3.520
HD	2	3958	4450	234	42,008,000	0.180	1.000
HD	1	4092	4453	238	6,000,296	0.026	7.001
SOR	3	815	723	236	7,016,000	0.030	1.000
SOR	2	4600	8790	232	2,002,080	0.009	3.448
MIN	3	2448	4573	266	7,023,000	0.026	1.000
MIN	2	4074	7405	238	2,002,244	0.008	3.090
LU	3	807	1033	256	1,011,527,460	3.951	1.000
LU	2	1255	1725	260	1,008,293,931	3.878	1.019
LU	1	3158	4965	249	678,002,028	2.723	1.451

level the time taken for scheduling was less then 10 seconds, with most completing in less than 1 second, which may be considered negligible when compared to the minutes taken for synthesis and place and route.

The results in Table 4.4 show that, in all eight cases, the optimum (fastest) solution occurs when pipelining above the innermost loop. However, it should be noted that the fastest implementation does not usually coincide with the shortest schedule as there is some degradation in the clock frequency of the pipelines as we move towards the outermost loop. In every case the fastest implementation occurs one to three loop levels above the innermost loop as these levels offer the best tradeoff between the scheduling gains and the

clock frequency. This implies that the pipelining approach presented here offers advantages over the standard innermost only approaches used in existing hardware compilers [9, 64, 79, 89, 92, 93, 101, 104].

The degradation in the maximum clock frequency varies across the eight implemented loops. The edge detection datapath is a small, simple circuit and so the critical path through the complete design lies within the controller. Hence we see decline in the clock frequency as the loop pipelining moves up from level 4 to 3 and the controller becomes more complex. The motion estimator also has a relatively simple datapath and so the critical path for levels 5 and 6 again lies within the controller. There is a sharp drop in the clock rate as we move up to levels 3 and 4, but this is not due to the controller. Levels 3 and 4 require the implementation of imperfectly nested operations which increase the datapath complexity, moving the critical path into the datapath. When levels 1 and 2 are pipelined the imperfect operations are scheduled differently and this reduces the length of the critical path, allowing the clock rate to increase above that achieved for levels 3 and 4. The arithmetic units in the floating point implementations form the critical path in each case and so there is virtually no degradation in clock frequency for these loops.

There is an increase in FPGA resource usage as pipelining moves towards the outermost loop. For simple circuits such as the edge detector and the motion estimator the fastest solution has an ALUT and register usage roughly double that for the innermost loop. This is because the edge detector and motion estimator have datapaths that are relatively small and therefore comparable to the size of the controller. Hence a significant increase in the size of the controller leads to a significant increase in the overall resource usage. The more complex datapaths of the matrix multiplication and hydrodynamics kernels lead to less noticeable increases in the resource usage as we move towards the outer loops. This is because the controller is small relative to the floating point units used in these examples and so the relative increase in resource usage is also small. The increased resource usage for the outer loops of the MINRES, SOR and LU decomposition examples is mainly due to the inclusion of extra imperfectly nested floating point operations. In every case the designer must decide whether the increase in speed achieved in pipelining

at an outer loop level is necessary, or warrants the extra resource usage incurred.

As explained in Section 3.5 the heuristic algorithms used to place and route FPGAs lead to solutions that may be sub-optimal, *i.e.* the maximum clock frequency achieved may be lower than is actually possible. The potential error in the clock frequency is estimated to be between 5 and 10% so it is possible that one or more inner loop solutions are 10% slower than possible. In the worst case the solutions pipelined above the innermost level could have achieved their maximum clock frequency so the overall speedup reported here could be 10% higher than the true figure. However the speedup achieved over the innermost solution in each case is at least 16.9% so the best pipelining solution for each loop is still above the innermost level.

4.8.3 Comparison with Loop Interchange and Inner Loop Pipelining

While it has been shown that outer loop pipelining can offer advantages over the direct implementation of inner loop only methods, the role of loop interchange has not yet been considered. The matrix multiplication example has a loop carried dependence at the inner loop level that forces an initiation interval of 7 cycles when pipelining the innermost loop. However, there are no dependences at the outer loop levels, so interchanging either of the two outer levels to the innermost loop and then pipelining will leave the initiation interval bound only by the resource constraints, as is the case when pipelining above the innermost loop. However, despite the apparent similarity in the results produced for this case, outer loop pipelining does still offer a number of advantages over loop interchange and inner loop pipelining. Firstly, as pointed out in [13], some loops may not be interchanged due to dependences. Also, as with the hydrodynamics kernel, it is possible for there to be loop carried dependences at all loop levels. In the case of the hydrodynamics kernel the initiation interval will be 42 cycles, no matter how the loop is interchanged. Hence interchange and inner loop pipelining will produce a solution that is roughly 7 times slower than our approach.

Another advantage of the approach presented here is the ability to deal with imperfectly nested operations, especially imperfectly nested memory accesses. If there are

operations nested imperfectly at any level which we wish to interchange to the innermost loop, these operations must be moved into the innermost loop (and their executions guarded against). This may force a larger number of stages in the pipeline than our approach can offer. It may also force a larger stage length if the operations are memory accesses since the minimum stage length (T) is determined by the numbers of perfectly nested accesses to each memory. The matrix multiplication is a good example of this as there is a write to the external memory nested above the innermost loop. Interchanging either outer loop to the inner loop will force this write into the innermost loop and increase the minimum T from 2 to 3 cycles, reducing the speed of the final solution to roughly two thirds of that offered by our approach.

Two other potential advantages of our approach over interchange and inner loop pipelining are the reduction in the number of cycles spent flushing and filling the pipeline and the potential for data reuse. All of the speedup gained in the motion estimation example is gained because the outer loop pipelines are filled and flushed less frequently. In the LU decomposition example it is possible to buffer a column of matrix data on chip when the outer loop is pipelined. This reduces the number of perfectly nested memory accesses to the external memory from 3 to 2 per iteration, allowing a minimum T of 2 cycles instead of 3.

While our approach can offer advantages over interchange combined with inner loop pipelining, that does not mean that loop interchange has no role in improving results achieved when pipelining above the innermost loop. However, the potential gains of combining loop interchange and outer loop pipelining in hardware have not yet been considered and this is left as future work. It should be noted however that interchange was considered in combination with outer loop pipelining in the original SSP work [13] and was shown to be of benefit.

4.9 Summary

In this chapter an existing methodology for pipelining software loops above the innermost loop level has been adapted for use in generating FPGA based hardware coprocessors. The existing Single Dimension Software Pipelining approach has been extended to allow the initiation interval and stage length of a pipeline to take different values, offering an improvement in performance of 7 times in one example. A simplified method for dealing with imperfectly nested instructions has also been introduced which reduces control complexity. Furthermore, a search of the scheduling space has been developed such that the schedule with the shortest execution time (in clock cycles) is found.

The scheduling tool has been applied to nine test loops. In all but one case, when the resulting coprocessors are targeted to an Altera Stratix II FPGA, the fastest solution is found when the loop is pipelined one to three levels above the innermost loop. While there may be degradation in the clock frequency of the resulting hardware when pipelining is extended above the innermost loop, the decreases in the schedule length have been shown to outweigh this factor. As a result speedups over the innermost loop solution were achieved ranging from 1 (no speedup) to 7 times, with an average speedup of 2.9 times. These results indicate that, while pipelining above the innermost loop may not provide significant gains in every case, adding this capability to the toolbox of transformations used by hardware compilers could certainly be of use in exploiting parallelism in hardware coprocessors. This seems especially true when targeting floating point kernels as the long latencies of the (normally) deeply pipelined arithmetic units can lead to long loop carried dependences and large initiation intervals.

Chapter 5

Memory Optimisation

5.1 Introduction

In the previous chapter an outer loop pipelining approach was developed that took a fixed memory subsystem as an input and optimised around it. In this chapter memory optimisation steps are integrated into the scheduling framework to automatically allocate array data to physical memories and infer data reuse structures, producing a schedule and supporting memory subsystem that are mutually co-optimised.

Data may be reused where an array element is read multiple times, or written to and then read. By buffering array elements that may be reused in an on-chip scratch pad memory or FIFO one may reduce accesses to the off-chip memories, which often form the bottlenecks in FPGA based designs. In a typical nested loop there may be a variety of options for exploiting on-chip data reuse. There may also be some freedom to allocate arrays to the physical memories to maximise parallel access. Which reuse options are selected may affect how the arrays should be assigned to the memories (as the exploitation of data reuse reduces the number of accesses to a given array), and both of these issues will affect how the loop may be scheduled to minimise execution time. Array to memory placement and data reuse schemes for FPGAs have received significant attention in previous work, and some methodologies have been presented to link these issues to scheduling [9, 36, 65, 78, 82]. However, despite the existing work, to the best of

our knowledge, there is no work that brings together the various scratch pad memory and FIFO based data reuse schemes, that links the selection of reuse options to array to memory placement, and that links both of these issues to scheduling such that the shortest schedule is found for a given loop on a given target platform.

While the work in [82], [78] and [36] integrates array to memory assignment with scheduling, data reuse decisions are not included. [65] and [36] include data reuse, but do not link this to array to memory placement or provide a framework for selecting the optimal set of reuse options. [9] produces autonomous buffers for 'windows' of data (as used in image processing) to maximise pipeline speed, but does not support other forms of data reuse. The contribution of this chapter is to provide a single Integer Linear Programming formulation to simultaneously place arrays in the available memories and select the appropriate data reuse options form the available set. Three cost functions are presented to be used when optimising the memory subsystem, and it is shown how these can be integrated into the existing loop pipelining framework to co-optimise the schedule and the supporting memory subsystem, which normally acts as the bottleneck in FPGA based designs.

The remainder of this chapter is divided into seven further sections. Section 5.2 reviews the outer loop pipelining framework into which the memory optimisation approach is integrated. Section 5.3 describes the formulation for array to memory assignment, while Section 5.4 details the data reuse options included in this methodology and Section 5.5 describes the integration of the data reuse selection problem with array to memory assignment. In Sections 5.6 and 5.7 it is shown how the memory subsystem is iteratively updated during scheduling. Section 5.8 presents the results obtained when this methodology is applied to seven benchmark algorithms and Section 5.9 summarises the chapter.

5.2 Background

In the previous chapter an approach for extending an existing outer loop pipelining approach [13], originally developed for software for VLIW processors, was extended for FPGA

Algorithm 2 : Searching the pipelining solution space for each loop level. Z_p represents the number of stages required to accommodate all of the imperfectly nested operations up to the pipelined level. The comments (//) indicate where functions should be inserted to update the memory subsystem during scheduling

```
1: // memory_minimise_T();
2: T = find_T_min();
3: while ( T < bound_T(T) ) do
4:    // memory_minimise_S(T);
5:    S = find_S_min(T);
6:    while ( S < bound_S(T, S) ) do
7:       // memory_minimise_Z_p(T, S);
8:       Z_p = find_Z_p_min(T, S);
9:       while ( Z_p < bound_Z_p(T, S, Z_p) ) do
10:         // memory_minimise_II(T, S, Z_p);
11:         II = find_II_min(T, S, Z_p);
12:         while ( II < bound_II(T, S, Z_p, II) ) do
13:            // memory_minimise_S_tot(T, S, Z_p, II);
14:            cycles = schedule(T, S, Z_p, II);
15:            best = min(cycles, best);
16:            II++;
17:         end while
18:         Z_p++;
19:      end while
20:      S++;
21:   end while
22:   T++;
23: end while
```

hardware. This approach works by overlapping the executions of iterations at a given level in a nested loop. The iterations of all levels nested below the pipelined level are executed sequentially. A new iteration at the pipelined level is started every II cycles and iterations of all loop levels above the pipelined level are executed sequentially. Unlike inner loop pipelining, where the sole goal is to minimise the initiation interval (II) of the pipeline, outer loop pipelining presents a more complicated solution space where a number of scheduling parameters may be traded. A search scheme was presented to find the set of values, such that the schedule length is minimised, for the number of clock cycles per pipeline stage, T, the number of pipeline stages for the perfectly nested operations, S, the initiation interval, II, and the number of stages for the imperfectly nested operations at each level in the loop, Z_i (where i is the loop level). The simplified pseudo code for this search is given in Algorithm 2. Where variables are passed as inputs to a function it implies that the function must work within a fixed values for these variables. For example, the function 'find_S$_{min}$(T)' must minimise S for a fixed value of T.

In the previous chapter it was assumed that an array to memory map is supplied as an input to the scheduling tool, and there is no exploitation of data reuse. Ideally one would like to determine the optimum data reuse set and array to memory placement prior to scheduling as this would allow us to retain the original methodology. However, as all of the scheduling parameters can be affected by the properties of the memory subsystem, this may not be a practical solution. As the scheduling search progresses the desired properties from the memory subsystem will change. For instance, at first one would require the memory subsystem such that T is minimised, but as the search progresses the value of T may be increased in favour of minimising S or II. Hence this work proposes that the memory subsystem is continually updated during scheduling. The commented lines in Algorithm 2 show how functions to update the data reuse selection and array to memory placement should be added to the scheduling search so that both the schedule and memory subsystem can be co-optimised to produce the fastest pipelined solution.

The goal of the work presented in this chapter is to find the fastest possible pipelined implementation for a nested loop. However, there are currently a number of restrictions to the methodology presented here. Firstly, no automated loop unrolling strategies have yet been considered, though the user may unroll the loop manually and input this unrolled version. The work in this chapter also does not consider array partitioning or duplication, though these may again be specified manually by the user. These restrictions are tackled in the following chapter.

5.3 Array to Memory Placement

Existing work on mapping arrays to memories in FPGA based systems considers the problem of assigning arrays to specific blocks of on chip or off-chip memory [36, 78]. However, as the number of embedded memory blocks on modern FPGAs has increased, this approach has become increasingly impractical as it leads to a large solution space with many equivalent solutions. A more practical approach may be to group the arrays into *logical memories* and to then assign each logical memory to one of the types of memory resource available, rather than any particular instance of a given type.

5.3.1 Problem Formulation

This problem has been formulated using Integer Linear Programming (ILP) [80]. We are given a set of N arrays, $\mathbf{A} = \{a_1, ..., a_N\}$, and a set of M memory resource types, $\mathbf{T} = \{t_1, ..., t_M\}$. There are num_k instances (banks) of type t_k. Embedded FPGA memories may generally be configured to offer different combinations of width (word length) and depth (number of words) [141]. Each t_k is therefore considered to have con_k configurations. The arrays are mapped to a set of N logical memories[1], $\mathbf{L} = \{L_1, ..., L_N\}$, by the set of binary variables, d_{ij}. d_{ij} is one if array a_i is mapped to logical memory l_j and zero otherwise. The logical memories are mapped to the memory types by another set of binary variables, b_{jkm}, such that b_{jkm} is one if logical memory l_j is mapped to banks of resource type t_k using configuration m. To determine how many banks of each resource type are consumed by each logical memory a set of integer variables, $banks_{jkm}$, are introduced, where each value of $banks_{jkm}$ denotes the number of banks of type t_k (under configuration m) used by memory l_j. For a valid array to memory assignment the constraints represented by equations (5.1) to (5.5) must be satisfied. W_{Mkm} represents the number of words in a single bank of type t_k when using configuration m. W_{Aikm} represents the number of words in array a_i when it is implemented in configuration m of memory type t_k. Both W_{Mkm} and W_{Aikm} are constants. The number of words in an array may vary depending on the resource type and configuration because multiple elements of an array may be packed into a single (larger) memory word if the array is only ever read during the loop.

$$\forall a_i \in \mathbf{A}, \qquad \sum_{j=1}^{N} d_{ij} = 1 \tag{5.1}$$

$$\forall l_j \in \mathbf{L}, \qquad \sum_{k=1}^{M} \sum_{m=1}^{con_k} b_{jkm} \leq 1 \tag{5.2}$$

$$\forall l_j \in \mathbf{L}, \qquad N \cdot \sum_{k=1}^{M} \sum_{m=1}^{con_k} b_{jkm} \geq \sum_{i=1}^{N} d_{ij} \tag{5.3}$$

[1] One possible solution is to assign each array to a different memory, so N is the minimum number of logical memories that must be included for a complete solution space.

$$\forall t_k \in \mathbf{T}, \qquad \sum_{j=1}^{N} \sum_{m=1}^{con_k} banks_{jkm} \leq num_k \tag{5.4}$$

$$\forall l_j \in \mathbf{L}, \forall t_k \in \mathbf{T}, \forall m \in (1 : con_k),$$

$$banks_{jkm} \cdot W_{Mkm} \geq \sum_{i=1}^{N} (d_{ij} \cdot W_{Aikm}) \quad + \quad (b_{jkm} - 1) \cdot \sum_{i=1}^{N} W_{Aikm} \tag{5.5}$$

Equation (5.1) constrains each array to be assigned to single logical memory. The constraints represented by inequalities (5.3) ensure that each logical memory with one or more arrays assigned to it is assigned to at least one memory resource type, while inequalities (5.2) ensure that each logical memory is assigned to, at most, a single resource type. Inequalities (5.4) ensure that no more resources are used than are available. The constraints represented by (5.5) determine the number of banks of each resource type consumed by each logical memory. The first term on the right hand side (RHS) of (5.5) determines the total number of words assigned to logical memory l_j. If l_j is assigned to configuration m of type t_k then b_{jkm} takes a value of one. As a result the second term on the RHS of (5.5) equates to zero and the number of banks consumed is forced to be sufficient to accommodate all arrays assigned to l_j. If l_j is not assigned to configuration m of t_k, b_{jkm} will take a value of zero and term two on the RHS of (5.5) will be equal to the maximum possible value of term one. As a result the RHS of (5.5) will be less than or equal to zero. Hence the value of $banks_{jkm}$ can be minimised to zero.

A number of different cost functions are employed with this formulation during the search for an optimal pipeline schedule; these are discussed in Section 5.6.

5.3.2 Variable Reductions

To reduce the number of binary variables in the problem, enumeration, as described in [78], is used to eliminate duplicate solutions from the search space. A mapping from array a_i to logical memory l_j is only allowed if $j \leq i$. This enumeration reduces the number of d_{ij} binary variables from N^2 to $\frac{1}{2}(N^2 + N)$, while still allowing all possible groupings of arrays in a single memory.

The number of b_{jkm} variables in the formulation is also reduced by limiting the number of resource types and configurations available to each logical memory. However, this is done in such a way that the optimum solution[2] is never cropped. When the array to logical memory placement options are enumerated, they are done so by order of increasing width (word length) so that every array that may be assigned to l_j has a width less then or equal to that of a_j. Let $width_j$ be the width of array a_j. Logical memory l_j is considered to have a nominal width of $width_j$. When selecting which configurations of each type could potentially be targeted by l_j a number of candidate widths are considered, all greater than or equal to $width_j$, that may allow multiple elements of arrays a_1 to a_{j-1} to be packed into a single memory word. For example, if $width_j$ were 18 bits and array a_{j-1} had a width of 8 bits then we would also consider a width of 24 bits as a candidate as this would allow three elements of a_{j-1} to be packed into a single memory word. The packing of multiple array elements into a single memory word is only considered for arrays that are read but not written during the loop operations. This because extra complications arise when attempting to write a single array element when multiple array elements are stored in each memory word [74].

For each candidate width the configuration from each memory type is selected that would leave the fewest empty bits in each word. Multiple banks of a width less than the candidate width may be concatenated horizontally to produce a memory of sufficient width. Where two configurations offer the same number of empty bits per word, the configuration with the largest width for each memory block is selected. Most embedded FPGA memories may be configured to have a width of one bit, so any prime candidate widths would lead to this configuration being selected. Obviously, if the width is large (17 bits for example), this could lead to clock frequency issues when the design is actually implemented on an FPGA. The user is therefore allowed to set a limit on the number of banks that may be concatenated to produce larger word lengths.

To further reduce the number of b_{jkm} variables in the problem the number of words in array a_j is taken as the nominal size of logical memory l_j. l_j may then only be targeted to

[2]That with minimum cost according to the cost functions presented in Section 5.6.

memory types whose storage capacity across all its instances is sufficient to accommodate it. Without these optimisations one could expect each logical memory to produce over 30 b_{jkm} variables as embedded FPGA memories can have up to 10 configuration options. The reductions provided by these optimisations will vary from application to application, but in many cases they can be significant. For example, applications that use arrays whose widths are multiples of 8 bits (such as floating point matrix or image processing applications) will only consider one configuration for each memory type.

5.3.3 Bindings

The freedom may not exist to place every array in any given resource type. For example, in an application partitioned across a microprocessor and an FPGA, certain arrays may be constrained to be placed in off-chip memories so that they may be accessed by the microprocessor. In such cases we must include extra constraints to bind the arrays to the specified memory resource type(s). These are described by equation (5.6) for the case when array a_i is constrained to use configurations of resource type t_k.

$$\forall j \geq i, \qquad \sum_{m=1}^{con_k} b_{jkm} = d_{ij} \tag{5.6}$$

5.4 Data Reuse

In this work array data is reused through scratch pad memories [9, 68] and shift registers or FIFOs [36, 65]. This can reduce the number of accesses to off-chip memories, which often form the bottleneck in FPGA based systems, and improve pipeline throughput. Cache based reuse options are not considered as their variable latencies do not fit with the pipelining methodology used here.

5.4.1 FIFOs

The inference of shift registers and FIFOs has previously been considered for inner loop pipelining [36, 65], but here the concept is extended for pipelining above the innermost

loop. A FIFO may be inferred for data reuse where a memory read operation is dependent on a previous read or write in the loop, provided the dependence distance vector [140] has a constant length. Let the dependence vector for a nested loop with L levels be $< i_1, ..., i_L >$, with each i_n representing the number of iterations separating the dependent operations at level n in the loop. A FIFO must be created for each level with a non-zero element within the dependence vector. For example, given a loop with four levels of nesting containing the dependence vector $< 3, 2, 0, 1 >$, this specifies that data is reused after three iterations of the outermost loop (level 1), a further two iterations at level 2 and a further iteration at the innermost loop. As such three separate FIFOs must be instantiated. The first FIFO carries the data across the 3 outer loop iterations and so it must have the capacity to hold all the data created by the source operation of the dependence in 3 iterations of the outermost loop. Likewise the second and third FIFOs must have the capacity to store the data generated during 2 iterations at loop level 2 and 1 iteration of the innermost loop respectively. From here on the set of FIFOs required to carry data across a dependence vectors is referred to as a *FIFO chain*. The three FIFOs required in the given example must remain as separate FIFOs when they are implemented and cannot simply be grouped together to form one larger FIFO. This is because it must be possible to fill each FIFO with data independently at the start of iterations at the different loop levels.

Any FIFO inferred for loop level n must be filled with data from memory prior to the start of the first iteration at that level so that the first i_n iterations are supplied correctly. Going back to the previous example with the dependence vector $< 3, 2, 0, 1 >$, the data used by the sink operation of the dependence during the first 3 outer loop iterations must be loaded into the FIFO for the outer loop prior to the start of the loop execution. Further to this, at the start of each outer loop iteration, the data used during the first two iterations of loop level 2 on that outer loop iteration must be loaded into the FIFO for level 2. Likewise, at the start of each iteration at loop level 3, the data used during 1 innermost loop iteration must be loaded into the FIFO for the innermost loop. These fill routines essentially represent extra imperfectly nested loop operations that must be added if the reuse option is selected for implementation.

By exploiting data reuse along a dependence with a FIFO chain, the number of accesses to the memory housing the original array is reduced by 1. This can allow a lower value of cycles per pipeline stage (T) to be achieved as the minimum T is defined by the ratios of accesses to ports across the system memories. It can also potentially allow a smaller number of pipeline stages (S) or a smaller initiation interval to be achieved. This is because accesses that need to be executed serially without the FIFO chain might be executed in parallel with the FIFO chain, possibly reducing the latencies of critical paths through the dependence graph. The cycles added to the schedule by the fill routines for each FIFO chain must be traded against the scheduling gains offered.

5.4.2 Buffers

In this work a 'buffer' is an on-chip scratch pad memory. The buffer is used to store part of an array and some (or all) of the reads to this array are tasked to the buffer instead of the main memory to which the array is assigned. As with the FIFO chains, if a buffer is selected for implementation, the number of accesses to the memory housing the original array will be reduced. This can allow a lower value of T, S and/or II to be achieved. However, extra imperfectly nested operations may need to be added to the loop to read the data from the host memory of the array in question and write it to the buffer before it can be read from. Buffers may be specified that can be written to by the operations of the innermost loop and these may require fewer (or no) extra imperfect operations to initialise them.

Some simple buffers may be inferred from the dependence vectors in the target loop where the array index for data being read is independent of one or more of the loop indices. For example, a buffer may be inferred for the loop in Figure 5.1(a) as the address function for the read is independent of the loop iterator for level 2. For a given outer loop iteration the data used within each iteration of at level 2 in the loop is the same. Hence the data may be written to a buffer at the start of each outer loop iteration and read multiple times. Figure 5.1(b) shows how the buffer would be implemented in C code. The size of the buffer is determined by the product of the loop bounds for all levels nested

```
x = 0;
for (i = 0; i < 1000; i++){
    for (j = 0; j < 500; j++){
        for (k = 0; k < 200; k++){
            x += a[i][k];
        }
    }
}
```

(a)

```
int buffer[200];
x = 0;
for (i = 0; i < 1000; i++){
    for (m = 0; m < 200; m++){
        buffer[m] = a[i][m];
    }
    for (j = 0; j < 500; j++){
        for (k = 0; k < 200; k++){
            x += buffer[k];
        }
    }
}
```

(b)

Figure 5.1: An example loop for which a buffer may automatically be inferred. (a) The original C code. (b) A C code implementation of the buffer.

below level n, where n is the lowest (most deeply nested) level in the loop for which the buffer's address functions are independent of the loop iterator.

Other more complicated buffers, such as those exposed by the tools presented in [9] and [68], may be supplied to the methodology/tool presented here by specifying the size of the buffer, the read operations that are assigned to it in the loop and the number of writes at each level in the loop required to initialise the buffer. If the buffer is also written to by the perfectly nested operations of the loop, a list of the write operations must also be supplied.

5.4.3 Pre-optimisation of the Reuse Options

In the following section the ILP formulation for array to memory placement is extended to also select which data reuse options are implemented. The number of variables in the

ILP increases with the number of data reuse options, so reducing the number of data reuse options sent to the ILP where possible is desirable. Where the data dependences in a loop are such that a read operation could potentially be fed by more than one data reuse option, it may be possible to reduce the number of reuse options, without any risk of pruning the optimal solution, using some simple comparisons. Essentially, any buffer or FIFO chain may be removed as a data reuse option for a given read operation if another FIFO chain or buffer exists that requires fewer memory resources and has the same or fewer fill operations at every level in the loop.

Intra-loop data reuse may be exploited where two operations are linked by a dependence vector which has all zero elements ($< 0, 0, 0 >$ in a loop with three levels of nesting for example). Implementing such a reuse option requires no memory resources, aside from some registers which are abundant on modern FPGAs, and adds no extra fill operations to the loop. As such exploiting intra-loop reuse is considered to have zero cost, and so all intra-reuse options are selected for implementation prior to further optimisation using the ILP.

As a final pre-optimisation step the sharing of FIFOs across two or more FIFO chains is considered. If two dependences in the loop share the same source operation and have common values in their dependence vectors, then one or more of the FIFOs in their resulting FIFO chains may be shared, reducing the resource usage if both are selected for implementation. FIFOs may be shared for all loop levels up to level n, starting from the outermost level and working inwards, if the two dependence vectors have the same values for each element up to and including level n. Any non-equal elements in the dependence vector break the common FIFO chain and lower levels in the loop cannot share FIFOs, even if the dependence vectors have the same values for further elements. For example, the two dependence vectors $< 1, 2, 1, 4 >$ and $< 1, 2, 2, 4 >$ can share the FIFOs for levels 1 and 2 across their resulting FIFO chains, but not levels 3 or 4. Two dependence vectors $< 1, 2, 1, 4 >$ and $< 2, 2, 1, 4 >$ cannot share any FIFOs.

Table 5.1: Summary of the set notation used to describe the memory optimisation ILP formulation.

Set	Member	Meaning
$\mathbf{A}$	a_i	Set of arrays in the target algorithm
$\mathbf{T}$	t_k	Set of available memory resource types
$\mathbf{L}$	l_j	Set of logical memories to which arrays may be allocated
$\mathbf{L_B}$	l_j	Set of logical memories to which buffers may be allocated
$\mathbf{E}$	e_n	Set of possible buffers
$\mathbf{F}$	f_n	Set of possible FIFO chains
$\mathbf{R}$	r_i	Set of perfectly nested read operations
$\mathbf{S}$	s_n	Set of possible shared FIFOs
$\mathbf{Z_i}$	e_n, f_n	Set of possible reuse options that serve read r_i
$\mathbf{F_n}$	f_a	Set of FIFO chains for which shared FIFO s_n may be a component

5.5 Integrating Data Reuse with Array to Memory Placement

The data reuse options will compete with the array to memory placement for the available memory resources, while the set of reuse options selected will alter the number of accesses to each array and affect how the arrays should be assigned to memories to maximise parallel access. Hence the problems of array to memory placement and data reuse selection are not independent. In this section the ILP formulation presented in Section 5.3.1 is updated to integrate the process of selecting the optimum set of data reuse options into the existing array to memory placement formulation. Tables 5.1 and 5.2 recap the notation used in Section 5.3.1 and summarise the notation introduced in the remainder of this section and in Section 5.6.

We are given a set of P buffers, $\mathbf{E} = \{e_1, ..., e_P\}$, and a set of Q FIFO chains, $\mathbf{F} = \{f_1, ..., f_Q\}$. A second set of P logical memories, $\mathbf{L_B} = \{l_{N+1}, ..., l_{N+P}\}$, is created and the buffers are assigned to these logical memories by a set of binary variables, x_{ij}. x_{ij} is one if buffer e_i is assigned to logical memory l_j and zero otherwise. The set of b_{jkm} binary variables described in Section 5.3.1 is extended to assign the logical memories in $\mathbf{L_B}$ to the available resource types. The set of integer variables, $banks_{jkm}$, is also extended to denote the number of banks of each resource type consumed by the new

Table 5.2: Summary of the constants and variables used in the memory optimisation ILP formulation.

Name	Type	Meaning
N	Constant	Number of arrays in the target algorithm
M	Constant	Number of available memory resource types
num_k	Constant	Number of instances of memory resource type t_k
con_k	Constant	Number of width and depth configurations for resource type t_k
W_{Mkm}	Constant	Number of words in each instance of memory type t_k using configuration m
W_{Aikm}	Constant	Number of words in array a_i when targeted to instances of type t_k using configuration m
d_{ij}	Binary variable	1 if array a_i is assigned to logical memory l_j and 0 otherwise
b_{jkm}	Binary variable	1 if logical memory is allocated to instances of resource type t_k using configuration m and 0 otherwise
$banks_{jkm}$	Integer variable	Number of instances of resource type t_k consumed by logical memory l_j (using configuration m)
P	Constant	Number of possible buffers
Q	Constant	Number of possible FIFO chains
x_{ij}	Binary variable	1 if buffer e_i is assigned to logical memory l_j and 0 otherwise
y_{nkm}	Binary variable	1 if FIFO chain f_n is implemented using instances of resource type t_k and configuration m
B_{nkm}	Constant	Number of instances of resource type t_k consumed by FIFO chain f_n (using configuration m)
R	Constant	Number of possible shared FIFOs
u_{nkm}	Binary variable	1 if shared FIFO is implemented using instances of resource type t_k and configuration m
G_{akm}	Constant	Number of instances of resource type t_k consumed by shared FIFO s_n (using configuration m)
Acc_{ij}	Real Variable	Number of accesses to logical memory l_j for array a_i in a single iteration of the innermost loop
Acc_i	Constant	Number of accesses to array a_i in a single iteration of the innermost loop
$ports_k$	Constant	Number of ports on a single instance of memory resource type t_k
Iss_k	Constant	Minimum number of cycles between the start of successive accesses to memory type t_k

logical memories. A further set of binary variables, y_{nkm}, are created to assign the FIFO chains to memory resource types. y_{nkm} is one if FIFO chain f_n is assigned to configuration m of resource type t_k and zero otherwise. The constraints represented by (5.7) to (5.10) must be added to those listed in Section 5.3.1 to ensure a valid memory subsystem is produced. Inequality (5.4) must also be replaced by inequality (5.11) to ensure that the

combined usage of memory resources across the arrays and data reuse options does not exceed those available. In inequality (5.9), $\mathbf{R}$ represents the set of all read operations at the innermost level of nesting in the target loop. $\mathbf{Z_i}$ represents the set of reuse options that serve read operation r_i. In inequality (5.10), W_{Bikm} is a constant representing the number of words in buffer e_i when it is targeted to configuration m of type t_k. The B_{nkm} term in inequality (5.11) is a constant representing the number of banks of type t_k consumed if FIFO chain f_n is targeted to configuration m of that type. Note that if FIFO chain f_n uses any FIFOs that may be shared with other FIFO chains, as described in Section 5.4.3, then the resources used by the shared FIFOs are not included in the B_{nkm} values. These are dealt with later in this section.

$$\forall l_j \in \mathbf{L_B}, \qquad \sum_{k=1}^{M} \sum_{m=1}^{con_k} b_{jkm} \leq 1 \tag{5.7}$$

$$\forall l_j \in \mathbf{L_B}, \qquad P \cdot \sum_{k=1}^{M} \sum_{m=1}^{con_k} b_{jkm} \geq \sum_{i=1}^{P} x_{ij} \tag{5.8}$$

$$\forall r_i \in \mathbf{R}, \sum_{f_n \in \mathbf{Z_i}} \sum_{k=1}^{M} \sum_{m=1}^{con_k} y_{nkm} + \sum_{e_k \in \mathbf{Z_i}} \sum_{j=N+1}^{N+P} x_{kj} \leq 1 \tag{5.9}$$

$$\forall l_j \in \mathbf{L_B}, \forall t_k \in \mathbf{T}, \forall m \in (1 : con_k), \quad banks_{jkm} \cdot W_{Mkm} \geq \sum_{i=1}^{P} (x_{ij} \cdot W_{Bikm}) + (b_{jkm} - 1) \cdot \sum_{i=1}^{P} W_{Bikm} \tag{5.10}$$

$$\forall t_k \in \mathbf{T}, \qquad \sum_{j=1}^{N+P} \sum_{m=1}^{con_k} banks_{jkm} + \sum_{n=1}^{Q} \sum_{m=1}^{con_k} (B_{nkm} \cdot y_{nkm}) \leq num_k \tag{5.11}$$

The constraints represented by inequalities (5.7), (5.8) and (5.10) constrain the buffer logical memories just as inequalities (5.2), (5.3) and (5.5) constrain the array logical memories. Inequality (5.9) ensures that, where there are multiple reuse options available that supply the same read operation, at most one is selected for implementation. They also ensure that each FIFO chain can be assigned to (at most) one memory resource type and that each buffer can be assigned to (at most) one logical memory. As with the array to

memory placement ILP described in Section 5.3, the number of variables in the problem is reduced by enumerating the buffer to logical memory assignment and by cropping the resource types and configurations available for each logical memory or FIFO chain where possible.

To deal with the shared FIFOs described in Section 5.4.3 an extra set of binary variables must be included for each possible shared FIFO. Given the set of R shared FIFOs, $\mathbf{S} = \{s(1), ..., s(R)\}$, the binary variable u_{nkm} is one if shared FIFO s_n is assigned to configuration m of resource type t_k and zero otherwise. The shared FIFO must be implemented in one resource type if any of the FIFO chains to which it belongs are selected for implementation. This condition is enforced by the constraints represented by (5.12) and (5.13). $\mathbf{F_n}$ represents the set of FIFO chains that may use shared FIFO s_n. X_n represents the number of FIFO chains in $\mathbf{F_n}$.

$$\forall s_n \in \mathbf{S}, \qquad \sum_{k=1}^{M} \sum_{m=1}^{con_k} u_{nkm} \leq 1 \tag{5.12}$$

$$\forall s_n \in \mathbf{S}, \qquad X_n \cdot \sum_{k=1}^{M} \sum_{m=1}^{con_k} u_{nkm} \geq \sum_{f_a \in \mathbf{F_n}} \sum_{k=1}^{M} \sum_{m=1}^{con_k} y_{akm} \tag{5.13}$$

To account for the resources used by the shared FIFOs, the constraints represented by (5.11) must be updated to those in (5.14). The G_{akm} term in (5.14) is a constant representing the number of banks of type t_k consumed if shared FIFO s_a is targeted to configuration m of that type.

$$\forall t_k \in \mathbf{T}, \qquad \sum_{j=1}^{N+P} \sum_{m=1}^{con_k} banks_{jkm} \; + \sum_{n=1}^{Q} \sum_{m=1}^{con_k} (B_{nkm} \cdot y_{nkm}) + \sum_{a=1}^{R} \sum_{m=1}^{con_k} (G_{akm} \cdot u_{akm}) \;\leq\; num_k \tag{5.14}$$

A number of different cost functions are employed with this formulation during the search for an optimal pipeline schedule. The cost functions are discussed in Section 5.6, but we note here that several of them are dependent on the number of accesses to each physical memory in each iteration of the innermost loop level. Since the number of accesses

to each array may vary depending on which data reuse options are selected a further set of real variables, Acc_{ij}, are added to the ILP formulation. Acc_{ij} represents the number of accesses to logical memory l_j during a single iteration of the innermost loop due to array a_i, and each Acc_{ij} value is constrained by inequality (5.15). $\mathbf{Z_i}$ in inequality (5.15) represents the set of reuse options that supply any read operation to array a_i, while Acc_i represents the number of accesses to array a_i with no data reuse. If array a_i is assigned to logical memory l_j then d_{ij} will take a value of one. The RHS of inequality (5.15) will then equal the number of accesses to array a_i minus the number of accesses that are removed due to data reuse. If array a_i is not assigned to logical memory l_m the value of d_{ij} will be zero and so the RHS of inequality (5.15) will always be less than or equal to zero. Each Acc_{ij} variable is implicitly bound to be greater than or equal to zero within the ILP. Hence the number of accesses to logical memory l_j can be determined as the sum of the Acc_{ij} variables for all arrays that may be assigned to it.

$\forall a_i \in \mathbf{A}, \forall l_j \in \mathbf{L},$

$$Acc_{ij} \geq d_{ij} \cdot Acc_i - \sum_{f_n \in \mathbf{Z_i}} \sum_{k=1}^{M} \sum_{m=1}^{con_k} y_{nkm} - \sum_{e_n \in \mathbf{Z_i}} \sum_{k=N+1}^{N+P} x_{nk} \quad (5.15)$$

5.6 Cost Functions

Algorithm 2 (in Section 5.2) lists the functions required to optimise the memory subsystem as the scheduling options for outer loop pipelining are explored. The five functions have five different goals:

1. Minimise the cycles per stage, T.
2. Minimise the perfectly nested stages, S, with T fixed.
3. Minimise the imperfectly nested stages at the pipelined level, Z_p, with T and S fixed.
4. Minimise the initiation interval, II, with T, S and Z_p fixed.
5. Minimise the total number of stages executed in a single iteration of the loop at the pipelined level, S_{tot}, with T, S, Z_p and II fixed.

Ideally we would like to derive a linear cost function for each goal to allow the proposed ILP formulation to be used in each case. The opportunities for linearising each goal are explored in this section.

5.6.1 Minimising T

For the minimisation of T the derivation of a linear cost function is relatively simple. This is because the minimum stage length is determined by the number of clock cycles required to execute all accesses to each physical memory in a single iteration of the innermost loop, ignoring all dependence constraints [36]. Creating a (real) variable for T, its minimum value can be determined using the constraints represented by inequality (5.16). Iss_k represents the minimum cycles between successive accesses to a memory bank of type t_k, and $ports_k$ represents the number of ports to each bank of type t_k. X_j is the maximum number of accesses that can be assigned to logical memory l_j. There must be a separate constraint for each logical memory for each type in which it may be implemented as the various types may have different values of Iss_k and $ports_k$. The second term in the RHS of (5.16) essentially voids the constraint if logical memory l_j is not implemented in memory type t_k ($e_{jkl} = 0$) as it will force the RHS of the inequality to be less than or equal to zero.

$\forall l_j \in \mathbf{L}, \forall t_k \in \mathbf{T},$

$$T \cdot ports_k \;\geq\; \sum_{i=1}^{N} Acc_{ij} \cdot Iss_k \;+ \Big(\sum_{m=1}^{con_k} e_{jkl} - 1 \Big) \cdot X_j \cdot Iss_k \quad (5.16)$$

With the T variable bound by equation (5.16), the cost function when minimising the number of cycles per stage is simply the minimisation of T. This formulation also allows us to bound the value of T when other cost functions are used to meet different goals in the scheduling process.

5.6.2 Minimising S and II

Unfortunately it is not simple to generate linear cost functions to minimise the values of S or II as both of these factors depend on how exactly the memory operations may be

scheduled once the memory subsystem has been set. Essentially the lengths (latencies) of the longest paths through the dependence graph must be minimised. If the minimisation of any path length requires multiple array accesses to be scheduled at the same time, the memory must be optimised to provide sufficient ports for the concurrent access. Work has been done on optimising the memory to minimise path lengths in dependence graphs [82], but the methods involved are complicated and of exponential time complexity. A decision was taken not to implement such a scheme due to its complexity relative to the limited gains it is likely to achieve in practical cases. In Chapter 4 it was shown that the value of T has a great effect on the final schedule length, while the values of S and II have only small effects. For this reason we ignore goals 2 & 4 from the earlier list. Instead lower bound values of S and II are used to bound how far the solution found by the scheduling search is from the most optimistic lower bound. The lower bound values of S and II are calculated based on the minimum lengths of the longest paths and longest cyclic paths through the data dependence graph assuming no resource constraints. Polynomial time algorithms exist to solve both of these problems [35, 146] so these values can be found relatively quickly.

5.6.3 Minimising Z_p

As was the case for the values of S and II, the final value of Z_p that may be achieved is dependent on the latencies of critical paths through the dependence graph. Hence it cannot be guaranteed that the search will find the memory subsystem for the minimum number of imperfect stages without complex methods for examining critical paths. However, a simple linear heuristic can be provided to allow a near minimum Z_p to be achieved in most cases. Note that each time we attempt to optimise the memory subsystem for the minimum Z_p (goal 3 in the list) the values of T and S are already fixed for the relevant section of the scheduling search. For reasons that are explained in Chapter 4, imperfect stages must always be added to the schedule in multiples of the number of perfect stages, S. The number of sets of S imperfect stages that must be included can be estimated based on two factors. The first is the length of the critical path through the imperfect operations

in the dependence graph. To this end Z_p is bound by (5.17), where Lat_{imp} is the latency of the longest path through the imperfectly nested operations.

$$Z_p \geq \left\lceil \frac{Lat_{imp}}{S \cdot T} \right\rceil \tag{5.17}$$

The second factor is the number of imperfectly nested accesses to each memory. Each set of S stages has $T \cdot S$ cycles into which operations may be scheduled, but S stages will be executed in parallel. This means that, within each set of S stages, there are only T modulo 'slots' into which the memory accesses may be scheduled so that they do not conflict with any other accesses to the same port. For a memory composed of banks of resource type t_k, the number of accesses, acc_k, that may be made in any set of S stages is defined by equation (5.18). Note that acc_k is a constant value for each resource type.

$$acc_k = \left\lfloor \frac{T}{iss_k} \right\rfloor \cdot ports_k \tag{5.18}$$

Z_p is added to the ILP formulation as an integer variable and its value is constrained by (5.17) and (5.19). Imp_i represents the number of imperfectly nested accesses to array a_i and Y_j represents the maximum number of accesses that may be assigned to logical memory l_j. Z_p+1 is used on the left hand side of (5.19) because the constraints determine the minimum number of sets of stages required to accommodate all of the loop operations. This value includes the one set of perfectly nested stages which are not included in Z_p. The third term on the right hand side is again used to void the constraint in the case when logical memory l_j is not assigned to type t_k.

$\forall l_j \in \mathbf{L}, \forall t_k \in \mathbf{T}$,

$$acc_k \cdot (Z_p + 1) \geq \sum_{i=1}^{N} Acc_{ij} \; + \; \sum_{i=1}^{N} (d_{ij} \cdot Imp_i) \; + \; \Big(\sum_{m=1}^{con_k} e_{jkl} - 1 \Big) \cdot Y_j \tag{5.19}$$

5.6.4 Minimising S_{tot}

The final cost function required in this work, used to minimise the number of stages executed in one iteration of the loop at the pipelined level (S_{tot}), must also be heuristic

based due to the complications of how operations may actually be scheduled. S_{tot} is determined as a weighted sum of the number of sets of stages required for each loop level and the number of sets of stages required at each level to initialise data reuse options.

For each loop level n above the innermost level we create an integer variable, Z_n, representing the number of sets of S imperfect stages included at that level. Each Z_n is bounded according to inequality (5.17), except in this context Lat_{imp} will vary for each loop level and is the length of the critical path through operations up to level n. Each Z_n is also constrained by inequality (5.19), with Z_p replaced by Z_n and Imp_i replaced by Imp_{ni}, the number of accesses to array a_i nested at levels up to and including level n.

For each loop level n above the innermost level we create an integer variable, I_n, representing the number of sets of S stages required to accommodate the reuse initialisation operations for that level. Each I_n is bound by the time taken to write the initialisation data to each FIFO chain and buffer filled at level n. When filling a FIFO chain, only one write may be issued in each set of S stages. This is because there is only one write during the perfectly nested operations and the fill operations must match this to avoid port conflicts when the pipeline is full. As a result the FIFOs filled at level n constrain I_n according to inequalities (5.20) and (5.21). FF_{ni} is the number of writes needed to initialise FIFO chain f_i at level n. If a shared FIFO may be used in FIFO chain f_i at level n, then FF_{ni} takes a value of 0. SS_j represents the number of writes required to fill shared FIFO s_j. $\mathbf{S_n}$ represents the set of shared FIFOs that require filling at loop level n.

$$\forall f_i \in \mathbf{F}, \qquad I_n \ \geq \ \sum_{k=1}^{M} \sum_{m=1}^{con_k} y_{ikm} \cdot FF_{ni} \tag{5.20}$$

$$\forall s_i \in \mathbf{S_n}, \qquad I_n \ \geq \ \sum_{k=1}^{M} \sum_{m=1}^{con_k} u_{ikm} \cdot SS_i \tag{5.21}$$

When filling a buffer, e_i, with R_i accesses in the perfectly nested operations, we may write up to R_i values in each set of S stages during initialisation. As a result the buffers filled at level n constrain I_n according to (5.22), where FB_{ni} is the number of writes required to initialise buffer e_i at level n.

$$\forall e_i \in \mathbf{E}, \qquad I_n \ \geq \ \sum_{j=1}^{P} x_{ij} \cdot \left\lceil \frac{FB_{ni}}{R_i} \right\rceil \tag{5.22}$$

The number of stages required at each level in the loop to initialise the reuse options is also bound by the minimum time taken to read all of the necessary data from the system memories. This factor is modeled by (5.23).

$$\begin{aligned} \forall l_j \in \mathbf{L}, \forall t_k \in \mathbf{T}, \qquad acc_k \cdot I_n \ \geq \ & \sum_{i=1}^{Q} \sum_{k=1}^{M} \sum_{m=1}^{con_k} y_{ikm} \cdot FF_{ni} \\ & + \sum_{s_a \in \mathbf{S_n}} \sum_{k=1}^{M} \sum_{m=1}^{con_k} u_{akm} \cdot SS_a \ + \ \sum_{i=1}^{P} \sum_{j=1}^{P} x_{ij} \cdot \left\lceil \frac{FB_{ni}}{R_i} \right\rceil \end{aligned} \tag{5.23}$$

5.7 Integration with scheduling

With only heuristic cost functions to optimise the memory subsystem for the minimum Z_p and S_{tot}, and with no cost functions for S and II, it cannot be guaranteed that the optimal pipelined solution will always be found. However, the heuristics used will never overestimate the minimum Z_p or S_{tot} that may be achieved. Hence these values can be used to produce lower bounds during the search. Likewise, lower bounds are placed on the values on the values of S and II using polynomial time algorithms [35, 146]. Hence, with some modifications to the search algorithm, detailed in Algorithm 3, it can either produce the optimal pipelined solution (within the restrictions described in Section 5.2) or state how far the solution produced is from a lower bound execution time (in clock cycles).

There are a few points to note in Algorithm 3. Firstly, not only can it not optimise the memory to minimise S or II, but it also cannot bound their values when optimising for Z_p or S_{tot}. As a result, once the memory has been optimised for the minimum Z_p (with S already fixed), it must check that the target loop may still be scheduled within the given S stages. This purpose is served by the 'check_S()' function. If the given S cannot be achieved with the memory subsystem for the minimum Z_p, the memory is re-optimised for the minimum T and scheduling within S stages is attempted for this. If scheduling fails at this point the search calculates the lower bound schedule length for that S and moves

Algorithm 3 : Searching the pipelining solution space for each loop level.

```
1: T = memory_minimise_T();
2: while ( T < bound_T(T) ) do
3:    S = ⌈ lat_s/T ⌉; II_min = ⌈ lat_ii/T ⌉;
4:    while ( S < bound_S(T, S) ) do
5:       Z_p = memory_minimise_Z_p(T);
6:       fail = check_S(T);
7:       if (fail = true) then
8:          memory_minimise_T();
9:          fail = check_S(T);
10:      end if
11:      if (fail = false) then
12:         while ( Z_p < bound_Z_p(T, S, Z_p) ) do
13:            S_tot_min = memory_minimise_S_tot(T, Z_p);
14:            fail = check_Z_p(T, S);
15:            if (fail = true) then
16:               update_lower_bound(T, S, II_min, Z_p, S_tot_min);
17:               // revert to memory for min Z_p or S & recheck
18:               fail = revert_and_check(T, S, Z_p);
19:            end if
20:            if (fail = false) then
21:               II = find_II_min(T, S, Z_p);
22:               while (II < bound_II(T, S, Z_p, II) ) do
23:                  cycles =schedule(T, S, Z_p, II);
24:                  best = min(cycles, best);
25:                  II++;
26:               end while
27:            end if
28:            Z_p++;
29:         end while
30:      else
31:         S_tot_min = memory_minimise_S_tot(T, Z_p);
32:         update_lower_bound(T, S, II_min, Z_p, S_tot_min);
33:      end if
34:      S++;
35:   end while
36:   T++;
37: end while
```

onto the next S. Since the optimisation and bounding of the Z_p value is heuristic based, it cannot guarantee that the given value of Z_p (or S) will be achievable after optimisation for minimum S_{tot} or Z_p. Hence, after memory optimisation for S_{tot}, the 'check_Z_p()' is used to schedule the target loop for the given value of Z_p (and S). If this fails the algorithm finds the new lower bound schedule length and reverts to the memory subsystem optimised for minimum Z_p and rechecks. If this also fails it reverts to the memory for minimum T and rechecks again. If this fails the search skips onto the next Z_p.

5.8 Results

The combined memory optimisation and outer loop pipelining methodology has been applied to seven test loops and the results are presented in Table 5.3. In each case, along with the dependence graph for the test loop, a list of the memory resources were input, plus a list of buffers that may be inferred using other methodologies [9, 68]. The on-chip memory resources were assumed to be those found on the Altera Stratix II EP2S30. A single 16MByte bank of off-chip SRAM was included for each loop, except the hydrodynamics kernel. In this case two 16MByte banks of off-chip SRAM were included as there are six 1000x1000 element floating point matrices to accommodate. The C code for the test loops is provided in Appendices C and D. Note that for the Edge Detection kernel the two innermost loops were fully unrolled manually before the memory optimisation and pipelining approach is applied. For the Median Filter the three innermost loops were fully unrolled manually. This was done because the unrolled versions of the loops better demonstrate the ability of the memory optimisation techniques to allow increased parallelism to be exploited. Automated unrolling techniques are explored in the following chapter.

In each case the loop was pipelined with no automated memory optimisations applied (the 'NONE' option in Table 5.3) as the baseline comparison for the results with automated memory optimisation. In this case the arrays used by each kernel were manually assigned to the off-chip SRAM. Note that the 'NONE' option is equivalent to the results that would be produced by the methods presented in the previous chapter. Hence any improvement in performance over the 'NONE' option when memory optimisation is considered shows the improvement in performance relative to the results of the Chapter 4. For all but two of the loops we include results for two forms of memory optimisation. The 'UNBOUND' option in Table 5.3 gives the results for pipelining using the integrated memory optimisation with no user-supplied bindings for arrays to be placed in specific memories. Also included are results for the case when the arrays are explicitly bound to the off-chip SRAM, which may be more likely in real examples. This is the 'BOUND' option in Table 5.3. There is no 'BOUND' option for the matrix-matrix multiply and hydrodynamics kernels as the arrays in these cases are too large to store in on-chip resources.

Table 5.3: Results for edge detection (ED), motion estimation (ME), matrix-matrix multiply (MMM), hydrodynamics (HD) [142], successive over relaxation (SOR) [143], MINRES (MIN) [144] and median filter (MED) kernels. Pipelining results are presented for no memory optimisation (NONE), memory optimisation with no bindings (UNBOUND) and memory optimisation with user supplied array to memory bindings (BOUND). The reduction in off-chip accesses is relative to the case with no memory optimisations (NONE). The bandwidth utilisation column lists the percentage of the available off-chip memory bandwidth used.

Loop	Memory optimisation	No. FIFOs (used/total)	No. buffers (used/total)	Off-chip accesses reduced by:	Bandwidth Utilisation
ED	NONE	-	-	-	99.99%
ED	UNBOUND	7/8	0/8	90.0%	93.70%
ED	BOUND	7/8	1/8	79.9%	98.13%
ME	NONE	-	-	-	99.98%
ME	UNBOUND	0/28	0/8	50.0%	49.2%
ME	BOUND	0/28	1/8	48.0%	99.98%
MMM	NONE	-	-	-	99.95%
MMM	UNBOUND	0/2	1/5	50.0%	99.99%
HD	NONE	-	-	-	91.66%
HD	UNBOUND	4/5	0/7	27.2%	98.52%
SOR	NONE	-	-	-	99.99%
SOR	UNBOUND	2/2	0/4	50.1%	99.99%
SOR	BOUND	2/2	1/4	50.0%	99.89%
MIN	NONE	-	-	-	99.98%
MIN	UNBOUND	0/0	0/3	50.0%	99.97%
MIN	BOUND	0/0	1/3	49.9%	99.87%
MED	NONE	-	-	-	99.99%
MED	UNBOUND	7/8	0/8	90.0%	93.07%
MED	BOUND	8/8	0/8	79.9%	99.74%

With no array to memory bindings we were able to achieve an average speedup improvement of 4.0x over the 'NONE' option. This is due to an 50% average reduction in the number of off-chip memory accesses through a combination of on-chip array assignment and data reuse. With the array bindings in place the average speedup improvement over the seven loops is reduced to 2.71x. We note that in most cases there is a comparable reduction in off-chip memory accesses between the 'BOUND' and 'UNBOUND' solutions, with the small reduction in speedup due to the time taken to fill the extra data reuse options that had to be inferred. For the edge detection and median filter examples there is a more significant drop in accesses reduction, causing the speedup to be halved.

Let the access ratio for each example be defined as the sum of the accesses to the off-chip memory divided by the sum of the sizes of the arrays assigned to the off-chip

Table 5.4: Results for edge detection (ED), motion estimation (ME), matrix-matrix multiply (MMM), hydrodynamics (HD) [142], successive over relaxation (SOR) [143], MINRES (MIN) [144] and median filter (MED) kernels. Pipelining results are presented for no memory optimisation (NONE), memory optimisation with no bindings (UNBOUND) and memory optimisation with user supplied bindings (BOUND). The speedup is relative to sequential execution. The access ratio is defined as the sum of the accesses to the off-chip memory divided by the sum of the sizes of the arrays assigned to the off-chip memory. The DEV column specifies the deviance of the solution found from the lower bound schedule length determined during the search.

Loop	Memory optimisation	Access ratio	T	Cycles	Dev (cycles)	Speedup
ED	NONE	5.00	10	655,370	-	4.70
ED	UNBOUND	1.00	1	69,905	0	44.06
ED	BOUND	1.02	1	136,225	0	22.61
ME	NONE	25.0	2	3,277,326	-	3.99
ME	UNBOUND	12.3	1	1,638,415	0	7.98
ME	BOUND	13.0	1	1,704,207	0	7.67
MMM	NONE	667	2	2,002,000,030	-	8.49
MMM	UNBOUND	334	1	1,002,039,000	0	16.96
HD	NONE	1.571	6	6,000,294	-	13.67
HD	UNBOUND	1.143	4	4,060,392	8000	20.20
SOR	NONE	1.998	2	2,004,118	-	9.51
SOR	UNBOUND	1.000	1	1,000,039	0	19.03
SOR	BOUND	1.000	1	1,003,059	0	18.98
MIN	NONE	1.996	2	2,002,242	-	8.50
MIN	UNBOUND	1.000	1	1,000,273	0	17.01
MIN	BOUND	1.000	1	1,003,218	0	16.96
MED	NONE	5.00	10	655,370	-	5.30
MED	UNBOUND	1.00	1	70,415	510	49.32
MED	BOUND	1.02	2	136,718	493	25.40

memory. If each array element is read or written (but not both) at least once, as in the seven example loops, the access ratio has a lower bound of one. A value of one implies there are no further possibilities for reusing the data in the off-chip memory. For five of the test loops the methodology presented here achieved accesses ratios that were close to or exactly one, even with the array bindings in place. In each case the data reuse option(s) with minimum fill requirements were selected, so the only way to achieve further acceleration is to partition arrays across memory banks for increased parallel access. The matrix multiply and motion estimation examples have access ratios far in excess of one, so there is still potential for further data reuse. However, in both cases the pipeline stage length (T) has been minimised to one cycle. This means that the memory subsystem is no longer the bottleneck in the system and further speedup can only be achieved through

Table 5.5: Implementation results for the edge detection and matrix multiply kernels. ALUTs are the basic logic elements of the Stratix II device family, while MRAM, M4K and M512 are the three types of embedded memory resource.

Loop / type	ALUTs	Reg	MRAM / M4K / M512	F_{max} (MHz)
ED / NONE	207	254	0 / 0 / 0	410
ED / BOUND	854	811	1 / 1 / 6	363
MMM / NONE	1175	1361	0 / 0 / 0	253
MMM / UNBOUND	1199	1757	0 / 8 / 0	249

unrolling the loop to create extra parallelism. Extending the methodology presented here to include array partitioning and loop unrolling is tackled in the next chapter.

The utilisation of the off-chip memory bandwidth for each benchmark is listed in Table 5.3. In all but three cases the bandwidth utilisation is in excess of 98% indicating that the combined data reuse and pipelining approach has successfully utilised this resource, but in most cases the inclusion of data reuse actually reduced the off-chip bandwidth utilisation. This small reduction is due to the fact that less data is now read from the off-chip memory and more is read from on-chip memories. For the 'UNBOUND' versions of the Edge Detection, Median Filter and Motion Estimation kernels there is a more significant drop in off-chip bandwidth utilisation, but this is due to at least one of the large arrays used by these algorithms being assigned to on-chip memories. In these cases the utilisation of the on-chip memories storing these arrays is over 99%.

The 'DEV' column in Table 5.3 lists how far each optimised solution is from the the lower bound solution derived during the search. Note that the lower bound is estimated using 'best case' results and will never be overestimated. In most cases the scheduling search was able to find the lower bound solution, despite the use of heuristics. For the hydrodynamics and median filter kernels the solutions were only 8000 cycles (0.2%) and 510 cycles (0.7%) respectively from the lower bounds, suggesting that the heuristics used can guide the search effectively.

By introducing data reuse structures into designs one can significantly reduce schedule lengths, but extra resources will be consumed – both in terms of memory resources to store the data and logic resources to control the reuse structures. There may also be some degradation in the clock frequency of the design. Table 5.5 shows the implementation results on a Stratix II EP2S30 device for the edge detection and matrix multiply kernels. The 'BOUND' edge detection example makes use of a relatively large number of reuse structures and so we see a significant increase in the resources used, as well as a 15% drop in clock rate. However, a 4.8x scheduling speedup was achieved over the 'NONE' case, giving an overall acceleration of 4x. The matrix multiply indicates the tradeoff in a case where fewer reuse options are implemented. There is a relatively small increase in resource usage, and only a 2% drop in clock rate compared to a scheduling speedup of 2x over the 'NONE' case. In each case the designer must decide whether the increase in speed is necessary or warrants the extra resource usage.

As in the previous chapters the designs were synthesised, placed and routed using the Design Space Explorer utility of the Altera Quartus II tool (v9.1) [100]. The potential error in the maximum clock frequency is estimated to be roughly 5 to 10%. This means that the solution with no data reuse exploitation could be 10% slower than is actually possible, while the solution with data reuse exploitation could already be at its maximum clock frequency. In the worst case the speedup figures presented could therefore be overestimated by 10%, but this is still much less than the 2x to 4x speedup reported so the data reuse is still worthwhile.

5.9 Summary

In this chapter a combined array to memory assignment and data reuse selection approach has been presented and combined with outer loop pipelining to provide co-optimised schedules and memory subsystems. The results for seven test loops have shown an average speedup of up to 4x over the methods presented in Chapter 4 which do not optimise the memory with the schedule. Despite our reliance on heuristics during parts of the optimisation, the solutions found were on or close to the lower bound schedule lengths that may be achieved, indicating the potential utility of this approach.

Chapter 6

Array Partitioning and Loop Unrolling

6.1 Introduction

Our existing high level synthesis methodology generates a single pipeline to implement a nested loop, along with a co-optimised memory subsystem. We have previously shown that this approach can produce near optimal pipelines (shortest schedule) for the specified target platform, in the absence of automated loop unrolling, but the lack of some form of loop unrolling limits the potential for parallel execution. To address this limitation we modify and extend the optimisation approach of Chapter 5 to include a form of loop unrolling, which is used in most hardware compilers [8, 9, 36].

However, it was also noted in the previous chapter that, in some cases, all opportunities for data reuse had been exhausted, even without unrolling. Unrolling the loop in such a case will provide a scheduler with more operations that can potentially be scheduled in parallel, but without methods to extract data from off-chip memories more quickly, the rate of data input may be insufficient to feed further parallel operations. To this end unrolling must be accompanied by methods to split array data across multiple memory banks (memory ports) where appropriate, so that sufficient parallel access can be achieved to support the parallel execution of unrolled iterations. There is existing work

on the partitioning of arrays [84], but it requires the loop unroll factor to be known prior to partitioning. The problem this raises is that we do not know how many unrolled iterations we can support until we know how the arrays may be partitioned. This leads to a potentially inefficient trial and error based approach.

In this work we extend the methods presented in [84], integrating partitioning, unrolling, pipelining, array to memory allocation and data reuse selection into a single, combined optimisation approach. The goal of this approach is to produce schedules with lengths as close as possible to the minimum schedule length that may be achieved for a the given loop on a given target platform. When applied to eight test loops our methodology leads to an average speedup over sequential execution of 300x, and gets within 10% of an optimistic lower bound schedule length for 5 of the test loops.

6.2 Extensions to Increase Parallelism

In this chapter, in order to increase the level of parallelism produced, we modify and extend the optimisation approach of Chapter 5 to include both array partitioning and a form of loop unrolling. In the three subsections that follow we provide an overview of our extended approach. Section 6.2.1 details the top level function called for each level in the loop. This function searches the pipelining options for the given level, and the unroll options for all levels up to and including the pipelined level. Two further high level functions are called during the execution of the top level function and these are described in Sections 6.2.2 and 6.2.3. The three high level functions described here call in turn a number of lower level functions, the details of which are described in the sections that follow.

6.2.1 Top Level of the Scheduling Search

The top-level function described by Algorithm 4 is executed for each level in the loop to explore the pipelining options at the given level, as well as the unrolling options for each level up to and including the pipelined level. Unrolling is not considered for levels

Algorithm 4 : Top level function to search the pipelining and loop unrolling options for level P in a nested loop. L is the number of levels of nesting in the loop. 'parts' and 'parts_out' are data structures describing how arrays are partitioned. 'unrolls' is an integer array which denotes the proposed level of unroll at each level in the loop. Arguments to a function preceded by '&' are altered by the function.

```
 1: function cycles = pipeline_level(P)
 2:
 3: parts = find_array_partitions();
 4: T = find_T_min(parts);
 5: lower_bound = 0;
 6: best = ∞;
 7: while ( lower_bound < best ) do
 8:    for ( i = P to L ) do
 9:       unrolls[i] = 1;
10:    end for
11:    temp_best = search_level(T, P, parts, &parts_out, &unrolls);
12:    if ( temp_best < best ) then
13:       best = temp_best;
14:    end if;
15:    T = T + 1;
16:    total_its = 1;
17:    for ( i = P to L ) do
18:       unroll = maximise_unroll(T, i, parts);
19:       total_its = total_its*⌈iterations[i]/unroll⌉;
20:    end for
21:    lower_bound = total_its*T
22: end while
23: return best;
```

above the pipelined level as analysis for dependences active above the pipelined level is not included in the current framework. Also, imperfectly nested operations above the pipelined level are assumed to execute on the system's host microprocessor and are not implemented on the FPGA. Hence these levels cannot be unrolled on the FPGA.

The 'pipeline_level' function begins by attempting to partition the target algorithm's arrays so that each access in a single iteration of the innermost loop addresses a different array partition. The partitions are found using the methods proposed in [84], which are discussed in Section 6.5. Once the original arrays have been partitioned, each partition is treated as a separate array and the ILP formulation for memory optimisation described in Chapter 5[1] can be used unchanged to produce a memory subsystem such that the number of cycles per stage in a single pipeline, T, is minimised. With the true mini-

[1]The memory optimisation process both allocates arrays (array partitions in this case) to memories and selects data reuse options for implementation.

mum value of T having been found, the search of the pipelining solution space proceeds for increasing values of T.

The 'search_level' function, described in Section 6.2.2, is called for each value of T in the search range, exploring the pipelining and unrolling options available. In this context 'search_level' returns the best solution found for pipelining at level P, exploring loop unrolling options for all levels up to and including level P. It also modifies an integer array ('unrolls') to indicate the best unroll value found for each level, and an updated set of array partitions ('parts_out'), but these are only of use in the recursive context of the 'search_level' function, which is described in Section 6.2.2.

In the 'pipeline_level' function the best scheduling result returned by 'search_level' is compared to the best result found for any previous values of T, and the lower value kept. The search can then increment T and calculate an optimistic lower bound schedule length for the new T. The lower bound is based around finding the maximum level of unroll that can be achieved for each level in the loop, within the memory restrictions of the target platform and the given value of T. The maximum unroll for each level is calculated independently for each level (*i.e.* assuming that there is no unroll at all other levels, hence the optimistic nature of the lower bound produced) by the 'maximise_unroll' function. This function uses the ILP formulation for memory optimisation described in Section 6.7.4 to find the maximum unroll for each loop level. With the maximum potential for unroll at each level known, we can estimate the minimum number iterations that must execute sequentially at each level. As stated in Chapter 4, the length of a pipelined schedule (in clock cycles) can be estimated by Equation (6.1), where L is the number of level in the loop nest and N_i is the number of iterations at level i in the loop. In the context of loop unrolling we can consider each N_i to be the number of sequential iterations at each level in the loop after unrolling.

$$cycles \approx T \cdot \prod_{i=1}^{L} N_i \tag{6.1}$$

The scheduling options are searched for increasing values of T until the lower bound schedule length for the next T is greater than the length of the best schedule found so far.

Algorithm 5 : Function to search the loop unrolling options for levels Q to L in a nested loop. Q may be any level up to and including the pipelined level. L is the number of levels of nesting in the loop. 'points' is a user specified value determining the number of unroll values that may be searched for each level in the loop. Arguments to a function preceded by '&' are altered by the function.

```
 1: function cycles = search_level(T, Q, parts_in, &parts_out, &unrolls)
 2:
 3: if ( Q > L ) then
 4:    return ∞;
 5: else
 6:    best_1 = search_level(T, Q+1, parts_in, &parts_out, &unrolls);
 7:    if ( Q < L ) then
 8:       parts_out = update_partitions(T, Q, parts_out);
 9:       max_unroll = maximise_unroll(T, Q, unrolls, parts_out);
10:       min_fill = minimise_fill(T, Q, unrolls, parts_out);
11:       best_2 = schedule(T, Q, best_1, max_unroll, unrolls, parts_out);
12:       if ( best_2 < best_1 ) then
13:          unrolls[Q] = max_unroll;
14:          best_1 = best_2;
15:       end if
16:       best_1 = search_unroll(T, Q, best_1, max_unroll, min_fill, &unrolls, parts_out,
                                 points);
17:    end if
18:    for ( i = Q to L ) do
19:       unrolls_temp[i] = 1;
20:    end for
21:    parts_temp = update_partitions(T, Q, parts_in);
22:    max_unroll = maximise_unroll(T, Q, &unrolls_temp, parts_temp);
23:    min_fill = minimise_fill(T, Q, unrolls_temp, parts_temp);
24:    best_2 = schedule(T, Q, best_1, max_unroll, unrolls_temp, parts_temp);
25:    unrolls_temp[Q] = max_unroll;
26:    best_2 = search_unroll(T, Q, best_2, max_unroll, min_fill, &unrolls_temp,
                              parts_temp, points);
27:    if ( best_2 < best_1 ) then
28:       best_1 = best_2;
29:       unrolls = unrolls_temp;
30:       parts_out = parts_temp;
31:    end if;
32:    return best_1;
33: end if
```

6.2.2 Searching the Unroll Options for Multiple Levels

In this section the operation of the 'search_level' function is described and the pseudo code for the function is provided in Algorithm 5. The purpose of 'search_level' is to explore the unroll options for all levels up to and including the pipelined level, returning the unroll factor for each level that allows for the shortest schedule. The function also attempts to split the input array partitions into a larger number of smaller partitions in order to

maximise the potential for unroll at one or more levels. The goal is to further partition the arrays so that groups of iterations at a candidate level for unroll can be guaranteed to access different partitions. The details of the methods used by the 'update_partitions' function, which is called by 'search_level' to update the input partitions for unrolling at the given level, are given in Section 6.5.

Ideally we would like the freedom to search all the combinations of unroll factors for the levels in the loop, but this could lead to a very large solution space. For example, even for a relatively modest loop with only three levels of nesting and only 20 iterations at each level, there would be 8000 possible combinations of unroll factors. As will be shown later in this chapter, significant effort may be required to optimise the memory subsystem and schedule for each combination of unroll factors. This makes a full search impractical and so the search space has to be limited. The limitations imposed in this work are as follows:

1. Only a limited number of unroll options may be explored for each level in the loop. The maximum number of unroll options to be searched is set by the user using the 'points' variable in Algorithm 5. Details of how the candidate unroll options are selected are given in Section 6.2.3.

2. The unroll options are searched for the innermost level first and the outermost level last. When unrolling at any level, Q, the best set of unroll factors for levels $Q+1$ to L have already been found[2]. When unrolling at level Q the existing unroll scheme for levels $Q+1$ to L may either be kept, placing additional constraints on the array partitioning scheme and potentially limiting the unroll at level Q, or the existing unrolling scheme may be discarded entirely in favour of maximising the unroll at level Q.

The 'search_level' function uses recursion to search the unrolling options at each of the levels in turn. The function is first called to search the unrolling options at level P (the pipelined level), but before the unroll options are searched for this level another instance

[2]The unroll factor for level $Q+1$ is selected assuming no unroll at level Q.

of the function is called to search the unroll options at level $P+1$. The recursion continues until the function is called to search the unroll options at level $L+1$ – the non-existent level nested below the innermost loop. In this case 'search_level' returns instantly and the search of the unroll options progresses for the loop levels in order, with the innermost level searched first.

The instance of 'search_level' called for level Q will first call for anther instance of 'search_level' to search all levels nested below Q in the loop. When this instance returns it provides the length of the shortest schedule found so far (the 'best' variable), the unroll factors for each level used to obtain the best schedule (the 'unrolls' array) and the array partitioning scheme required to achieve the best schedule ('parts_out'). The unroll options for level Q are then searched twice; level Q is first unrolled under the constraints that the unroll factors and partitions for levels $Q+1$ to L are kept, and then with the unroll factors for the levels below reset to 1 and the original partitions derived in the 'pipeline_level' function restored. In both cases the array partitions are updated to maximise the potential for unrolling at level Q. The maximum level of unroll at level Q is found, as well as the minimum time taken to fill any data reuse options that are necessary to meet the target value of T. These two values are used to bound the search for the best unroll factor for level Q.

A user defined variable, 'points', determines how many unroll values for the current level are searched. These values are investigated by the 'search_unroll' function, which is described in Section 6.2.3. Before any other unroll values are searched, the scheduling options for the maximum unroll factor are investigated. The 'maximise_unroll' function returns the maximum unroll that may be achieved for the target loop level, with the unroll values at all other levels constrained to fixed values. This is the same function that is used in Algorithm 4 to find the maximum unroll at each level, but in Algorithm 4 the unroll factors for all other levels are fixed to be one, while here they may be greater than one. The 'minimise_fill' function performs a similar task as 'maximise_unrolls', but its goal is the minimisation of the pipeline stages that must be included to initialise/fill any data reuse options that must be included to meet the target unroll factors specified for each

loop level. The unroll factor for the target level is set to one during 'minimise_fill' so that the minimum fill for all values of unroll at that level can be found. The details of the ILP formulation for memory optimisation that underpins both these functions are given in Section 6.7. The 'schedule' function performs a detailed scheduling search for each set of candidate unroll values, potentially returning a valid schedule and corresponding memory sub-system if one can be found that improves on the execution time of the best schedule found so far. The details of this function are provided in Section 6.3. At the end of the 'search_level' function the solutions produced with and without unrolling at the levels below level Q are compared against the shortest schedule found so far. The schedule length, unroll factors and array partitions are then returned.

6.2.3 Searching the Unroll Options for a Single Level

In this section the operation of the 'search_unroll' function is described and the pseudo code for the function is provided in Algorithm 6. The purpose of 'search_unroll' is to explore the unroll options for a single given level and return the best solution found. As was mentioned in the previous section, the unrolling solution space for most real loops may be too large to search exhaustively. As a result, using the 'points' variable, the user specifies the number of unroll values at each level for which the scheduling options are searched. In the worst case the function places these points evenly throughout the unroll range between the 'max_unroll' value and 1 (no unroll)[3]. However, using knowledge of the lower bound fill time that can be achieved for any value of unroll (the 'min_fill' input variable), it may be possible to iteratively prune lower values of unroll from the search space and decrease the intervals between unroll values.

Essentially the fill time for the reuse options in a pipeline would be expected to increase with the degree of unroll, but the relationship will often be non-linear and difficult to predict. It is possible that there may be no increase in fill time as unroll increases, but is should never decrease as unroll increases, as an increase in the degree of loop unroll should never lead to fewer reuse structures being required. Hence, if the fill time for an

[3]Note that scheduling for the 'max_unroll' option has already been considered before entering the 'search_unroll' function.

Algorithm 6 : Function to search the loop unrolling options for level Q in a nested loop. Q may be any level up to and including the pipelined level. L is the number of levels of nesting in the loop. 'points' denotes the number of unroll values that may be searched and 'base' denotes the lowest value of unroll that may be investigated. Arguments to a function preceded by '&' are altered by the function.

```
 1: function cycles = search_unroll(T, Q, best_in, max_unroll, min_fill, &unrolls, parts,
                                    points)
 2:
 3: its_below = 1;
 4: for ( i = Q+1 to L) do
 5:    its_below = its_below*⌈iterations[i]/unrolls[i]⌉
 6: end for
 7: base = 1;
 8: points_loc = points;
 9: best_1 = best_in; local_min_fill = min_fill;
10: while ( base < max_unroll ) do
11:    unroll = max_unroll;
12:    while ( unroll > base) AND ( lower_bound < best_in ) do
13:       unroll = unroll - 1;
14:       lower_bound = ⌈iterations[Q]/unroll⌉*its_below*T + local_min_fill;
15:    end while
16:    if ( lower_bound ≥ best_in ) then
17:       unroll = unroll + 1;
18:    end if
19:    if ( unroll < max_unroll ) then
20:       local_min_fill = minimise_fill_given_unroll(T, Q, unroll, unrolls, parts);
21:       lower_bound = ⌈iterations[Q]/unroll⌉*its_below*T + local_min_fill;
22:       if ( lower_bound < best_in ) then
23:          best_2 = schedule(T, Q, best_1, unroll, unrolls, parts);
24:          if ( best_2 < best_1 ) then
25:             unrolls[Q] = unroll;
26:             best_1 = best_2;
27:          end if
28:       end if
29:       points_loc = points_loc - 1;
30:    end if
31:    base = round((max_unroll - unroll)/points_loc) + unroll;
32: end while
33: return best_1;
```

unroll value of X is Z cycles, the lower bound fill time for an unroll value of $X + 1$ is Z cycles.

Before scheduling is considered for any unroll values, the lower bound execution time for decreasing unroll values between 'max_unroll' and the 'base' value[4] is estimated, assuming that the minimum fill value may be achieved in each case. The lowest value of unroll is found such that the lower bound schedule length is less than the current

[4]The 'base' value is initially set to one and increases as the search progresses.

best schedule length. If this value of unroll is less than the maximum unroll, the actual minimum fill time that can be achieved for this unroll is found and the lower bound schedule length updated. The minimum fill that may be achieved for a given unroll is found using the 'minimise_fill_given_unroll' function. This is another function that makes use of ILP based memory optimisation, the formulation for which is described in Section 6.7. The scheduling options for the candidate unroll value are then explored if the updated lower bound is less than the best solution found so far. The number of points still to be searched is decremented and the 'base' value for the search is increased to the next point in the search above the current unroll value, assuming that the remaining points in the search are spaced evenly between the current unroll value and the maximum unroll. The minimum fill can also be updated to that found for the current unroll, since all subsequent unroll values will be larger than the current unroll and therefore cannot have a minimum fill less than this. The process then repeats until the next candidate point is the maximum unroll value, indicating the end of the search for the given loop level. The details of the 'schedule' function are included in Section 6.3.

6.3 Scheduling For Fixed Unroll Values

In the overall scheduling search described in the previous section, the 'schedule' function is always called with fixed values for the unroll factors at each level in the loop. The number of cycles per pipeline stage, T, is also fixed at this point in the search. The goal of the 'schedule' function is to attempt to find the optimal set of values for the number of perfectly nested pipeline stages, S, the initiation interval, II, the number of imperfectly nested pipeline stages, Z_i, at each level, i, in the loop, and the number of pipeline stages for filling data reuse structures. A module schedulo for the target loop is also found and the memory subsystem is updated during this phase of the scheduling search. The pseudo code for the 'schedule' function is provided in Algorithm 7.

Since the unroll factor for each loop level is fixed at this point, the operation of the 'schedule' function is almost identical to the scheduling search algorithm described in the previous chapter (Algorithm 3 in Section 5.7), which finds attempts to find the optimal

Algorithm 7 : Searching the pipelining solution space for each set of candidate unroll factors.

```
1: S = estimate_S_min();
2: II_min = ⌈ lat_ii/T ⌉;
3: while ( S < bound_S(T, S) ) do
4:    Z_p = memory_minimise_Z_p(T);
5:    fail = check_S(T);
6:    if (fail = true) then
7:       memory_maximise_unroll(T);
8:       fail = check_S(T);
9:    end if
10:   if (fail = false) then
11:      while ( Z_p < bound_Z_p(T, S, Z_p) ) do
12:         memory_minimise_S_tot(T, Z_p);
13:         fail = check_Z_p(T, S);
14:         if (fail = true) then
15:            // revert to memory for min Z_p or S & recheck
16:            fail = revert_and_check(T, S, Z_p);
17:         end if
18:         if (fail = false) then
19:            II = find_II_min(T, S, Z_p);
20:            while (II < bound_II(T, S, Z_p, II) ) do
21:               cycles =schedule(T, S, Z_p, II);
22:               best = min(cycles, best);
23:               II++;
24:            end while
25:         end if
26:         Z_p++;
27:      end while
28:   end if
29:   S++;
30: end while
```

schedule and memory subsystem for a fixed unroll factor of 1 at each level. The Integer Linear Programming formulation for modulo scheduling, which is used by the 'check_S', 'check_Z_p', 'find_II$_{min}$' and 'schedule' functions, must be updated to deal with the inclusion of loop unroll, and the details of this are described in Section 6.4. The ILP formulation used by the memory optimisation functions ('memory_minimise_unroll', 'memory_minimise_Z_p' and 'memory_minimise_S$_{tot}$') is altered significantly from the previous chapter and is described in Section 6.7. The only other notable change from Algorithm 3 to Algorithm 7 is the removal of the outermost loop level to search a range of T values since this loop is included in a higher level function and T is fixed within the 'schedule' function.

6.4 Modulo Scheduling with Loop Unrolling

To increase the parallelism achieved for the target loop by the FPGA coprocessor unrolling is considered at one or more loop levels. Unrolling a loop by a factor of N creates N copies of each operation in the loop. Explicitly unrolling a loop prior to modulo scheduling will therefore increase the number of operations that must be scheduled by a factor of N, increasing the available solution space and potentially increasing the run time of a given scheduling algorithm. In this work ILP has been used to perform the modulo scheduling, and the worst case run time for an Integer Linear Program grows exponentially with the number of variables in the problem [80]. Explicitly unrolling a loop N times prior to scheduling would increase the number of variables in the modulo scheduling ILP by a factor of N, potentially rendering it unsolvable on a desktop computer.

To allow the existing modulo scheduling formulation (with a small extension) to be used with loop unrolling without the number of variables scaling linearly with the unroll factor, scheduling is considered for a single loop iteration without unrolling, as before, and the scheduled datapath duplicated N times for an unroll factor of N. To ensure that any dependences between operations in different iterations at an unrolled loop level are honoured, the start times of consecutive iterations at each loop level may be offset by some value, OS_i (where i is the level of the loop for which the offset is applied). The value of OS_i required for each unrolled loop level can be calculated by the modulo scheduling ILP with the addition of one extra integer variable for each unrolled loop level, used to model the OS_i values, and the inclusion of extra constraints to model additional dependences in the loop that would be automatically honoured if unrolling were not considered. Before these additional constraints are considered we must first revisit and revise the methods used to simplify the dependence graph prior to outer loop pipelining that were proposed in the original Single Dimension Software Pipelining (SSP) work [13]. The restriction that iterations of all loop levels nested both above and below the pipelined loop level must run sequentially allows certain dependences in the graph to be ignored as they will automatically be honoured in the final schedule. Since iterations at any level, i, nested above the pipelined level execute sequentially, any dependence that

crosses iterations at level i will automatically be honoured and can be removed from the dependence graph without affecting the output. Likewise, any dependence that crosses multiple iterations at any level nested below the pipelined level will also be honoured because the iterations at this level also run sequentially. Once these dependences have been removed from the graph, all that remain are dependences with all zero elements in the dependence distance vector (intra-loop dependences) and dependences with non-zero elements in the dependence distance vector at only the pipelined level. Figure 6.1(a) shows an example dependence graph and Figure 6.1(b) shows the simplified version of the same graph assuming that pipelining is to be applied at loop level 2.

In this work the same restrictions on the sequential execution of loop levels above and below the pipelined level are maintained, but are modified to allow unrolled iterations to execute in parallel. For any loop level above or below the pipelined level which is unrolled N times, the executions of the N unrolled iterations may overlap in time, but the groups of N iterations must execute sequentially. In other words, the resulting iterations of the loop produced when unrolling is applied will execute sequentially, but within each of these unrolled iterations there are N copies of the original loop whose executions may overlap in time[5]. If a given loop level (not the pipelined level) is unrolled by a factor of N then any dependence carried across N or more iterations at that level will automatically be satisfied as each group of N iterations executed sequentially. As a result any such dependence may be pruned from the dependence graph prior to pipelining, just as dependences carried over one or more iterations above or below the pipelined level may be pruned from the graph in the original SSP work. Obviously this requires the value of unroll to be known for each loop level prior to any modulo scheduling run, but this is not a problem as the modulo scheduling is applied at the inner levels of the search described in Algorithm 6 where candidate unroll values have been fixed. Let us look again at the example dependence graph in Figure 6.1(a), but assume that pipelining is performed at loop level 2, level 1 is unrolled by a factor of 3 and level 3 is unrolled by a factor of 4. In this case the graph may be simplified as shown in Figure 6.1(c).

[5]The extent to which these iterations overlap is depends on the data dependences present in the loop.

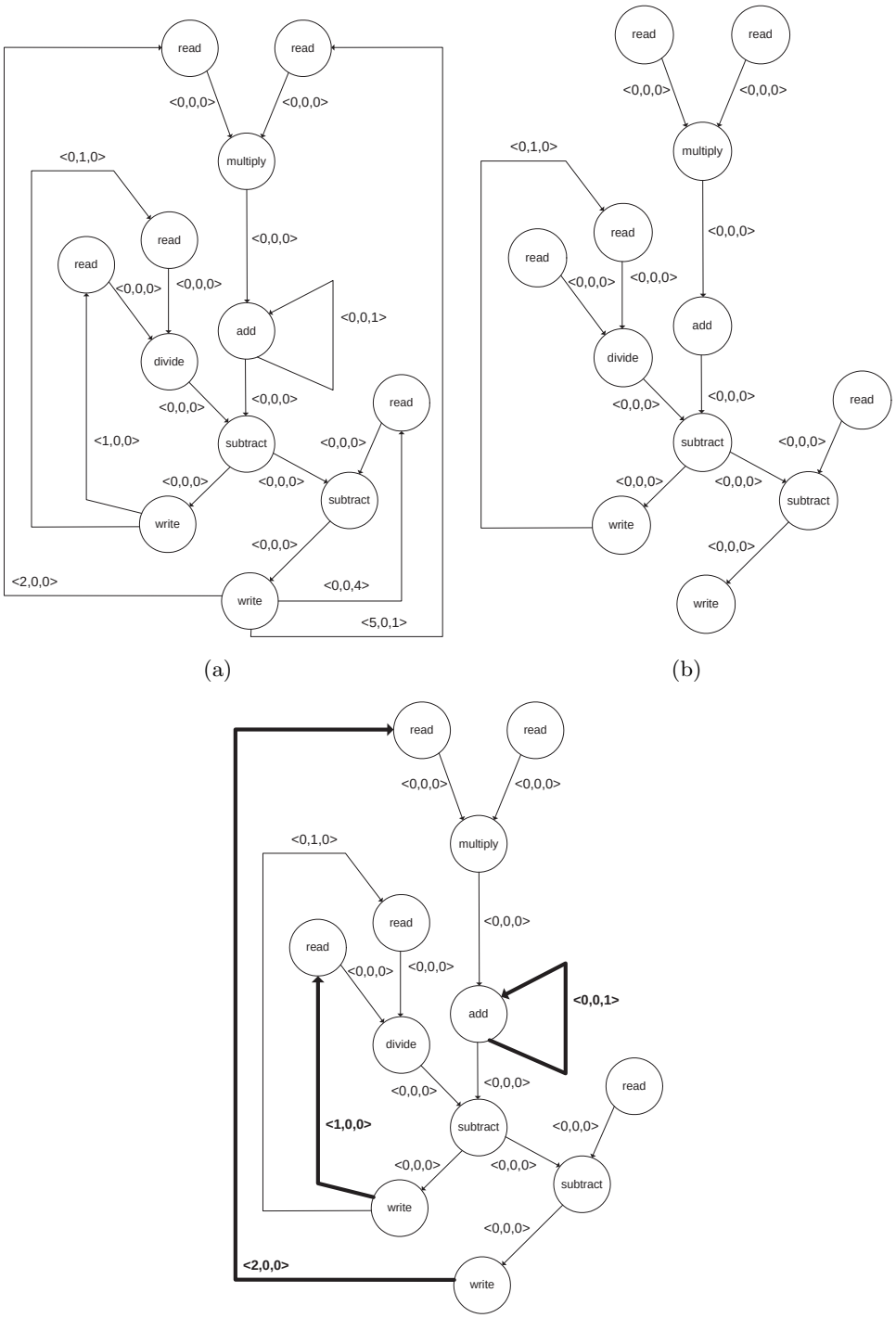


Figure 6.1: Example of dependence graph simplification for outer loop pipelining (a) Original dependence graph. (b) Simplified graph for pipelining at loop level 2 with no unrolling. (c) Simplified graph for pipelining at level 2 with unroll factors of 3 and 4 for levels 1 and 3 respectively.

Each dependence remaining in the graph after simplification results in a constraint in the ILP of the form shown in Inequality (6.2). x_n and x_m are integer variables representing the start times of nodes n and m, while l_m is an integer constant representing the latency of node m. This constraint assumes there is a data dependence between nodes n and m, with m as the source operation and n as the sink. Each OS_i is an integer variable representing the time offset between the start of consecutive unrolled iterations at level i in the loop, while each d_{nmi} is an integer constant representing the number of iterations spanned by the dependence between nodes n and m at level i.

$$x_n + \sum_{i=1}^{L} d_{nmi} \cdot OS_i \geq x_m + l_m \tag{6.2}$$

Of course the simplification of the dependence graph is only valid if steps are taken to ensure that the sequential execution of iterations at loop levels nested above and below the pipelined level is maintained. As specified in Chapter 4 for the case where there is no loop unrolling, the sequential execution of these loop levels can be assured if the end time of the final operation in a given iteration of the innermost loop is less than or equal to the end time of the final pipeline stage for that iteration. This constraint is summarised by Inequality (4.7) in Section 4.6.2 (Chapter 4). However, once unrolling through pipeline duplication has been included, these constraints are no longer sufficient to ensure that the sets of unrolled iterations execute in sequence, as required for the dependence simplification to hold true. Fortunately there is a simple solution to this problem that allows schedules to be generated that are similar to those that would have been produced had the loop be unrolled explicitly prior to scheduling. This solution is explained using a trivial example.

Figure 6.2 is a pictorial representation of a section of an example schedule for a loop with two levels of nesting which has been pipelined at the outermost level and unrolled by a factor of 3 (*i.e.* 3 duplicate pipelines) at the innermost level. The numbers in brackets above and below each iteration in Figure 6.2 represent the iteration number. For example, (0,2) represents inner loop iteration 2 of outer loop iteration 0. It is assumed that the operations in a single iteration of the innermost loop may be scheduled into 4 stages (numbered 0 to 3 in Figure 6.2) and that start time of each pipeline must be offset by

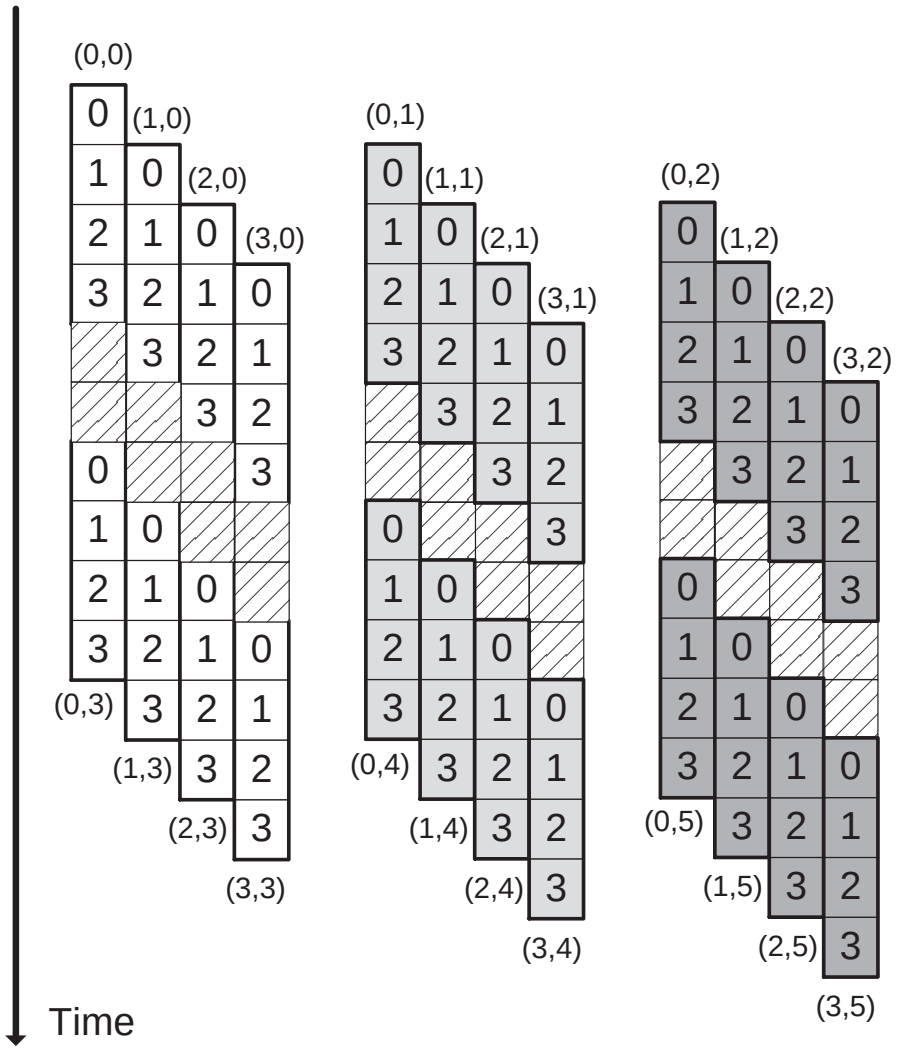


Figure 6.2: Pictorial representation of the schedule for a double nested loop which has been unrolled by a factor of 3 at the innermost level and pipelined at the outermost level. The numbers in brackets above and below each iteration represent the iteration number. Each numbered box represents the execution of a pipeline stage, with the enclosed number denoting the number of the stage.

one pipeline stage relative to the previous pipeline in the chain to meet the dependence constraints[6]. The schedule has been set so the requirement for each set of 3 unrolled iterations to complete before the next set begins is honoured. For example, it can be seen in Figure 6.2 that iteration (0,2) completes its execution before iteration (0,3) begins.

From Figure 6.2 we can see that simply constraining the number of stages in the

[6]These values are chosen arbitrarily for this example.

pipeline to be such that the end time of final stage for each inner loop iteration is greater than or equal to the end time of the iteration's last operation is not sufficient to ensure that each set of 3 unrolled iterations execute sequentially. For example, this constraint alone would allow iteration (0,3) to begin its execution immediately after iteration (0,0) completes, which is too early. Pipeline 0 must hold for the equivalent of 2 pipeline stages after iteration (0,0) has completed before iteration (0,3) may begin. Extra constraints could be included in the scheduling formulation to ensure this constraint is honoured, but this would be an inefficient solution. While Pipeline 0 delays until the time at which iteration (0,3) may begin, up to two stages become idle at once. The same is also true for the other two pipeline copies. Each shaded gray box in the schedule represents a time step where a pipeline stage is idle. Ideally we would like to fill these idle stages with processing from other iterations whose dependences allow them to be execute at these times to maximise the efficiency of the system. Fortunately there is a relatively simple method to accomplish this. Instead of considering the three copies of the pipeline as separate entities, each with 4 stages that allow the executions of 4 iterations to be overlapped, we can think of the pipelines as a single entity which potentially has a larger number of pipeline stages. Figure 6.3 shows how the three pipelines in our example can be merged to form a single larger pipeline. Because Pipelines 1 and 2 are each offset by one stage from the previous pipeline, the merged pipeline has two additional stages, making 6 in total.

With the pipelines merged into a single, larger entity it is now possible to overlap the executions of 6 outer loop iterations, as opposed to only 4 when the pipelines were considered separately. This schedule is shown in Figure 6.4.

The scheduling of the operations shown in Figure 6.2 is unchanged in Figure 6.4; the only difference is that additional operations from outer loop iterations 4 and 5 have been scheduled into the 'gaps' in the schedule. For example, the final stage of iteration (0,2) and the first stage of iteration (0,3) are each highlighted with a '0' and it can be seen that iteration (0,2) still completes its execution before iteration (0,3) begins. The boxes in Figure 6.4 marked with 'X' show executions of stage 0 from the original Pipeline

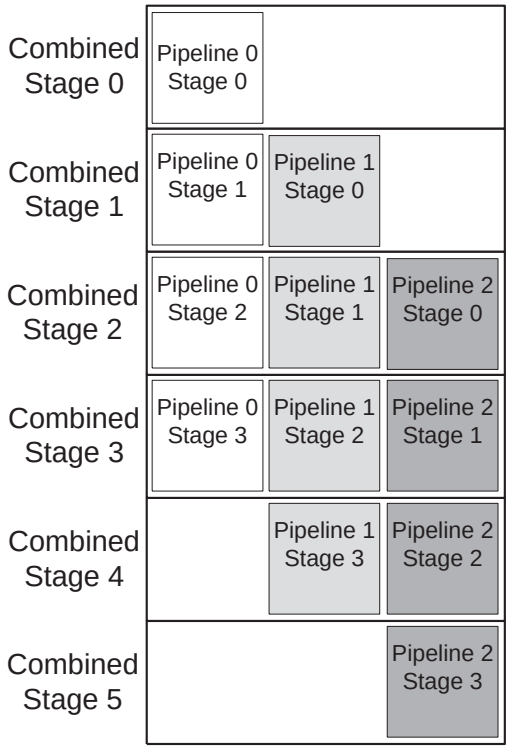


Figure 6.3: Merging 3 unrolled pipeline copies to form a single pipeline with additional stages. Each square box represents a single stage from one of the unrolled pipeline copies.

0 for different iterations. We can see in Figure 6.4 that this stage is now active at every time step, as desired. Furthermore, no additional dependence constraints need be added to the modulo scheduling ILP to achieve these results. The only additional constraints are those required to calculate the number of stages in the merged pipeline, which are represented by Inequality (6.3). A constraint of this form must be added for each node in the dependence graph which could potentially be the operation with the latest end time within each iteration of the innermost loop (*i.e.* any operation that is not the source for a dependence whose sink is an operation in the same iteration). T represents the number of cycles per pipeline stage, S represents the number of perfectly nested stages in the merged pipeline and OS_i and U_i represent the offset and unroll factor for each level i. x_n is the scheduled start time for node n in the dependence graph and l_n is the latency of node n.

$$S \cdot T \geq x_n + l_n + \sum_{i=1}^{P} OS_i \cdot U_i - 1 \tag{6.3}$$

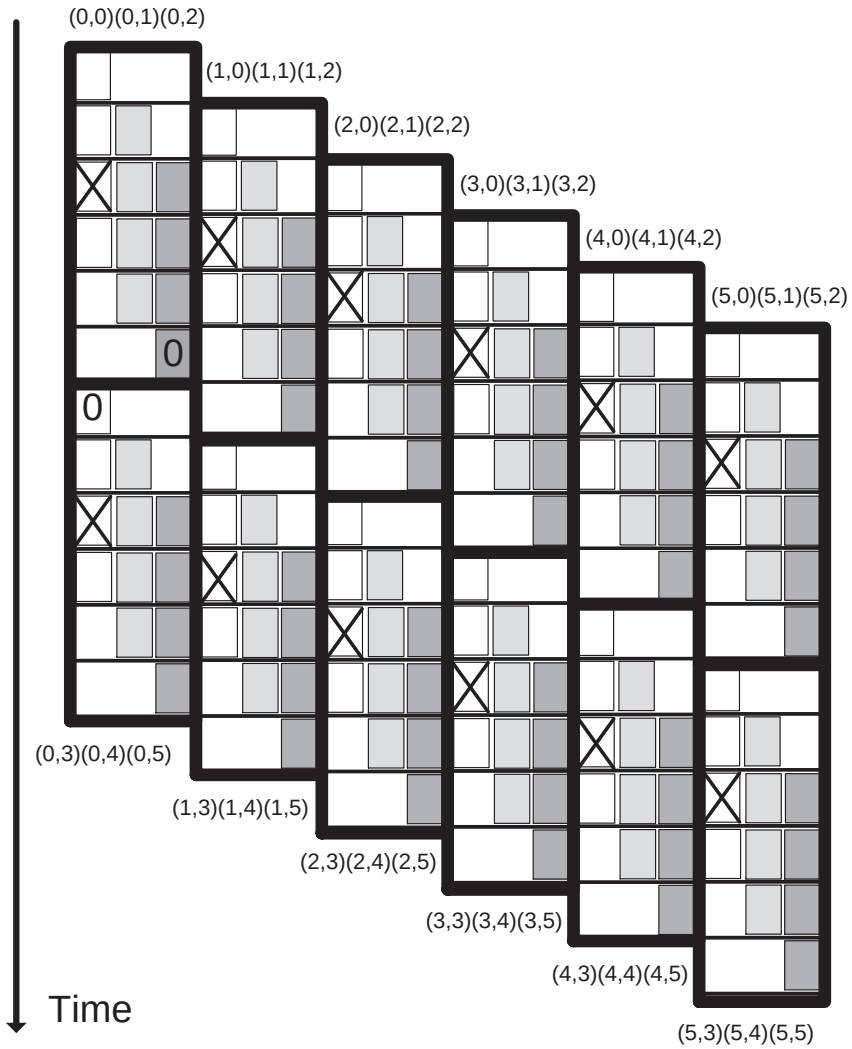


Figure 6.4: Pictorial representation of the overall schedule for the loop in Figure 6.2 using the merged pipeline in Figure 6.3. The numbers in brackets above each iteration represent the number of the iteration being executed.

6.5 Array Partitioning

A number of methods have been presented in previous work that devise schemes for partitioning array data across multiple physical memories so that parallel access can be increased [82–85]. These methods are typically combined with loop unrolling in an attempt to maximise parallel execution. The loop is unrolled by a given degree (potentially at more than one level if the loop is nested), and the address functions for all of the accesses in one iteration of the unrolled loop are analysed.

Of the partitioning schemes proposed in existing literature, perhaps the most general methodology is described in [84]. Relatively simple greatest common divisor (*gcd*) based tests are used to determine whether address functions access mutually exclusive sets of array elements. For multi-dimensional arrays, each dimension is considered separately since two address functions must only be mutually exclusive at a single dimension to ensure they never access the same array elements. The *gcd* based tests only hold for address expressions that are affine functions of the loop index variables, but accesses with non-affine functions for one or more array dimension may still be partitioned if at least one dimension has affine address functions and these can be shown to be mutually exclusive. The *gcd* tests proposed in [84] also only hold if the target loop has been normalised so that each loop level's index variable has unit increment from one iteration to the next.

Let A_n represent access n to a given array, and $F_{nk} = \sum_{m=1}^{M} a_{nkm} \cdot i_m + b_{nk}$ be the address function for dimension k of access A_n. i_m is the loop index variable for level m in a nested loop with M levels. Each a_{nkm} is an integer co-efficient and b_{nk} is the constant integer offset for dimension k of access n. [84] defines the stride of access n at dimension k, s_{nk}, to be $gcd(a_{nk1}, ..., a_{nkM})$[7]. Given two accesses to the same array, A_1 and A_2, they address mutually exclusive sets of array elements if either of the following two conditions are met for any array dimension k:

1. $b_{1k} \neq b_{2k} \qquad AND \qquad s_{1k} = s_{2k} = 0$

2. $b_{1k} \mod gcd(s_{1k}, s_{2k}) \neq b_{2k} \mod gcd(s_{1k}, s_{2k})$

The proof for this condition is provided in [84], along with a full description of the partitioning methodology. Once the array partitions have been created, they must be allocated to physical memories on the target platform. A process to allocate partitions to memories is described in [84], but in this work an ILP formulation is used that also combines data reuse decisions. This is similar to the formulation proposed in the previous chapter and is described in Section 6.7. In this work we apply loop partitioning in two contexts. The first context is to partition arrays in the absence of any loop unrolling such that the number

[7]If F_{nk} has only one non-zero a_{nkm} co-efficient at level L, then s_{nk} is defined to be a_{nkL}. If there are no non-zero co-efficients then s_{nk} is defined to be 0.

of cycles per stage, T, can be minimised. The second context is to partition the arrays so that the degree of loop unroll at a given loop level can be maximised. We address these two uses separately.

6.5.1 Partitioning to Minimise T

As described in Section 6.2.1, array partitioning is considered with no loop unrolling at the start of the pipelining search for each loop level. This process is undertaken by the 'find_array_partitions' function in Algorithm 4 (Section 6.2.1). The function uses the partitioning tests proposed in [84] (and summarised in the previous section) unaltered to determine whether two address functions from the original target algorithm accesses mutually exclusive sets of array elements.

The address functions for all accesses nested at levels up to and including the pipelined level in the loop (without unrolling at any level) are considered one at a time. A new partition is created for the first access to be analysed. Subsequent accesses are then tested for exclusivity against every access in each partition, with three possible outcomes:

1. If the current access is found to be mutually exclusive to every access in each partition then a new partition is created for the current access.

2. If the current access is found to be mutually exclusive to every access in all but one existing partition then the access is added to that partition.

3. If the current access is found not to be mutually exclusive to any access in more than one existing partition, then all these partitions must be merged to form a larger partition and the current access is added to the merged partition.

Once the target algorithm's arrays have been partitioned, each resulting partition can be treated as a separate array throughout the rest of the pipelining search for the given loop level. The array partitions created during this are referred to as *sub-arrays* from here on. The sub-arrays may be further partitioned later in the search to facilitate loop unrolling (as described in the following section). The sub-arrays will also be allocated

to physical memories using the ILP formulations described in Section 5.3 (Chapter 5) and Section 6.7.

As stated earlier, the goal of this phase of array partitioning is to allow array data to be split across physical memories so that the number of cycles per pipeline stage, T, can be minimised. The minimum value of T is determined, post array to memory allocation, by the ratios of accesses[8] to memory ports across all the physical memories. The hope with partitioning is that sub-arrays can be found so accesses from the innermost loop are split across multiple partitions. The partitions can then potentially be placed in different physical memories, reducing the minimum accesses per port that can be expected, and hence the minimum T.

6.5.2 Partitioning to Maximise Loop Unroll

The potential for effective loop unrolling at any level in a nested loop can be dramatically increased if arrays can be partitioned so that the accesses from each of the unrolled iterations address different partitions. Of course unrolling can be applied in the absence of array partitioning but, with the available memory bandwidth acting as the bottleneck in most FPGA based systems, the likelihood is that any significant level of unroll will result in accesses from the unrolled iterations competing for the same memory port(s). Unrolling a loop N times will essentially create N parallel pipelines for implementing the loop, but in the worst case each pipeline could have a stage length N times longer than the pipeline generated without unrolling. Each pipeline will spend most of the time in an idle mode while it waits for its scheduled slot to use the memory port(s). This will result in a resource usage that in N times higher for little or no speedup over the solution found without unrolling.

The array partitioning approach presented in [84] can be used to partition arrays to maximise parallelism in the presence of loop unrolling, but the methodology in [84] requires the loop to be unrolled by a given factor prior to the application of the partitioning tests. In the pipelining search proposed in Section 6.2 there is a need to find the maximum level

[8]The number of accesses in a single iteration of the innermost loop.

of unroll that can be achieved for a given loop level with a constrained, fixed value of T. To find this value using the existing partitioning tests without alteration or extension would require a trial-and-error based search of the unrolling options at the given level. The loop would need to be explicitly unrolled by increasing (or decreasing) factors. For each unroll factor the partitioning tests would be applied, the array data allocated to physical memories and the best data reuse options selected so that every access at the innermost level of the unrolled loop can be made in a window of T clock cycles (ignoring any dependences). If this process fails then the candidate unroll factor is too large and not feasible. Iterating over multiple unroll factors and finding a feasible memory subsystem for each would be a lengthy process, so the *gcd* tests proposed in [84] are extended to allow the maximum unroll for a given level with a given T to be found more directly.

The process for finding the maximum unroll for a given level is broken into two stages. The first stage (the 'update_partitions' function in Algorithm 5) takes a set of input sub-arrays and attempts to further split each sub-array so that, if the loop is unrolled N times, each unrolled iteration will access a different partition. For each input partition, the 'update_partitions' function returns all values of N such that the unrolled iterations access different partitions. The second stage (the 'maximise_unroll' function in Algorithm 5) selects which partitioning scheme (value of N) to use for each input array partition, allocates the array data to physical memories and selects data reuse options to maximise the achievable unroll. The first stage of this process is explained here while the details of the second stage are described in Section 6.7. Given an existing sub-array and a loop level at which to unroll, the partitioning algorithm proceeds according to one of two cases depending on the number of address functions that access the partition.

Single Address Function per Partition

If the sub-array is accessed by only one address function in the loop then simple methods, based on the existing *gcd* based tests, can be used to determine the maximum number of sets of iterations at the unroll level that access mutually exclusive sets of data. The address functions for each array dimension can be considered independently of other dimensions

as two instances of the address function must only be mutually exclusive at one dimension for the sets of data accessed by the full address function to be non-overlapping. Given a loop with M levels of nesting for which level m is being considered for unrolling, the address function, F, for any array dimension will be of the form shown in Equation (6.4). Each a_k is an integer constant, as is b. i_k represents the loop iterator for level k.

$$F = \sum_{k=1}^{M} a_k \cdot i_k + b \tag{6.4}$$

Any, all or none of the a_k values may be 0, but partitioning to unroll at level m can only be considered for dimensions with a non-zero a_m value. If the loop is unrolled by a factor of N at level m and normalised so that there is unit step on the unrolled loop iterator, the N instances of the original address function that result will be of the form shown in Equation (6.5). F_p is the unrolled address function for instance p of the original address function, with p in the range 0 to $N-1$.

$$F_p = \sum_{k=1}^{m-1} (a_k \cdot i_k) + \sum_{k=m+1}^{M} (a_k \cdot i_k) + (N \cdot a_m \cdot i_m) + b + (p \cdot a_m) \tag{6.5}$$

The goal is to determine which values of N produce unrolled address functions that all access non-overlapping sets of array data. Recall that the stride, s, of an address function at a given dimension is defined to be $gcd(a_1, ..., a_M)$. For the each of the unrolled address functions s is defined by Equations (6.6) and (6.7)[9].

$$s' = gcd(a_1, a_2, ..., a_{m-1}, a_{m+1}, ..., a_M) \tag{6.6}$$

$$s = gcd\big(\ N \cdot a_m\ ,\ s'\ \big) \tag{6.7}$$

According to the partitioning tests proposed in [84], the N unrolled address functions will accesses non-overlapping sets of array elements if each address function, F_p, has a unique the *partition number*, as defined by Equation (6.8).

$$partition\ number = (b + p \cdot a_m) \mod s \tag{6.8}$$

[9]In the case where $m = 1$ we define $s' = gcd(a_{m+1}, ..., a_M)$. Likewise, if $m = M$ we define $s' = gcd(a_1, ..., a_{m-1})$. If all a_k values are zero except for a_m then s is defined to be $N \cdot a_m$.

For simplicity we can ignore the b term in (6.8) as it simply adds a constant offset to every *partition number*. It should be clear that, for a set of unrolled address functions with a stride of s, the maximum number of unique *partition numbers* will be s since any value modulo s will return an integer in the range 0 to $s-1$. This makes it relatively simple to determine which values of N will produce mutually exclusive partitions as the are two conditions which are sufficient to ensure this is the case:

1. N must be a factor of s', the value of which is defined by Equation (6.6). Recall that the value of s is defined as $gcd(\ N \cdot a_m\ ,\ s'\ )$. If N is a factor of s' then the value of s, and hence the maximum number of partitions, will be at least as great as N.

2. N must be chosen so that $gcd(s, a_m) = 1$. This condition arises from the fact that the address function for each unrolled iteration must have a unique *partition number*, which is summarised by the set of inequalities represented by (6.9). As shown in Appendix B, this is equivalent to the condition that $gcd(s, a_m) = 1$.

$$\forall k \in [1 : N-1],\ \forall j \in [0 : k-1] \qquad (k \cdot a_m) \mod s \neq (j \cdot a_m) \mod s \quad (6.9)$$

These two conditions may be checked relatively quickly for any candidate N, so all integer N values between 2 and s' may be checked. In the case where all a_k values are zero except for a_m, each iteration at level m will access a set of array elements that does not overlap with any other iteration. Hence, if there are Q iterations at level m, all integer values of N up to and including Q will result in unrolled iterations that access mutually exclusive partitions. In such a case there is no need to apply the *gcd* tests.

Multiple Address Functions per Partition

If the sub-array is accessed by multiple address functions in the loop then further partitioning becomes more complicated. This is because the address functions access overlapping sets of array data[10]. For the input partition to be further divided to allow unrolling, it

[10]If this were not the case then the sub-array would have been further partitioned by the 'find_array_partitions' function in Algorithm 4.

is necessary for each address function to require the data to be divided in the same way. For example, it would not be acceptable for one address function to require two array elements to be in the same partition while another requires them to be in different partitions. Due to this additional complication, partitioning for multiple address functions is only considered in cases where the accesses to the partition are linked by read dependences which have distance vectors with constant, known values for element m (where m is the unroll level being considered). This does reduce the solution space and means that not all possible partitioning options will be found, but also reduces the complexity of finding the partitions compared to the general case.

Assume that the input sub-array has been further partitioned independently for each address function according to the methods described in the previous section, and that a value of unroll, N, has be found that results in N partitions for every address function. With the array data partitioned into N sets, each set should contain the data used by every Nth iteration at the unroll level. As such the partition accessed by a given address function in each iteration of the loop at the unroll level can be determined as i_m mod N, where i_m is the loop iterator for the unroll level. However, given two address functions, the labeling of the partitions (*i.e.* the number assigned by the value of i_m mod N) will not be consistent between the two functions. Both address functions will, for instance, access a partition labeled '1' on iteration 1 at the unrolled level, but they may not necessarily access the same array data. This is not a problem, so long as there is some one-to-one function that maps that partitions for one address function to the partitions of the second address function. This is summarised by condition (6.10). The $\sum_{k=1}^{M} a_{1k} \cdot i_{1k} + b_1$ represents the affine address function for access 1, while $\sum_{k=1}^{M} a_{2k} \cdot i_{2k} + b_2$ represents the address function for access 2. Each i_{1k} value represents the loop iterator for level k for a given instance of address function 1, and likewise i_{2k} for address function 2. $\mathcal{F}$ represents the one-to-one mapping function from the partitions for address function 1 to the partitions for address function 2.

$$\forall i_{1k}, \forall i_{2k} \left(\sum_{k=1}^{M} a_{1k} \cdot i_{1k} + b_1 = \sum_{k=1}^{M} a_{2k} \cdot i_{2k} + b_2 \iff (i_{1m} \mod N) = \mathcal{F}(i_{2m} \mod N) \right) \tag{6.10}$$

Table 6.1: Resulting address functions and partition accesses patterns for the sample loop with an unroll factor of three. 'pipe 0', 'pipe 1' and 'pipe 2' are the three unrolled iterations of the loop. Unrolling the loop three times will require three array partitions (call them p0, p1 and p2) and the final three columns list the partition accessed by each unrolled iteration for each address function.

Original address functions	Unrolled address functions			Partition accessed		
	pipe 0	pipe 1	pipe 2	pipe 0	pipe 1	pipe 2
2i	6i	6i + 2	6i + 4	p0	p1	p2
2i + 2	6i + 2	6i + 4	6i + 6	p1	p2	p0
2i + 4	6i + 4	6i + 6	6i + 8	p2	p0	p1

In the general case there may be no simple way to determine if there is a valid mapping between the two partition sets that meets condition (6.10). However, there will always exist such a mapping function in the case where the array accesses associated with the given address functions are linked by dependences which have distance vectors with known, constant values for element m. If element m of the distance vector is d_m, then at all points where address functions accesses the same array element the condition $i_{2m} = i_{1m} + d_m$ holds true. Taking both sides modulo N we get Equation (6.11), which is equivalent to Equation (6.12), and this defines a one-to-one mapping function from (i_{2m} mod N) to (i_{1m} mod N), as required to meet condition (6.10).

$$i_{2m} \mod N = (i_{1m} + d_m) \mod N \tag{6.11}$$

$$i_{2m} \mod N = ((i_{1m} \mod N) + (d_m \mod N)) \mod N \tag{6.12}$$

Let us consider an example loop in which a sub-array is accessed in one iteration by three address functions – $[2i]$, $[2i + 2]$ and $[2i + 4]$, where i is the loop iterator. Table 6.1 lists the address functions that result from each of the three original functions if the loop is unrolled by a factor of 3, along with the partition accessed by each address function. The three accesses within each unrolled iteration are executed sequentially, while the three unrolled iterations execute in parallel. For each original address function, the unrolled iterations all access different partitions, and the same partition is never accessed by more than one iteration at once.

6.5.3 Partitioning for Unroll at Multiple Loop Levels

So far the methods presented in this work have only considered partitioning in conjunction with unrolling at a single given level. However, as described in Section 6.2, the proposed top level methodology for the scheduling search does allow unrolling and partitioning at multiple levels in the loop. Extending the methodology to fully search the partitioning space for multiple loop levels would be compute intensive as the number of partitioning options grows rapidly as the loop level increases. At the innermost loop the sub-arrays would be partitioned according to the methods presented so far. At the next level in the loop the partitioning options would need to be investigated separately for each of partitioning schemes uncovered for the innermost level. This process will continue for each level in the loop, with the number of partitioning options growing exponentially as the level increases. However, restrictions have already been placed on the search of the unroll space, as described in Section 6.2, and these restrictions help reduce the partitioning options that need to be considered.

Recall that the unrolling options are searched for one loop level at a time, starting with the innermost loop and working outwards. When unrolling at a given level, m, only two options are considered for unrolling at all levels below m; either the set of unroll factors returned for these levels by the unroll search for level $m+1$ are retained, or they are all reset to 1. This restriction on the unroll solution space means that, within each of the two branches of the search, the unroll factors for all levels below m in the loop are fixed, known values. With the addition of one further restriction to the methodology, the partitioning methods proposed for unrolling at a single loop level can be applied directly, without modification, to the case with unroll at multiple loop levels. This restriction need only be enforced on the branch of the unroll search where it is assumed that the best unroll factors for all lower levels must be kept, and can be explained as follows. When unrolling is considered for level $m-1$, the best (shortest) schedule including unroll at all levels up to and including $m+1$ is returned, and the unroll factors used to obtain this best schedule will be retained along one branch of the unroll search for level m.

The memory subsystem required to achieve this best schedule is also returned by the unroll search for level $m-1$ and is passed as an input to the search for level m. A single partitioning scheme has been selected for each array from the available set to achieve this memory subsystem, and this work places the restriction that this partitioning scheme must be kept during the unroll search for level m is the unroll factors for the levels below are kept. For example, assume that unrolling has been considered for the innermost level of a loop and the best schedule has been found for an unroll value of 4. If some sub-array A was split into 4 partitions to achieve this unroll then these partitions must be maintained when unrolling at level 2 in the loop if the unroll factor of 4 is maintained for the innermost loop. The existing partitions may be ignored on the other branch of the search where the unroll factor for the innermost level is set to 1. This additional restriction does, once again, reduce the solution space considered and may cause the optimal solution to be lost, but it is necessary to reduce the time complexity of the partitioning methodology.

The two restrictions described mean that the input to the partitioning process for any loop level is always a set of fixed array partitions (be they the original sub-arrays found by the 'find_array_partitions' function, or further partitions of these sub-arrays) with each being accessed by a set of fixed, known address functions. If partitioning and unroll have previously been considered for a lower loop level there may be a larger number of smaller partitions, but these may still be tested for partitioning for the current level in the same way. Consider a loop which has been unrolled for all levels below m, with some sub-array, A, split into N_k further partitions at each level k. The total number of existing partitions when unrolling at level m (keeping the existing unroll values for lower levels) is $\prod_{k=1}^{m-1} N_k$. These partitions are sets of non-overlapping array elements that service non-overlapping sets of iterations at each loop level. Hence, each may be split into N_m smaller partitions when unrolling at level m if it can be shown that they service non-overlapping sets of iterations at level m.

It may seem like increasing effort will be required to find the possible partitions as the unroll level increases since the partitioning tests should be applied to every input partition, and the number of input partitions could grow rapidly with the loop level.

However this is not the case. Let us consider an example loop which has been unrolled N times at the innermost level, with sub-array A partitioned into N sets. In the original loop without unrolling sub-array A is addressed at one dimension by function F, as defined by Equation (6.13). In (6.13) i_1 and i_2 represent the loop iterators for levels 1 and 2 respectively. After unrolling N times at the innermost level, each partition x of sub-array A^{11} will be accessed by the function F_x, described in Equation (6.14).

$$F = a_1 \cdot i_1 + a_2 \cdot i_2 + b \tag{6.13}$$

$$F_x = N \cdot a_1 \cdot i_1 + a_2 \cdot i_2 + x \cdot a_1 + b \tag{6.14}$$

Within each partition the value of $x \cdot a_1 + b$ is a constant. As previously stated in Section 6.5.2, any constant in an address function may be ignored during the *gcd* based tests for partitioning with unroll as it merely offsets all of the modulo values produced in a uniform manner. The only values that affect whether partitioning is possible for unroll at level 2 are $N \cdot a_1$ and a_2, and these are constant across all of the partitions of sub-array A. Hence the partitioning tests need only be applied once for each sub-array, no matter how many times the array has been partitioned for unroll at lower levels.

6.6 Data Reuse Through FIFOs

As in Chapter 5, FIFOs may be inferred to reuse data where a memory read operation is the sink for a dependence with a constant distance vector. The addition of loop unrolling to the methodology has implications for how FIFOs may be inferred and so the rules for the creation of FIFOs presented here differ from those presented in the previous chapter. As previously stated, in pipelined hardware implementations unrolling a loop is akin to creating a pipeline for the original loop and instantiating multiple copies – one for each unrolled iteration. Any FIFO that is inferred from a dependence which carries data across iterations at an unrolled loop level (*i.e.* has a non-zero value in the dependence distance vector at an unroll level) will pass data from one pipeline copy to another. We refer to

[11] where each x is an integer in the range 0 to $N - 1$.

this type of FIFO as an *External FIFO* from here on. Conversely, any FIFO inferred from a dependence with a distance vector that has zeros for all unroll levels will carry data within a single pipeline copy. We refer to these FIFOs as *Internal FIFOs* from here on.

The Internal FIFOs will have the same properties as the FIFOs presented in Chapter 5, except that multiple copies of each FIFO will be required if the loop is unrolled as each unrolled iteration will require its own FIFO. Details of the selection process for deciding which Internal FIFOs to include in the final design are described in Section 6.7. The remainder of this section focuses on the inference of External FIFOs as these have different properties to those presented in the previous chapter.

The inclusion of External FIFOs in a design can allow the maximum value of unroll that may be achieved for a given number of clock cycles per pipeline stage (T) to be increased. The maximum level of unroll for a given T is determined by the number of copies of each access[12] that may be completed with a window of T clock cycles. 'Carrying' data from a loop iteration in one pipeline copy to a loop iteration in another pipeline copy using a FIFO allows an extra copy of the access to be executed within the T cycles, on top of those which can be executed through ports to the host memory of the array in question. For example, consider a dependence between some memory write operation, w_a, and some memory read operation, r_b, with a distance vector $< 1,\ 1,\ 1 >$ in a loop with 3 levels of nesting and with N iterations at the innermost level. Let us assume that we are attempting to schedule for a T value of 1 clock cycle per pipeline stage and that there is only a single memory port available through which any copies of the r_b access may be executed. In such as case, if we unroll the loop by some factor U[13] then only one of the unrolled iterations/pipeline copies could be completed in any T cycle window through the memory port. However, if $U - 1$ copies of the External FIFO are inferred then all U unrolled iterations can execute in parallel, with the first iteration in each group of U iterations fed by the memory port and the other $U - 1$ iterations fed by the FIFOs.

Where a FIFO is inferred for a dependence with non-zero distance vector values at levels other than the unroll level, it must be filled with data prior to the first iteration

[12]Each access from a single iteration of the innermost loop before unrolling.
[13]With $U \leq N$.

at each of these levels. Returning to the previous example dependence of $< 1,\ 1,\ 1 >$ with unrolling at the innermost loop, in this case data is carried over one iteration at both the outer and middle loop levels so a full outer loop iteration and a further middle loop iteration must be executed before the reused data in the FIFO is valid. To cope with this the data required by these iterations must be read from memory and written to each of the copies of the FIFO. As with the FIFOs in Chapter 5 these fill operations add extra imperfectly nested operations to the loop if the FIFO is selected for implementation, and these must be traded against reductions in the schedule length achieved through unrolling.

The option also exists to supply all U unrolled iterations with data through FIFOs, with the first iteration in each set of U unrolled iterations fed by a FIFO carrying data from the final iteration of the previous set instead of a memory port. In this case the memory port may be used to supply another access in the innermost loop, potentially allowing a lower value of T to be achieved than would be possible if this access were assigned to a memory port. However, this additional FIFO will require extra operations to initialise it with data, on top of those required by the other $U - 1$ FIFOs. The data consumed in the first iteration at the unroll level is not reused from any previous iteration, and so this data must be initialised into the FIFO at the start of each new iteration at the loop level above the unroll level.

In this work a dependence may only be used to infer a FIFO for unroll at the first loop level for which there is a non-zero entry in the dependence distance vector. For example, a dependence with a vector of $< 1, 0, 1, 0 >$ may be used to infer a FIFO for unrolling at one level above the innermost loop. A dependence with a vector of $< 1, 1, 1, 1 >$ may be used to infer a FIFO for unrolling at the innermost loop level. A further restriction is placed on the inference of FIFOs where more than one unrolled iteration is supplied with data directly from a memory port. Let U_{mem} be the number of unrolled iterations at a given loop level for which copies of access acc_i are supplied by memory ports. The number of FIFOs inferred to service other copies of acc_i is restricted to be an integer multiple of U_{mem}. This restriction is imposed to simplify the fine grain scheduling of loop iterations. With U_{mem} pipeline copies supplied by memory ports and with no FIFOs

inferred to provide additional parallel access, there would only U_{mem} parallel pipeline copies in the system, each of which would execute N/U_{mem} iterations[14] (where N is the total iterations at the unroll level). Since each of the pipeline copies executes the same number of iterations they all follow a common schedule, offset from each other by fixed numbers of clock cycles[15]. The inclusion of FIFOs may allow the number of pipeline copies to be increased above U_{mem}, but the array from which acc_i is accessed is still divided into the same U_{mem} sections. To maintain the same schedule simplification once FIFOs are included, each of the U_{mem} groups of N/U_{mem} iterations accessed from each port must be divided across the same number of pipeline copies. For this to hold true the number of FIFOs inferred for access acc_i must be a multiple of U_{mem}.

6.7 Memory Optimisation

As in Chapter 5, an ILP formulation is used to iteratively update the memory subsystem during the scheduling process, allocating array data to physical memories and selecting data reuse options for implementation to optimise a number of cost functions. The formulation is similar to that presented in Section 5.3, but array partitioning, array duplication and the option of multiple copies of each reuse option are included to allow for loop unrolling. The inclusion of these options increases the search space and therefore the number of variables in the ILP. To counter this increase in the number of variables some heuristic decisions must be made prior to writing the ILP input to reduce the search space and the number of variables required. This section describes both the ILP formulation and the heuristic decisions used to reduce the search space.

6.7.1 Array to Memory Placement

The ILP formulation presented in the previous chapter allows data from two or more arrays to be allocated to the same bank of on-chip memory. However, it could be argued

[14]Where U_{mem} is not a factor of N, each pipeline will execute at least $\lfloor N/U_{mem} \rfloor$ iterations, with N mod U_{mem} pipelines executing one additional iteration each.

[15]Each pipeline copy would be scheduled to implement $\lfloor N/U_{mem} \rfloor + 1$, with the final iteration not enabled in some of the pipeline copies.

that the relatively small size and large number of on-chip memory blocks relative to the size and number of the arrays in most real algorithms means that this facility is rarely required. For the eight benchmarks tested in the previous chapter the arrays are only grouped together in the larger, less common off-chip memories. The same results could be achieved using a simpler approach which allows arrays to be grouped together in off-chip memories, but which allows only a single array to be targeted to each on-chip memory.

The inclusion of array partitioning in the formulation means that there is potentially a larger number of smaller data structures to allocate to the memories. In cases where the sizes of the sub-array partitions are less than the sizes of the on-chip memory blocks and the number of partitions is greater than the number of on-chip blocks, there may be a need to allocate multiple partitions to a single on-chip memory bank. However, in most cases one could probably place multiple partitions from the same sub-array in the same bank, achieving the desired result without having to group data from multiple sub-arrays in the same on-chip memory. The exception to this would be when the combined size of multiple partitions from different sub-arrays is a better match to the on-chip block size than any combination of partitions from the same sub-array. In such cases the ability to place data from more than one array in a single on-chip memory block may be required to achieve the optimal solution. However, due to the increase in variables in the ILP required to allow array partitioning, there is a need to reduce the search space in an effort to control run times. As such the ILP is formulated so that multiple partitions from different arrays may be allocated to the same off-chip memory block, but only partitions from the same sub-array may be allocated to a single on-chip memory block. This restriction may exclude the optimal solution in some cases, but it is hoped that these cases are infrequent enough to justify the simplification of the ILP formulation. The notation used in the description of the ILP is summarised in Table 6.2.

The ILP must select a partitioning scheme for each sub-array in the target algorithm and allocate each array partition to a physical memory on the target platform. Since data from different sub-arrays may only be grouped together in off-chip memories, the on-chip and off-chip memory resources are treated differently. When targeting off-chip memories

Table 6.2: Summary of the notation used in the partitioning and array to memory allocation ILP formulation.

Name	Type	Meaning
$\mathbf{A}$ / a_i	Set / Member	Set of arrays in the target algorithm
$\mathbf{G}$ / g_x	Set / Member	Set of available off-chip memory blocks
$\mathbf{T}$ / t_k	Set / Member	Set of available on-chip memory resource types
N	Constant	Number of arrays in $\mathbf{A}$
R	Constant	Number of memory blocks in $\mathbf{G}$
M	Constant	Number of memory resource types in $\mathbf{T}$
num_k	Constant	Number of available instances of resource type t_k
P_i	Constant	Number of possible partitioning schemes for array a_i
z_{ij}	Binary variable	1 if array a_i is partitioned according to scheme j
c_{ik}	Binary variable	1 if array a_i is assigned to on-chip resource type t_k. c_{iM+1} is 1 if array a_i is assigned to an off-chip memory
$blocks_{ik}$	Integer variable	Number of instances of on-chip type t_k consumed by array a_i
$blocks_{ijk}$	Constant	Number of blocks consumed by array a_i if assigned to type t_k and partitioning scheme j is used
d_{ix}	Binary Variable	1 if array a_i is assigned to off-chip memory g_x
$parts_{ij}$	Constant	Number of partitions in array a_i when scheme j is used
$words_i$	Constant	Number of words in array a_i
$words_{ij}$	Constant	Number of words in each partition of array a_i if partitioning scheme j is used
$size_{ik}$	Integer variable	Number of words in off-chip memory g_x used by array a_i
$words_x$	Constant	Number of words in off-chip memory g_x
$accesses_i$	Integer variable	Number of accesses to array a_i after data reuse has been exploited
Acc_i	Constant	Number of accesses to array a_i with no data reuse
$accesses_{ix}$	Integer variable	Number of accesses to off-chip memory due to array a_i
y_{im}	Binary variable	If y_{im} is 1 then I_{im} unrolled iterations access each partition of array a_i
R_i	Integer variable	Number of accesses to array a_i for which data reuse is exploited
$ports_j$	Constant	Number of ports on a single instance of memory resource type t_j
$ports_x$	Constant	Number of ports on off-chip memory block g_x
iss_j	Constant	Minimum number of cycles between initiating concurrent accesses to the same port of an instance of on-chip memory type t_k
iss_x	Constant	Minimum number of cycles between initiating concurrent accesses to the same port of off-chip memory block g_x

an array partition is assigned to a specific off-chip memory block. When targeting on-chip memories an array partition is assigned to a specific type of on-chip memory resource (*e.g.* M4K or M512 on Altera Stratix II devices), but not to any specific instance of that type. The inputs to the system are the set of N sub-arrays, $\mathbf{A} = \{a_1, ..., a_N\}$, the set of

R off-chip memory blocks, $\mathbf{G} = \{g_1, ..., g_R\}$, and the set of M on-chip memory resource types, $\mathbf{T} = \{t_1, ..., t_M\}$. There are num_k instances (banks) of type t_k.

Each sub-array a_i may be partitioned according to one of P_i schemes. For each sub-array a set of P_i binary variables, z_{ij}, is created. z_{ij} is one if sub-array a_i is partitioned according to scheme j and zero otherwise. Each sub-array must be partitioned according to a one of the available partitioning schemes[16] and so the constraints represented by (6.15) are added to the ILP to enforce this requirement.

$$\forall a_i \in \mathbf{A}, \qquad \sum_{j=1}^{P_i} z_{ij} = 1 \tag{6.15}$$

For each sub-array a further set of $M + 1$ binary variables, c_{ik}, are created. For each k in the range 1 to M, c_{ik} is one if the partitions of sub-array a_i are allocated to instances of on-chip resource type t_k and zero otherwise. The c_{iM+1} binary variable is one if the partitions of sub-array a_i are allocated to any off-chip memory block and zero otherwise. Each sub-array must be assigned either to a single on-chip resource type or to off-chip memory. This condition is enforced by the constraints in Equation (6.16).

$$\forall a_i \in \mathbf{A}, \qquad \sum_{k=1}^{M+1} c_{ik} = 1 \tag{6.16}$$

A set of M integer variables, $blocks_{ik}$, are created for each sub-array to determine the number of banks of each on-chip resource type consumed by each sub-array. Each $blocks_{ik}$ variable denotes the number of banks of on-chip resource type t_k consumed by sub-array a_i, and is constrained by (6.17). Each $blocks_{ijk}$ value in (6.17) is a constant representing the number of banks of type t_k consumed by sub-array a_i when it is partitioned according to scheme j.

$$\forall a_i \in \mathbf{A}, \quad \forall t_k \in \mathbf{T}, \quad blocks_{ik} \geq \sum_{j=1}^{P_i} blocks_{ijk} \cdot z_{ij} - X + X \cdot c_{ij} \tag{6.17}$$

The total number of banks of each on-chip resource type consumed by all the sub-arrays

[16] One of the candidate partitioning schemes will always involve leaving the sub-array as a single entity.

must be no greater than those available on the given target device. The constraints represented by (6.18) are used to model this.

$$\forall t_k \in \mathbf{T}, \quad \sum_{i=1}^{N} blocks_{ik} \leq num_k \tag{6.18}$$

A separate set of constraints and variables must be used to map the sub-arrays to particular instances of the off-chip memories. For each sub-array a set of binary variables, d_{is}, is created. d_{is} is one if any one partition of sub-array a_i is mapped to off-chip memory g_s, and zero otherwise. Recall that the binary variable c_{iM+1} takes a value of one if sub-array a_i is assigned to any off-chip memory, and zero otherwise. Hence, for each sub-array a_i, the sum of the d_{is} variables should equal the number of partitions into which a_i is split if c_{iM+1} is one and zero otherwise. These conditions are enforced by constraints (6.19) and (6.20). In these constraints each $parts_{ik}$ is a constant representing the number of sets sub-array a_i is split into when partitioning scheme k is chosen. Z_i is used for simplicity to represent the maximum value of $parts_{ik}$ for each sub-array.

$$\forall a_i \in \mathbf{A}, \quad \sum_{s=1}^{R} d_{is} \geq \sum_{k=1}^{P_i} z_{ik} \cdot parts_{ik} - Z_i + Z_i \cdot c_{iM+1} \tag{6.19}$$

$$\forall a_i \in \mathbf{A}, \quad \sum_{s=1}^{R} d_{is} \leq Z_i \cdot c_{iM+1} \tag{6.20}$$

For each off-chip memory, the total size of all the partitions assigned to the memory must not be greater than the size of the memory. A set of real variables, $size_{is}$, is introduced for each sub-array a_i, with each $size_{is}$ variable denoting the number of words in off-chip memory g_s allocated to a partition of sub-array a_i. The constraints represented by (6.21) and (6.22) are introduced to enforce the size limitation of each memory. Each $words_{ik}$ value is a constant representing the number of words in each partition of sub-array a_i when partitioning scheme k is used. $words_i$ is a constant representing the total number of words in sub-array a_i, and hence the maximum number of words in any partition of a_i. Each $words_s$ is a constant which denotes the number of words in off-chip memory g_s.

$$\forall a_i \in \mathbf{A}, \forall g_s \in \mathbf{G}, \quad size_{is} \geq \sum_{k=1}^{P_i} z_{ik} \cdot words_{ik} - words_i + d_{is} \cdot words_i \tag{6.21}$$

$$\forall g_s \in \mathbf{G}, \quad words_s \geq \sum_{i=1}^{N} size_{is} \tag{6.22}$$

Recall that, within the top level search algorithm proposed in Section 6.2, the first factor for which the memory subsystem must be optimised is the minimisation of the pipeline stage length, T. As stated previously, the simpler ILP formulation for memory optimisation (which does not consider array partitioning) described in Section 5.6 can be used to find the minimum value of T. In this context each sub-array found by the 'find_array_partitions' function in Algorithm 4 (p. 166) is treated as a separate array within the simpler ILP formulation. Once the minimum T has been found, from then on the value of T is a fixed, known value at each point in the search. Hence, within the ILP formulation proposed here, the value of T must be bound to a fixed value and constraints introduced to ensure the bound is enforced.

For each sub-array a_i an integer variable, $accesses_i$, is created to model the number of accesses to a_i in a single iteration of the innermost loop (without unrolling). A further set of binary variables, y_{im}, is also included for each sub-array a_i. This set of binary variables is used to determine the number of unrolled iterations that will access each partition of sub-array a_i in the final system. y_{im} is one if I_{im} iterations access each partition, and zero otherwise. Allowing I_{im} iterations to access the same array partition is akin to creating I_{im} times as many partitions and grouping I_{im} partitions together in a single logical memory. As mentioned earlier, the solution space is restricted so that partitions from multiple sub-arrays can not be assigned to the same on-chip memory. However, it was suggested that allowing multiple partitions from the same sub-array to be grouped together in a single on-chip memory might help reduce the impact this limitation has on the search space.

The number of y_{im} variables required for each sub-array, along with the corresponding I_{im} values, is determined by the candidate value of T for the given point in the

search, and the minimum number of accesses to the sub-array in a single iteration of the innermost loop. Given a maximum number of ports per memory of 2 (which is typical of current FPGAs [4, 141]), a maximum $2 \cdot T$ accesses can be made to each partition of sub-array a_i during the course of a single innermost loop iteration. Let $accesses_{min_i}$ be the minimum number of accesses to sub-array a_i, and $accesses_{max_i}$ be the maximum[17]. As $accesses_i$ varies between these values the number of unrolled iterations that may access each partition will be given by equation (6.23).

$$I_{im} = \lfloor 2 \cdot T / accesses_i \rfloor \tag{6.23}$$

For every unique value of I_{im} found as $accesses_i$ varies in the range $accesses_{min_i}$ to $accesses_{max_i}$, a y_{im} variable is created[18]. Typically one would expect the candidate values of T to be relatively low (normally fewer than 4 clock cycles) so the number of y_{im} variables required for each sub-array should be limited to approximately 8.

The number of accesses to each partition of sub-array a_i during one innermost loop iteration ($accesses_i$) is constrained by Inequalities (6.24). Acc_i is a constant representing the number of accesses to sub-array a_i in a single iteration of the innermost loop in the absence of any data reuse. R_i represents the number of accesses to sub-array a_i for which data reuse has been inferred. The constraints governing the value of R_i are discussed in the following section. Rng_i represents the number of y_{im} variables for each a_i and X_i represents the product of Acc_i and the maximum possible I_{im} for the given a_i.

$$\forall a_i, \quad \forall m \in [1 : Rng_i], \quad accesses_i \geq I_{im} \cdot (Acc_i - R_i) - X_i + X_i \cdot y_{im} \tag{6.24}$$

For the on-chip memories (where only partitions from a single sub-array may be assigned to each bank) the constraints on the value of T are modeled by the inequalities represented by (6.25). X_i again represents the product of Acc_i and the maximum possible I_{im} for the given a_i.

$$\forall t_j \in \mathbf{T}, \forall a_i \in \mathbf{A}, \quad ports_j \cdot T \geq iss_j \cdot accesses_i - X_i + X_i \cdot c_{ij} \tag{6.25}$$

[17] The number of accesses to each sub-array may vary according to the data reuse options selected.

[18] If $accesses_{min_i}$ is zero then a value of 1 is used for the bottom end of the range of $accesses_i$ values to prevent an infinite number of y_{im} variables being created.

The constraints for off-chip memories are more complicated since partitions from multiple sub-arrays may be assigned to the same bank. For each off-chip memory block, g_s, an integer variable, $accesses_{is}$, must be created for each sub-array a_i to model the number of accesses to g_s during the execution of a single innermost loop iteration if a_i is assigned to it. The value of each $accesses_{is}$ is bound by the constraints represented by Inequality (6.26). The constraints on the value of T for the off-chip memories are shown in Inequality (6.27).

$$\forall g_s \in \mathbf{G}, \quad \forall a_i \in \mathbf{A}, \quad accesses_{is} \geq accesses_i - X_i + d_{is} \tag{6.26}$$

$$\forall g_s \in \mathbf{G}, \quad ports_s \cdot T \geq iss_s \cdot \sum_{a_i \in \mathbf{A}} accesses_{is} \tag{6.27}$$

6.7.2 Including Data Reuse Through Buffers

As was the case in Chapter 5 extra variables and constraints are added to the ILP formulation to integrate the selection of buffers to implement with the array to memory placement. Each buffer acts to copy part of an array that may be read more than once in the target loop and provide potentially increased bandwidth through which all the accesses to the array may be made. Just as we allow the main array itself to be partitioned across multiple memory banks, each buffer may also be partitioned. In this work we restrict the buffer to adopt the same partitioning scheme that is selected for the main array from which it reads data. For example, consider the simple example of a matrix-matrix multiplication, the C code for which is shown in Figure 6.5(a). In a single iteration of the outermost loop level the same column of data from matrix A is read by every iteration of the middle loop level. As shown in Figure 6.5(b), a buffer may be inferred to read this column of data at the start of each outer loop iteration and supply the data to each iteration of the middle loop level. If the middle loop level were to be unrolled then each unrolled iteration would still use the same A matrix data so partitioning the A matrix would not increase the potential for parallel access to it. Hence, since no partitioning would be considered for matrix A, no partitioning would be considered for the buffer. Conversely, if the innermost loop level is unrolled then each unrolled iteration will read different A matrix data and partitioning A so that consecutive matrix rows are in different memory banks will increase the potential

```
for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++){
        C[j][i] = 0;
        for (k = 0; k < 100; k++)
            C[j][i] += A[k][i]*B[j][k];
    }
```

(a)

```
for (i = 0; i < 100; i++){
    for (k = 0; k < 100; k++)
        buffer[k] = A[k][i]
    for (j = 0; j < 100; j++){
        C[j][i] = 0;
        for (k = 0; k < 100; k++)
            C[j][i] += buffer[k]*B[j][k];
    }
}
```

(b)

Figure 6.5: Example of buffer inference for a matrix-matrix multiply kernel (a) The C-code for the original kernel. (b) The kernel with a buffer inserted to store a column of matrix data on-chip.

Table 6.3: Summary of the notation used to include buffer selection in the ILP formulation.

Name	Type	Meaning
$\mathbf{B}$ / b_w	Set / Member	Set of available buffers
$\mathbf{ACC_i}$ / acc_{ix}	Set / Member	Set of accesses to array a_i
$\mathbf{B_{ix}}$ / b_w	Set / Member	Set of buffers that can reuse data for array access acc_{ix}
$\mathbf{B_i}$ / b_w	Set / Member	Set of buffers that can reuse data for array a_i
e_{wk}	Binary variable	1 if buffer b_w is implemented in memory blocks of type t_k
$accesses_w$	Constant	Number of accesses to buffer b_w
T	Constant	Number of clock cycles per pipeline stage
$blocks_{wk}$	Integer variable	Number of blocks of resource type t_k consumed by buffer b_w

for parallel access. Partitioning the buffer along the same boundaries will have the same effect and so this option is included in the ILP formulation. A summary of the notation introduced in this section is included in Table 6.3. The notation introduced in Table 6.2 is carried though to this section.

Given a set of possible buffers to implement, $\mathbf{B}$, for each buffer, b_w, in $\mathbf{B}$ a binary variable, e_{wk}, is created. The value of e_{wk} is one if b_w is selected for implementation using resources of on-chip memory type t_k, and zero otherwise. Recall that the set of binary

variables z_{ij} determines which partitioning scheme is adopted by array a_i[19]. If buffer b_w copies part of array a_i then its partitioning scheme is also determined by these z_{ij} variables. Recall also that the set of binary variables y_{im} determines how many unrolled loop iterations will access each partition of array a_i[20], and the same number of unrolled iterations will access each partition of buffer b_w if it copies data from a_i. Each buffer is restricted to be implemented in an on-chip memory resource type. A further set of integer variables, $blocks_{wk}$, denotes the number of blocks of type t_k consumed by buffer b_w. The constraints represented by Inequalities (6.28) to (6.30) are added to the ILP to ensure a valid system is produced. In (6.29), $\mathbf{T}$ is the set of of on-chip memory types and M is the number of types in this set. $blocks_{wkj}$ is a constant representing the number of blocks of memory type t_k that will be consumed by buffer b_w if it is partitioned according to scheme j, and X_{wk} represents the maximum value of $blocks_{wkj}$ across all the available partitioning schemes. In (6.30) $ports_k$ and iss_k represent the number of ports and number of cycles between successive accesses to the same port for memory type t_k. Each I_{im} is a constant representing the number of unrolled iterations assigned to each partition of array a_i, and therefore also each partition of b_w (assuming it copies part of a_i), when y_{im} is one. Rng_i is the number y_{im} variables for array a_i. $accesses_w$ is a constant representing the number of accesses to buffer b_w in a single iteration of the innermost loop and each Z_{wk} is a constant representing the maximum value of the sum term in Inequality (6.30).

$$\forall b_w \in \mathbf{B}, \qquad \sum_{k=1}^{M} e_{wk} \leq 1 \tag{6.28}$$

$$\forall b_w \in \mathbf{B},\ \forall t_k \in \mathbf{T}, \qquad blocks_{wk} \geq \sum_{j=1}^{P_i} blocks_{wkj} \cdot z_{ij} - X_{wk} + X_{wk} \cdot e_{wk} \tag{6.29}$$

$$\forall b_w \in \mathbf{B},\ \forall t_k \in T,$$

$$ports_k \cdot T \geq \sum_{m=1}^{Rng_i} iss_k \cdot accesses_w \cdot I_{im} \cdot y_{im} - Z_{wk} + Z_{wk} \cdot e_{wk} \tag{6.30}$$

[19] z_{ij} is one if partitioning scheme j is adopted by array a_i and zero otherwise.

[20] y_{im} is one if some number of unrolled loop iterations, I_{im}, accesses each partition of array a_i, and zero otherwise.

Inequality (6.18) (from Section 6.7.1) must also be updated as shown in Inequality (6.31) to include the on-chip memory resources consumed by the buffers and ensure the overall resource usage remains within the limits of the target platform. Recall that num_k is the number of blocks of resource type t_k.

$$\forall t_k \in \mathbf{T}, \quad \sum_{i=1}^{N} blocks_{ik} + \sum_{b_w \in \mathbf{B}} blocks_{wk} \leq num_k \tag{6.31}$$

In the Section 6.7.1 we also introduced an integer variable, R_i, for each array, a_i, to track the number of accesses at the innermost loop level to array a_i that are re-assigned to data reuse structures (either buffers or FIFOs). With the introduction of buffers to the system we may begin to constrain these values, as shown in Inequality (6.32). Recall that $\mathbf{A}$ represents the set of arrays in the system and that $accesses_w$ is the number of accesses to buffer b_w in a single iteration of the innermost loop. $\mathbf{B_i}$ represents the set of buffers that provide data reuse for array a_i.

$$\forall a_i \in \mathbf{A}, \qquad R_i \leq \sum_{b_w \in \mathbf{B_i}} \sum_{k=1}^{M} e_{wk} \cdot accesses_w \tag{6.32}$$

One final set of constraints, represented by Inequality (6.33) must also be added to ensure that each access to array a_i is only served by a single buffer, otherwise two buffers could be implemented which serve the same access and this would produce an R_i value of 2 when it should only be 1. In Inequality (6.33), $\mathbf{ACC_i}$ represents the set of accesses to array a_i. acc_{ix} represents access x to array a_i, and $\mathbf{B_{ix}}$ represents the set of buffers that serve access acc_{ix}.

$$\forall a_i \in \mathbf{A}, \forall acc_{ix} \in \mathbf{ACC_i}, \qquad \sum_{b_w \in \mathbf{B_{ix}}} \sum_{k=0}^{M} e_{wk} \leq 1 \tag{6.33}$$

6.7.3 Including Data Reuse Through FIFOs

As stated in Section 6.6, the FIFOs used in this work are split into two categories: Internal FIFOs and Extenal FIFOs. The ILP formulations required to select FIFOs from the two categories differ, with the formulation for External FIFOs requiring extra variables and constraints. The formulations for both cases are presented in the two sub-sections that

follow. The notation from the previous two sections (summarised in Tables 6.2 and 6.3) is carried through to this section and a small amount of additional notation introduced.

Internal FIFOs

Recall that Internal FIFOs may be inferred where a dependence with a memory read operation at its sink does not cross iterations of any unrolled loop level. Therefore, if as a result of unrolling at one or more loop levels there is a total of N unrolled iterations (pipelines), N copies of the Internal FIFO must be included if the FIFO is selected for implementation – one for each pipeline. This requirement for the number of FIFOs to match the number of pipelines is the only significant extension to the ILP formulation for selecting FIFOs presented in Chapter 5. Assume that we are given a set of Internal FIFOs, **FI**. For each FIFO, f_u, in **FI** a binary variable p_{uk} is created. Each p_{uk} takes a value of 1 if FIFO f_u is selected for implementation using on-chip memory resources of type t_k, and 0 otherwise. A set of integer variables, $blocks_{uk}$, is also created for each FIFO. Each $blocks_{uk}$ models the number of blocks of on-chip memory resource type t_k used by the inferred instances of FIFO f_u. Two sets additional sets of constraints are required to ensure a legal selection of Internal FIFOs. These extra constraints are shown in Inequalities (6.34) and (6.35). Recall that **T** represents the set of on-chip memory resource types, of which there are M, and num_k represents the total number of blocks of each type t_k. In (6.35), $blocks_single_{uk}$ is a constant representing the number of blocks of resource type t_k consumed by a single copy of FIFO f_u. N is the total unroll factor across all loop levels (*i.e.* the product of the unroll factors for each level). In this formulation N is a variable and the constraints to force it to model the unroll factor are described in Section 6.7.4.

$$\forall f_u \in \mathbf{FI}, \qquad \sum_{k=0}^{M} p_{uk} \leq 1 \tag{6.34}$$

$$\forall f_u \in \mathbf{FI},\ \forall t_k \in \mathbf{T}, \qquad blocks_{uk} \geq N \cdot blocks_single_{uk} - num_k + p_{uk} \cdot num_k \tag{6.35}$$

With the addition of Internal FIFOs to the formulation, the constraints modeling the overall memory resource usage of the system must again be updated. Inequality (6.31) from Section 6.7.2 must be updated as shown in Inequality (6.36) to take into account any memory blocks used by the Internal FIFOs.

$$\forall t_k \in \mathbf{T}, \quad \sum_{i=1}^{N} blocks_{ik} + \sum_{b_w \in \mathbf{B}} blocks_{wk} + \sum_{f_u \in \mathbf{FI}} blocks_{uk} \leq num_k \tag{6.36}$$

Constraints (6.32) and (6.33) from Section 6.7.2, which constrain the number of accesses to each array that are allocated to data reuse structures and limit each access to be implemented using only one data reuse option, must also be updated to include the Internal FIFOs. The updated constraints are represented by Inequalities (6.37) and (6.38). In these inequalities $\mathbf{FI_i}$ represents the set of Internal FIFOs that serve any access to array a_i. $\mathbf{ACC_i}$ represents the set of accesses to array a_i and $\mathbf{FI_{ix}}$ represents the set of Internal FIFOs that serve each accesses, acc_{ix}, in $\mathbf{ACC_i}$.

$$\forall a_i \in \mathbf{A}, \qquad R_i \leq \sum_{b_w \in \mathbf{B_i}} \sum_{k=1}^{M} e_{wk} \cdot accesses_w + \sum_{f_u \in \mathbf{FI_i}} \sum_{k=1}^{M} p_{uk} \tag{6.37}$$

$$\forall a_i \in \mathbf{A}, \forall acc_{ix} \in \mathbf{ACC_i}, \qquad \sum_{b_w \in \mathbf{B_{ix}}} \sum_{k=0}^{M} e_{wk} + \sum_{f_u \in \mathbf{FI_{ix}}} \sum_{k=1}^{M} p_{uk} \leq 1 \tag{6.38}$$

External FIFOs

Assume that we are given a set of External FIFOs, **FE**. For each FIFO, f_u, in **FE** a binary variable p_{uk} is created. Each p_{uk} takes a value of 1 if FIFO f_u is selected for implementation using on-chip memory resources of type t_k, and 0 otherwise. A set of integer variables, $blocks_{uk}$, is also created for each FIFO. Each $blocks_{uk}$ models the number of blocks of on-chip memory resource type t_k used by the inferred instances of FIFO f_u. Two sets additional sets of constraints are required to ensure a legal selection of External FIFOs. These extra constraints are shown in Inequalities (6.39) and (6.40). Recall that **T** represents the set of on-chip memory resource types, of which there are M, and num_k represents the total number of blocks of each type t_k. In (6.40), $blocks_single_{uk}$

is a constant representing the number of blocks of resource type t_k consumed by a single copy of FIFO f_u.

$$\forall f_u \in \mathbf{FE}, \qquad \sum_{k=0}^{M} p_{uk} \leq 1 \tag{6.39}$$

$$\forall f_u \in \mathbf{FE},\ \forall t_k \in \mathbf{T}, \qquad blocks_{uk} \geq N_{fu} \cdot blocks_single_{uk} - num_k + p_{uk} \cdot num_k \tag{6.40}$$

In Inequality (6.40) N_{fu} is the number of copies of FIFO f_u included in the system. In this formulation N_{fu} is a real variable and its value is constrained by Inequalities (6.41) to (6.45). In these inequalities α_u is an intermediate integer variable and NN_{fu} is an intermediate real variable for each f_u required to calculate the value of N_{fu} within the limitations of Integer Linear Programming. $\mathbf{P_i}$ again represents the set of partitioning options available for array a_i (assuming FIFO f_u services an access to array a_i) and Rng_i represents the range of different numbers of unrolled iterations that may be assigned to each array partition. Each $parts_{ij}$ represents the number of partitions array a_i is split into if partitioning scheme j is selected. X_{fu} represents the maximum number of copies of FIFO f_u that could be included (based on the resources that each would consume and the total resources available).

$$\forall f_u \in \mathbf{FE}, \forall z_{ij} \in \mathbf{P_i}, \qquad NN_{fu} \geq \alpha_u \cdot parts_{ij} - X_{fu} + z_{ij} \cdot X_{fu} \tag{6.41}$$

$$\forall f_u \in \mathbf{FE}, \forall z_{ij} \in \mathbf{P_i}, \qquad NN_{fu} \leq \alpha_u \cdot parts_{ij} + X_{fu} - z_{ij} \cdot X_{fu} \tag{6.42}$$

$$\forall f_u \in \mathbf{FE}, \forall y_{im} \in Rng_i, \qquad N_{fu} \geq NN_{fu} \cdot I_{im} - X_{fu} + y_{im} \cdot X_{fu} \tag{6.43}$$

$$\forall f_u \in \mathbf{FE}, \forall y_{im} \in Rng_i, \qquad N_{fu} \leq NN_{fu} \cdot I_{im} - X_{fu} + y_{im} \cdot X_{fu} \tag{6.44}$$

$$\forall f_u \in \mathbf{FE}, \qquad N_{fu} \leq \sum_{k=1}^{M} p_{uk} \cdot X_{fu} \tag{6.45}$$

Recall that, if FIFO F_u supplies array access acc_x, the number of copies of f_u included in the system must be an integer multiple of the number of unrolled iterations for which access acc_x is supplied by a memory port. The number of copies of access acc_x supplied by a memory port is a function of how the array that acc_x reads is partitioned, and how many copies of acc_x are assigned to execute sequentially on each port. These factors are deter-

mined for each array, a_i, by the z_{ij} and y_{im} binary variables respectively (Section 6.7.1). If z_{ij} is one then array a_i is split into $parts_{ij}$ partitions, and if y_{im} is one then I_{im} copies of acc_i access each partition. Constraints (6.41) to (6.44) therefore force the number of FIFOs to be a multiple of the number of memory accesses, while Constraint (6.45) forces the number of FIFOs to be zero unless the FIFO has been selected for implementation.

We also allow the unrolled copies of acc_i that would supplied by memory ports to be served by additional External FIFOs instead, freeing the port to be used by other accesses if necessary. However, these additional FIFOs must be treated differently to the other External FIFOs as they affect the value of cycles per pipeline stage (T) that may be achieved, rather than the level of unroll. For each FIFO f_u a further set of binary variables, pp_{uk}, is created. If pp_{uk} takes a value of one then the unrolled iterations that would otherwise be assigned to access memory ports are also fed by FIFOs which are implemented in resources of on-chip memory type t_k. For each access, acc_x, in the innermost loop of the target loop we include an integer variable, NP_x, which models the number of unrolled copies of the access served by memory ports. If pp_{uk} is set to one then NP_x additional FIFOs must be included in the system. A set of integer variables, $blocks_p_{uk}$, is used to model the number of blocks of each resource type consumed by the additional FIFOs. The additional variables are constrained by Inequalities (6.46) to (6.48). Inequality (6.46) forces the additional f_u FIFOs to be implemented in only a single memory resource type. $blocks_single_{uk}$ is a constant representing the number of blocks of type t_k consumed by a single instance of FIFO f_u.

$$\forall f_u \in \mathbf{FE}, \qquad \sum_{k=1}^{M} pp_{uk} \leq 1 \tag{6.46}$$

$\forall f_u \in \mathbf{FE}, \forall t_k \in \mathbf{T},$

$$blocks_p_{uk} \geq NP_x \cdot blocks_single_{uk} - num_k + pp_{uk} \cdot num_k \tag{6.47}$$

$$\forall f_u \in \mathbf{FE}, \qquad \sum_{k=1}^{M} pp_{uk} \leq \sum_{k=1}^{M} p_{uk} \tag{6.48}$$

With the addition of External FIFOs to the formulation, the constraints modeling the overall memory resource usage of the system must again be updated. Inequality (6.31) from Section 6.7.2 must be updated as shown in Inequality (6.49) to take into account any memory blocks used by the Internal FIFOs.

$$\forall t_k \in \mathbf{T}, \quad \sum_{i=1}^{N} blocks_{ik} + \sum_{b_w \in \mathbf{B}} blocks_{wk} + \sum_{f_u \in \mathbf{FI}} blocks_{uk} + \sum_{f_v \in \mathbf{FE}} blocks_{vk} \leq num_k \quad (6.49)$$

Constraints (6.32) and (6.33) from Section 6.7.2, which constrain the number of accesses to each array that are allocated to data reuse structures and limit each access to be implemented using only one data reuse option, must also be updated to include the External FIFOs. The updated constraints are represented by Inequalities (6.50) and (6.51). In these inequalities FE_i represents the set of External FIFOs that serve any access to array a_i. $\mathbf{ACC_i}$ represents the set of accesses to array a_i and $\mathbf{FI_{ix}}$ represents the set of External FIFOs that serve each accesses, acc_{ix}, in $\mathbf{ACC_i}$.

$$\forall a_i \in \mathbf{A}, \qquad R_i \leq \sum_{b_w \in \mathbf{B_i}} \sum_{k=1}^{M} e_{wk} \cdot accesses_w + \sum_{f_u \in \mathbf{FI_i}} \sum_{k=1}^{M} p_{uk} + \sum_{f_v \in \mathbf{FE_i}} \sum_{k=1}^{M} pp_{vk} \quad (6.50)$$

$$\forall a_i \in \mathbf{A}, \forall acc_{ix} \in \mathbf{ACC_i}, \qquad \sum_{b_w \in \mathbf{B_{ix}}} \sum_{k=0}^{M} e_{wk} + \sum_{f_u \in \mathbf{FI_{ix}}} \sum_{k=1}^{M} p_{uk} + \sum_{f_v \in \mathbf{FE_{ix}}} \sum_{k=1}^{M} pp_{vk} \leq 1 \quad (6.51)$$

6.7.4 Cost Functions

There are four functions called in the scheduling search algorithms presented in Section 6.2 that make use of the ILP formulation presented here, each requiring a different cost function to be maximised or minimised. In the four subsections that follow the formulation of the cost function for each of these functions is described, along with any additional constraints that are required.

Maximising Unroll

The 'maximise_unroll' function must optimise the memory subsystem so that the maximum unroll factor can be achieved for a given loop level within a fixed clock cycles per

pipeline stage (T). The unroll factors for all other loop levels have fixed values that must be honoured. A real variable, N, is used to represent the maximum unroll that may be achieved for the given level. The cost function that must be optimised is then the maximisation of N. A number of sets of constraints must be included in the formulation to force N to model the maximum unroll factor.

The maximum unroll factor that may be achieved is determined by the number of copies of each access that may be executed within a window of T clock cycles. For each access, acc_x, the number of parallel copies is dependent on two factors: the number of access copies that may execute through memory ports and the number of copies that may execute through FIFOs. Assuming access acc_x reads or writes to or from array a_i, the number of copies that may execute through memory ports is determined by the z_{ij} and y_{im} binary variables described in Section 6.7.1. A real variable, NP_i, is created for each array to model the number of copies of any access to that array that may be executed through memory ports, and is constrained by Inequalities (6.52) and (6.53). Each α_{ikm} value is a constant representing the number of copies of any access to array a_i that may be made through memory ports if z_{ik} and y_{im} are both one. Each X_i represents the maximum value of α_{ikm} across all the partitioning schemes.

$$\forall a_i \in \mathbf{A}, \qquad NP_i \geq \sum_{k=1}^{P_i} \alpha_{ikm} \cdot z_{ik} - X_i + X_i \cdot y_{im} \tag{6.52}$$

$$\forall a_i \in \mathbf{A}, \qquad NP_i \leq \sum_{k=1}^{P_i} \alpha_{ikm} \cdot z_{ik} + X_i - X_i \cdot y_{im} \tag{6.53}$$

For each access, acc_x, a real variable, U_x, is created to denote the total number of copies of acc_x that may be unrolled and is constrained to be the sum of copies that may be executed through ports and the number of copies that may be executed through FIFOs. These constraints, which assume acc_x accesses array a_i, are represented by equation (6.54). $\mathbf{FE_x}$ represents the set of External FIFOs that service access acc_x.

$$\forall acc_x, \qquad U_x = NP_i + \sum_{f_u \in \mathbf{FE_x}} N_{fu} \tag{6.54}$$

The maximum level of unroll that may be achieved is then constrained to be the minimum U_x across all the accesses in the loop.

$$\forall acc_x, \qquad N \leq U_x \tag{6.55}$$

Minimising the Imperfectly Nested Stages

As in the previous chapter the 'memory_minimise_Z_p' function is required to update the memory system so that the minimum number of imperfectly nested pipeline stages for all levels up to and including the pipelined level may be achieved. With the inclusion of loop unrolling in the methodology the minimum Z_p must now be achieved for given values of T, S, and unroll factors. Once again this function must be heuristic due to the difficulty in modeling which memory access operations would be required to execute in parallel to achieve a minimum number of stages (see Section 5.6.2 for further details).

As in Chapters 4 and 5, the imperfectly nested pipeline stages are included in integer multiples of the number of perfectly nested stages, S. Because there is no change from the previous chapter in how imperfectly nested stages are treated, the formulation for this cost function is almost identical to that for the 'memory_minimise_Z_p' function in Section 5.6.3. The additional complexity of loop unrolling is dealt with by constraints (6.52) to (6.55). Bounding the unroll value, N, to take the correct fixed input value ensures that the correct number of copies of each memory access operation may still be accessed in the given T cycle window. The only additional concern at this point in the search is to estimate the smallest number of sets of S imperfectly nested stages that must be included so that the imperfectly nested operations can map onto the same ports as used by the perfectly nested operations without conflict.

As was explained in Section 5.6.3, the number of accesses, acc_k, that may be made to a bank of resource type t_k or an off-chip memory block g_k in a window of S stages is defined by equation (6.56). Once again $ports_k$ represents the number of ports on a single bank of memory k, and iss_k is the minimum number of cycles between successive accesses to the same port.

$$acc_k = \left\lfloor \frac{T}{iss_k} \right\rfloor \cdot ports_k \tag{6.56}$$

In a window of $(Z_p + 1) \cdot S$ stages we require that a single instance of each perfectly nested and imperfectly nested memory operation assigned to each memory block can execute without conflict. Z_p is included as an integer variable in the ILP formulation and this constraint on its value is enforced for the on-chip and off-chip memory blocks respectively by Inequalities (6.57) and (6.58). $\mathbf{T}$ is the set of on-chip memory types and G is the set of off-chip memory blocks. As defined in Section 6.7.1, $accesses_i$ is a variable denoting the number of perfectly nested accesses to each array a_i once data reuse is accounted for, and $accesses_{ix}$ is the number of perfectly nested accesses to array a_i that are assigned to off-chip memory block g_x. In both inequalities imp_i is a constant representing the number of imperfectly nested accesses to array a_i. Recall that the c_{ik} binary variable is one if array a_i is assigned to on-chip memory type t_k and the d_{ix} variable is one of array a_i is assigned to off-chip memory block g_x. In this context X_i represents the maximum value of $accesses_i + imp_i$ for each array.

$$\forall a_i \in \mathbf{A}, \forall t_k \in \mathbf{T}, \qquad acc_k \cdot (Z_p + 1) \geq accesses_i + imp_i - X_i + X_i \cdot c_{ik} \tag{6.57}$$

$$\forall g_x \in \mathbf{G}, \qquad acc_x \cdot (Z_p + 1) \geq \sum_{a_i \in \mathbf{A}} accesses_{ix} + \sum_{a_i \in \mathbf{A}} imp_i \cdot d_{ix} \tag{6.58}$$

Minimising the Total Stages

The goal of the 'memory_minimise_S_{tot}' function is to minimise the number of stages, both perfectly and imperfectly nested, executed during the entire loop. This must be done within fixed values for T, S, Z_p and the unroll factors. As with the 'memory_minimise_Z_p' function, the formulation for the cost function used with 'memory_minimise_S_{tot}' now that loop unrolling is included is similar to that used in the previous chapter for no unroll (see Section 5.6.4). As before, integer variables must be included to model the number of imperfectly nested stages included for each level in the loop level[21], Z_i (where i is the

[21]While the number of imperfect stages for the pipelined level is fixed at this point, we are still free to vary the number of imperfect stages for the other loop levels.

loop level), and the additional imperfectly nested stages required at each level in the loop, ZF_i, to fill/initialise the data reuse structures that are included in the design. The cost function (the total number of stages) is just a weighted sum of the number of stages at each loop level, with the weights determined by the number of iterations and unroll factor at each level.

For each level i in the loop above the innermost level and up to the pipelined level the number of imperfectly nested stages required to execute the native loop operations (not the data reuse fill operations), Z_i, is determined by constraints identical to those used to model the Z_p cost function in the previous section.

The constraints required to model the value of each ZF_i variable are similar to those used in Section 5.6.4, but are complicated slightly by the inclusion of array partitioning to the formulation. The number of stages required to fill a reuse structure was fixed in Section 5.6.4, and the only consideration in calculating the total fill at each level was whether each structure was included in the design or not. The inclusion of array partitioning renders variable the number of ports through which array data, required to initialise a reuse structure, may be read. This means that the time taken to fill each reuse structure becomes a variable whose value must be modeled. For each buffer, b_x, an integer variable, $fill_b_x$, is used to denote the number of data array elements that must be read from each array partition for that buffer. The value of each $fill_b_x$ is constrained by Inequality (6.59). Z_x is a constant representing the maximum possible value of $fill_b_x$. $fill_b_{xj}$ is a constant representing the number of array elements read from each port to initialise buffer b_x when partitioning scheme j is used for array a_i. We are assuming that buffer b_x reuses data from array a_i.

$$\forall b_x \in \mathbf{B}, \qquad fill_b_x \geq \sum_{j=1}^{P_i} z_{ij} \cdot fill_b_{xj} - Z_x + Z_x \cdot \sum_{k=1}^{M} e_{xk} \tag{6.59}$$

A similar variable and constraint set is required for each internal FIFO. An integer variable, $fill_f_x$, represents the number of reads required from each memory port to initialise Internal FIFO f_x, and its value is constrained by Inequality (6.60). Here W_x is a constant representing the maximum possible value of $fill_f_x$. $fill_f_{xj}$ is a constant representing

the number of array elements read from each port to initialise buffer f_x when partitioning scheme j is used for array a_i.

$$\forall f_x \in \mathbf{FI}, \qquad fill_f_x \geq \sum_{j=1}^{P_i} z_{ij} \cdot fill_f_{xj} - W_x + W_x \cdot p_{xk} \tag{6.60}$$

The formulation is slightly different for the External FIFOs. Recall that for each External FIFO, f_x, we already have an integer variable, α_x, which partially defines the number of copies of f_x allocated to reuse data from each memory port. Multiple partitions of array a_i may be grouped together in a single memory block by the y_{im} binary variables and this can increase the number of FIFOs assigned to each memory port. We also have the pp_x binary variable for each External FIFO that defines whether the extra instance of External FIFO f_x is included to feed the first unrolled iteration in each set. The number of array elements that must be read from each memory port is therefore defined by Inequality (6.61).

$$\forall f_x \in \mathbf{FE}, \qquad fill_f_x \geq \alpha_x \cdot fill_f_{xm} + pp_x \cdot fill_f_{xm} - Z_{xm} + Z_{xm} \cdot y_{im} \tag{6.61}$$

With the size of the fill for each data reuse options defined, the number of sets of imperfect stages required at each level, ZF_i (where i is the loop level), can be defined. For the on-chip memories, where only a single array may be assigned to each, the number of sets of stages required at each level is bound by Inequality (6.62).

$$\forall a_j \in \mathbf{A}, \forall t_k \in \mathbf{T}, \qquad ZF_i \cdot acc_k \geq \sum_{f_x \in \mathbf{FI_j}} fill_f_x + \sum_{f_x \in \mathbf{FE_j}} fill_f_x + \sum_{b_x \in \mathbf{B_j}} fill_b_x \tag{6.62}$$

The formulation is slightly more complex for the off-chip memories, where multiple arrays may be assigned to the same block. An integer variable must be introduced for each array, a_j, for each off-chip memory block, g_s, to denote the number of reads for each array assigned to each off-chip memory. This variable, $fill_array_{js}$, is constrained by Inequality (6.63), and the imperfect stages required to read all the data for the fill for each off-chip memory is defined by Inequality (6.64).

$$fill_array_{js} \geq \sum_{f_x \in \mathbf{FI_j}} fill_f_x + \sum_{f_x \in \mathbf{FE_j}} fill_f_x + \sum_{b_x \in \mathbf{B_j}} fill_B_x - X + X \cdot d_{js} \tag{6.63}$$

$$\forall g_s \in \mathbf{G}, \qquad ZF_i \cdot acc_s \geq \sum_{a_j \in \mathbf{A}} fill_array_{js} \tag{6.64}$$

Minimising the Fill of Data Reuse Structures

To achieve a given combination of unroll factors and cycles per pipeline stage a certain number of data reuse structures may need to be included to allow sufficient parallel access. The purpose of the 'minimise_fill' and 'minimise_fill_given_unroll' functions, used in the search algorithm in Section 6.2, is to select the set of data reuse options that achieves the required unroll factors and cycles per stage with the minimum total fill across all loop levels. The difference between the two functions is that the 'minimise_fill' function operates for an unroll factor of one at the current target level, though the unroll factors may be greater than one at loop levels below this. The 'minimise_fill_given_unroll' function may require an unroll greater than one at the current target level. The cost function in both case is similar to the cost function required to minimise S_{tot}. It is merely a weighted sum of the ZF_i variables from the previous section.

6.7.5 Pruning the Partitioning Options

The number of binary variables in the proposed ILP formulation for array to memory allocation scales linearly with the number of partitioning options available for each sub-array. In the worst case, when the address function for any dimension of a sub-array is dependent only on the loop iterator for the unroll level, the sub-array may be split into any number of partitions from 1 to N, where N is the number of iterations at the unroll level. Such cases could lead to large numbers of binary variables and render the ILP unsolvable. To keep the size of the ILP within bounds that can reasonably be considered solvable the partitioning options for each sub-array are pruned (where necessary) so that a maximum O_p partitioning schemes are considered for each sub-array within the ILP. The value of O_p

is defined by the user in this approach. Reducing the solution space searched by the ILP formulation means that the optimum solution may not always be found as the partitioning scheme required could have been pruned. To reduce this risk the goal during pruning is to remove any infeasible solutions first, followed by those deemed less favourable, until only the O_p most promising options remain. As such it is hoped that the best solution will still be available for the ILP to find, though it cannot be guaranteed, especially if the user-set value of O_p is small.

Removing Infeasible Options

One of the limitations of the array to memory placement approach presented in this work is that, although sub-arrays may be partitioned, all partitions of the same sub-array must be assigned to banks of the same memory resource type. The size of each array may therefore limit which types of memory resource it may be stored in. For example, a 1000x1000 element floating point array is too large to be stored in the on-chip memory of any current FPGA so it can only be allocated to off-chip memory resources. Also, as previously mentioned, the user may impose a binding on any array, limiting its placement to a limited subset of the available memory types. These two factors will produce a list of memory resource types to which each array may be assigned. For a given target platform there will be fixed number of banks of each resource type, and the upper bound number of partitions into which any sub-array may be split is defined by the maximum number of available banks across all resource types to which it may be assigned. Any candidate partitioning scheme for each sub-array that breaches the upper bound can be pruned without affecting the search space.

Reducing the Partitioning Options

The partitioning schemes ultimately selected for implementation by the ILP formulation must allow the same level of unroll to be achieved across all the accesses in the loop. Hence the level of unroll that can be achieved for each partitioning scheme is of key interest when selecting which partitioning schemes should be discarded before running

the ILP. The options kept for each sub-array must overlap as much as possible with the those kept for the other sub-arrays in the values of unroll that may be achieved, otherwise the ILP will be limited to a few values of unroll or, at worst, be unsolvable. Due to the possibility of data reuse structures providing extra instances of an access, the level of unroll that may be achieved for a given partitioning scheme is not merely a function of the number of sub-array partitions. For each access to each sub-array the maximum number of data reuse structures that could be achieved within the resource constraints must also be considered when calculating the range of unroll value that may be achieved for each partitioning scheme.

Unrolling a loop by a given factor, U, will essentially lead to the creation of U copies of the pipeline to implement the loop operations that run in parallel. If U is a factor of the number of loop iterations at the unroll level, N, then each pipeline will execute N/U iterations. However, if U is not a factor of N then some pipelines will execute one more iteration than others, with the maximum number of iterations on any pipeline determined as $\lceil N/U \rceil$. The overall execution time for the unrolled loop will be a function of $\lceil N/U \rceil$, and hence it would not be expected to scale strictly monotonically with unroll. A smaller value of unroll may produce the same execution time as a larger value while using fewer logic resources to do so. For example, if a loop has 1000 iterations, unrolling it 500 times will have the same benefit as unrolling 999 times, but at roughly half the logic cost. This means that certain values of unroll will be more desirable than others. It therefore follows that array partitioning schemes which allow these more desirable values of unroll to be achieved will be preferable to those which allow less favourable values of unroll. The goal of this section is to reduce the number of partitioning options considered within the ILP formulation, and this criterion is useful in determining which partitioning schemes should be kept.

Another important issue when selecting which partitioning options to remove from the search space is memory resource usage. The memory resources consumed by each sub-array may often vary non-linearly with the number of partitions. The number of banks of a given resource type k consumed by sub-array A, $banks_{kA}$, can be found using

Table 6.4: Memory usage for seven array partitioning schemes.

Number of partitions	Partition size	Number of banks
10	768	10
8	960	8
5	1536	10
4	1920	8
3	2560	9
2	3840	8
1	7680	8

Equation (6.65). In (6.65) num_parts_A is the number of partitions into which sub-array A has been split, $part_size_A$ is the size of each sub-array partition and $block_size_k$ is the size of each bank of resource type k.

$$banks_{kA} = num_parts_A \cdot \lceil part_size_A / block_size_k \rceil \tag{6.65}$$

As the number of partitions increases the partition size decreases, and vice versa. In cases where $part_size_A$ is a multiple of $block_size_k$ the resource usage will be lower than any case where $part_size_A$ is slightly larger than some integer multiple of $block_size_k$. Hence an array partitioning scheme with a larger number of smaller partitions may actually require fewer memory resources than a scheme with fewer, larger partitions if the smaller partition size better matches the size of the memory blocks. Consider an example sub-array, A, with 7680 words, targeted to a memory resource type with 1024 words per bank. Assuming that the seven partitioning schemes proposed in Table 6.4 are feasible, each scheme will use the number of banks shown in the third column. The first two options in Table 6.4, with 10 and 8 partitions in sub-array A, offer a tradeoff between the number of memory ports available (*i.e.* the potential for unroll) and the number of memory banks used. However, all of the other partitioning options use at least the same number of banks as partitioning A into 8 sets, but offer fewer ports. Therefore, on appearance, there is no benefit in any partitioning option below 8 sets and so these options should be pruned. However, this is only true if each sub-array could be considered independently, and this is not the case.

The partitioning scheme chosen for each sub-array must ultimately allow the same

number of parallel copies for each access in the loop. Consider a second sub-array, B, which is accessed in the same loop as A. If sub-array B can only be partitioned into 3 sets then the unroll that may be achieved is constrained to a multiple of 3 (or to be 1, *i.e.* no unroll). Partitioning sub-array A into 8 or 10 sets will require the unroll to be a multiple of 8 or 10 respectively[22], so clearly neither of these schemes can be used. If B is partitioned into 3 sets then the only viable option from Table 6.4 for A is to also partition it into 3 sets. Thus we cannot simply ignore all partitioning schemes with fewer partitions and equal or greater resource usage than another scheme. However, the non-linear variance of consumed memory resources with the number of partitions can still be used to improve the range of unroll values that may be achieved using each partitioning scheme.

Assume we are given two partitioning schemes, P_1 and P_2, which split sub-array A into N_1 and N_2 sets respectively and consume $banks_1$ and $banks_2$ memory resources. If N_2 is a multiple of N_1 and $banks_2 = banks_1$ then P_1 would essentially be the same memory layout as P_2, but with extra multiplexors to select between the memory ports and reduce the number of array partitions. Hence selecting partitioning scheme P_1 in the memory optimisation would mean that sub-array A could be split into either N_1 or N_2 partitions without using more than $banks_1$ memory resources. This interchangeability of array partitioning schemes can allow certain schemes to be pruned from the solution space without affecting the number of different unroll values that can be achieved, but it requires the ILP formulation for memory optimisation to be modified. Given a set of partitioning schemes which each split sub-array A into some number of sets that is a multiple of N_1, and which all have a resource usage equal to $banks_1$, a new partitioning option can be created. This new type of partitioning scheme is referred to as a *combined partitioning scheme* from here on. If this *combined partitioning scheme* is selected for implementation by the ILP formulation then the number of partitions in sub-array A is not fixed to one specific value, but to be some multiple of N_1. An integer variable is used within the ILP to fix which multiple of N_1 used. Obviously the value of this integer variable must be bound to a fixed range as not all integer multiples of N_1 will be feasible. To simplify

[22]The value of unroll may be a multiple of the number of sets the sub-array is partitioned into because data reuse structures may be inferred to allow greater levels of unroll, but the same number of reuse structures must be instantiated for each array partition.

the bounding of this integer variable the individual partitioning schemes included within each *combined partitioning scheme* must allow an uninterrupted range of multiples to be achieved between 1 and some maximum value. Assume that partitioning scheme P_2 has a resource usage equal to scheme P_1, and splits sub-array A into $z \cdot N_1$ sets (where z is some integer > 1). P_1 and P_2 may only be grouped together in a *combined partitioning scheme* if there exist $z - 2$ other partitioning schemes which also have resource usages equal to P_1 and which split split sub-array A into the $z - 2$ other integer multiples of N_1 between 1 and z. These other partitioning schemes must also be grouped together within the *combined partitioning scheme*.

With all these issues in mind, the partitioning options available for each sub-array are pruned in two stages. In the first stage the following procedure is executed:

1. For each partitioning scheme for each sub-array, list all values of unroll that can be achieved for the given scheme when data reuse options are considered.
2. For each sub-array, create any possible *combined partitioning schemes*.
3. For each partitioning scheme for each sub-array, remove any values of unroll that cannot be achieved by all the sub-arrays.
4. For each partitioning scheme for each sub-array, remove any unroll values that are greater than another possible unroll value but which lead to the same number of loop iterations per pipeline.
5. For each sub-array, remove any partitioning schemes for which all corresponding values of unroll have been removed.

The second stage of the pruning procedure, which is run for each sub-array in turn (by order of increasing number of available partitioning schemes), is as follows:

1. For each partitioning scheme for the given sub-array, remove any further values of unroll that cannot be achieved by all the sub-arrays, using only the (up to) O_p partitioning schemes that have been kept in the case of sub-arrays for which the procedure has already been run.

2. For the given sub-array, remove any partitioning schemes for which all corresponding values of unroll have been removed.

3. Divide the range of unroll values between 1 and the maximum available unroll into $O_p - 2$ windows. Each window is considered filled if a partitioning scheme is selected to be kept such that at least one value of unroll in the window is still achievable.

4. For the given sub-array, the schemes with 1 partition and the maximum number of partitions are kept. The other $O_p - 2$ schemes to be kept are selected such that the number of filled windows is maximised.

6.8 Results

The combined scheduling and memory optimisation approach presented in this chapter has been applied to eight test loops. For each loop pipelining was attempted at each level, with the unroll possibilities examined for all levels up to and including the current pipelining level. A target platform was specified in each case featuring the largest Altera Stratix II FPGA (an EP2S180 device) and eight 4MB banks of off-chip SRAM. The methodology for pruning the partitioning options for the arrays in the loop requires the designer to specify the maximum number of options that are considered for each array within the memory optimisation ILP. For the results presented here a maximum of 8 partitioning schemes may be kept for each array. Of the eight test loops, three use 8-bit fixed point arithmetic, and the remaining five use single precision floating point number representation. The fixed point kernels are an image edge detector with a 3x3 pixel window, a motion estimator with a search window of +/-2pixels, and a 2D median filter with a 3x3 pixel window. The floating point kernels are a matrix-matrix multiply, an LU decomposition [145], implementations of the Minimum Residual [144] and Successive Over Relaxation [143] algorithms, and a hydrodynamics kernel taken from the Livermore loops [142]. All of the floating point kernels use matrices of order 1000. The C codes for the benchmarks are included in Appendices C and D.

Table 6.5 details the parameters produced by the scheduling search to provide the

Table 6.5: Scheduling results for edge detection (ED), motion estimation (ME), matrix-matrix multiply (MMM), hydrodynamics (HD), successive over relaxation (SOR), MINRES (MIN), LU Decomposition (LU) and median filter (MED) kernels. The bandwidth utilisation column lists the percentage of the available off-chip memory bandwidth used.

Loop	Pipeline level	Unroll factors	T	S	II	Z_p	Cycles	Bandwidth Utilisation
ED	1	$< 1, 8, 3, 3 >$	1	16	3	1	17,671	94.61%
ME	1	$< 1, 4, 5, 5 >$	1	16	1	1	16,463	99.56%
MED	1	$< 1, 8, 3, 3 >$	1	16	3	1	17,671	94.61%
MMM	1	$< 1, 8, 1 >$	1	20	1	1	126,125,000	99.37%
HD	1	$< 1, 1 >$	1	40	41	0	1,006,602	99.34%
SOR	1	$< 1, 8 >$	1	25	1	1	126,274	99.15%
MIN	1	$< 1, 8 >$	1	25	1	1	126,418	99.12%
LU	1	$< 1, 1, 48 >$	1	66	1005	2	9,077,486	27.54%

shortest schedule for each loop, as well as the overall schedule length. Recall that loop level one represents the outermost level, so in each case the best solution was found when pipelining the outermost loop. This should possibly not come as a surprise since we only allow unrolling at levels up to the pipelined level, so raising the pipelining level offers greater possibility for unroll in each case. In every case the best solution was found with a stage length (T) of one cycle. This means that, once each pipeline has filled (S initiation intervals have elapsed), every perfectly nested resource in the pipeline is utilised on every clock cycle until the end of a loop level with imperfectly nested operations. As a result, each pipeline should use both the logical resources and memory ports for a high percentage of the total execution time, representing good efficiency. In each case, except for the hydrodynamics kernel, unroll was achieved at one or more levels, increasing the parallelism achieved compared to the results in the previous chapter. No unroll could be achieved for the hydrodynamics kernel as its innermost loop features six accesses to five separate arrays that cannot be serviced by any data reuse structures. All of these arrays must be stored off-chip due to their size, and there are only 8 off-chip memory ports, so these arrays cannot be partitioned to provide the parallel access required for unroll. In Chapter 4 using just the outerloop pipelining approach we achieved an average speedup over sequential execution (a measure of parallelism) of 7.1x. With the addition of automated array to

memory placement and data reuse selection in Chapter 5 an average speedup of 24x was achieved over sequential execution. In this chapter, with the addition of array partitioning and unrolling to the methodology, an average speedup of 303x has been achieved which represents 43x improvement over pipelining alone, and a 12.6x improvement over pipelining with data reuse.

The bandwidth utilisation column in Table 6.5 details the percentage of the available off-chip memory bandwidth used by each kernel. In most cases the bandwidth utilisation is in excess of 99% indicating that, even with 8 off-chip SRAMs available. This suggests that the array partitioning methodology presented in this work has been successful at dividing arrays across memories to maximise the speed of execution. It is worth noting that, for most of the benchmarks the bandwidth utilisation has not dropped below that achieved in Chapter 5 where only a single bank of SRAM was available. The Edge Detection and Median Filter kernels fail to use 5.4% of the available bandwidth. This is due to some memory ports not being used on every clock cycle during the filling of data reuse structures and the execution of imperfectly nested pipeline stages. The low bandwidth utilisation of the LINPACK kernel is due to the fact that the main data array read by this algorithm is split over only two banks of SRAM (odd matrix rows in one bank and even in another, as in the manual implementation of Chapter 3). There is no other way to partition that data to allow for greater parallelism than is achieved in the solution presented here, so 27% represents the best possible bandwidth utilisation that the methods presented here could achieve. It is worth noting however that, in the LINPACK case, the level of parallelism achieved (number of unrolled iterations) is not bound by the available off-chip bandwidth, but is in fact bound by the number of on-chip SRAM banks available to implement data reuse buffers. The design proposed consumes all of the M4K memory blocks available on the Altera EP2S180 device specified.

In the previous chapter a reasonably tight lower bound schedule length was produced by the search. This lower bound estimated the minimum possible schedule available given the target platform and, although this lower bound might not always be achievable, it was used as a basis for comparison to determine the quality of the solution produced.

In the work in this chapter a number of heuristic decisions are made before the scheduling search, and restrictions are placed on the solution space explored by the search. As a result, it is not possible to produce such a tight lower bound for comparison. Instead a cruder lower bound is used for comparison, based on the minimum time that is required to access each array element used by the loop. For read only arrays or write only arrays, this the sum of the sizes of the arrays (or the parts of them used by the loop) divided by the number of off-chip memory ports, and for arrays that are both read and written, twice the array size must be included in the summation. Table 6.6 lists the absolute lower bound execution time for each loop based only on the time taken to access the arrays, along with execution time found by the scheduling search and the percentage deviance from the lower bound. We also include the access ratio for each loop (the number of accesses to off-chip memory divided by the lower bound number of accesses assuming all data reuse is exploited) and the percentage of array accesses in the original C-code loop supplied by data reuse structures. The final column lists the speedup of the schedule found over a completely sequential execution (*i.e.* the average parallelism achieved). Across the 8 loops tested we achieved an average speedup of 300x. While this figure sounds impressive, it is difficult to quantify relative to existing work. The percentage deviance from the lower bound gives a simpler basis for comparison, given that the goal of this work is to minimise schedule lengths.

For five of the loops tested the schedule found is within 10% of the absolute lower bound, suggesting that the methodology presented here can be reasonably successful. In these cases the extra cycles in the schedule can be attributed to the need to meet dependence constraints, which are not modeled in our simple lower bound, and a relatively small number of duplicate array accesses remaining in some cases due to the filling of some data reuse structures. However, the other 3 test loops fared much worse, with the schedule lengths achieved in two cases being two to three orders of magnitude greater than the predicted lower bound. For the hydrodynamics kernel the schedule is just over 34% longer than the lower bound. This is because the partitioning methodology could not partition one of the arrays read by the loop to allow unroll. As a result there is no benefit in partitioning any of the other four arrays used. The end result is that only 6 of

Table 6.6: Scheduling results for edge detection (ED), motion estimation (ME), matrix-matrix multiply (MMM), hydrodynamics (HD), successive over relaxation (SOR), MINRES (MIN), LU Decomposition (LU) and median filter (MED) kernels. The speedup figure is relative to completely sequential execution.

Loop	Achieved schedule length (cycles)	Lower bound (cycles)	Deviance from lower bound (%)	Access ratio	Access reduction (%)	Speedup
ED	17,671	16,384	7.8	1.02	90	270
ME	16,463	16,416	0.2	1.00	96	794
MED	17,671	16,384	7.8	1.02	90	196
MMM	126,125,000	375,000	33,533	334	50	134
HD	1,006,602	750,000	34.2	1.143	27	81
SOR	126,274	125,250	0.8	1.00	50	151
MIN	126,418	125,250	0.9	1.00	50	135
LU	9,077,486	250,125	3,629	8.0	97	663

the available 8 off-chip memory ports are used, removing 25% of the available bandwidth and increasing execution time by a similar percentage.

The matrix multiply kernel schedule is over 300x longer than the lower bound. This is because there was no scalable data reuse scheme available for one of the input matrices that could be implemented using the on-chip memory resources available on the EP2S180 device. When the rows of matrix two are multiplied by the columns of matrix one, all the rows of matrix two must be read from external memory for each column of matrix one. Hence each row of matrix two is read 1000 times. Given a device with 10 times the on-chip memory available on the EP2S180 device, all of matrix two could be buffered on-chip, partitioned over hundreds of memory blocks. The access ratio would then drop to 1 and the schedule length would approach the lower bound. We suspect that the actual minimum schedule that may be achieved given the available resources could be much closer to the schedule found than the lower bound predicted by the memory bandwidth available.

The situation is similar for the LU decomposition kernel as the schedule found is over 35x longer than the lower bound. Given enough on-chip memory to buffer the entire 1000x1000 element matrix on-chip the access ratio could be reduced by a factor of 8. However, this would still leave the schedule length around 4x greater than the lower

bound. This is partially due to the loop carried dependences in the loop that force the large initiation interval (see Table 6.5), and the iterative nature of the LU decomposition kernel. This suggests that simply basing the lower bound execution time on the minimum time for memory accesses may not be appropriate for every loop. However, a single precision floating point version of the LU decomposition hardware accelerator produced in Chapter 3 can be synthesised with 48 parallel pipelines (akin to the unroll of 48 used here) and the schedule length is around 7,500,000 cycles. The solution produced here is roughly 20% longer than this, which is probably a more accurate estimation of the deviance form the actual minimum schedule that may be achieved.

6.9 Summary

In this chapter we have presented unrolling and array partitioning extensions to the combined pipelining and memory optimisation approach presented in Chapter 5. A formal methodology has been presented to explore the unroll options for each level in the loop up to the pipelined level. Methods have also been described to expose the partitioning options available for each array so that accesses in unrolled iterations can be guaranteed to access separate memory banks. Both the unrolling and partitioning search spaces are pruned heuristically, meaning that the goal of finding the schedule with the shortest length cannot be guaranteed, but the results show that, in 5 of the 8 loops we were able to get within 10% of the most optimistic lower bound schedule length. This suggests that, despite these heuristic decisions, the search can be reasonably successful in finding 'good' solutions. The remaining three loops were between 30% and 300x slower than the optimistic lower bound schedule, but we can identify clear reasons why this is the case and suggest that the actual minimum schedule length is much closer to the achieved schedule length. However, we have also seen that the implementation of the LU decomposition kernel is still 20% slower than an optimised manual implementation, suggesting that there is still some way to go in providing methods that can match the intuitive decisions made by experienced designers when targeting more complicated examples.

Chapter 7

Conclusion

7.1 Summary

This thesis has examined methods for the co-optimisation of datapaths and memory subsystems implemented on FPGAs, with the goal of minimising the execution time for a given nested loop within the resources available on a given target platform. The methodology developed integrates loop pipelining, array partitioning, array to memory allocation and the exploitation of data reuse and can be applied to a single regularly nested loop with loop bounds that are affine functions of the loop iterators. The algorithm proposed consists of four main components:

1. A top level algorithm to search the high level scheduling parameters associated with pipelining a nested loop at an arbitrary level, namely the pipeline stage length, the number of pipeline stages for perfectly nested loop operations, the number of pipeline stages for the imperfectly nested operations at each level in the loop, the pipeline initiation interval and the unroll factor for each loop level.

2. An ILP formulation to schedule the loop operations for the given set of high level scheduling parameters.

3. An extension of the methods presented in [84] for determining how array data should be partitioned to maximise the potential for loop unroll at a given loop level.

4. An ILP formulation to optimise the memory subsystem for the given set of high level scheduling parameters. This formulation combines the selection of a partitioning scheme for each array, the allocation of array data to physical memories and the selection of data reuse structures to implement.

The methods presented in this thesis may only be applied to one nested loop at a time. The loop may be imperfectly nested but is restricted to be regularly nested. This means that each level in the loop may have any number of imperfectly nested instructions within it, but only one separate loop (though this loop may itself be regularly imperfectly nested). The methods presented can be applied to loops with bounds that are affine functions of the loop iterators for previous levels, but the optimality of the results for the methods in Chapter 4 can only be guaranteed for loops with fixed bounds. Pointers that cannot be de-referenced at compile time are not supported. Array partitioning can be inferred for array address functions that are affine functions of the loop iterators in at least one dimension. Simple data reuse can alow be inferred for array accesses with affine addressing. The application of the methods up to the end of Chapter 5 have been automated. The methods in Chapter 6 can be automated in the future, but have been applied manually to achieve the results presented here. The tool implementing the methods up to Chapter 5 requires the user to supply a text based description of the dependence graph for the loop and the loop bounds for each level. Further to this a text based description of the memories (both on-chip and off-chip) available on the target platform must be supplied, along with a text based description of any advanced data reuse options not automatically inferred in this work (such as those presented in [68]). At the output the tool produces a text description of the schedule for the datapath and the VHDL for the controlling state machine. In the future a C (or other language) front end and VHDL back end to generate the datapath could be added to the tool to improve ease of use.

By extending pipelining above the innermost loop using methods based on an existing approach for VLIW processors [13] we have achieved an average speedup over typical inner loop only methods of 2.9x. In the absence of loop unrolling and array partitioning our methodology is able to produce an optimal (shortest schedule) or near optimal data-

path and corresponding memory subsystem, with an upper bound placed on the deviance of the solution found from the fastest possible solution. For the loops tested the maximum deviance from the lower bound was 0.7%, with the optimum solution found in most cases. The inclusion of array partitioning and loop unrolling in the methodology forces a greater number of heuristics to be included to maintain tractability. This results in potentially less accurate lower bounds on the execution time and a greater deviance in the solution found from this lower bound, but for the majority of the loops tested this remained below 10% of the optimal schedule length. This suggests that the methods presented can produce good solutions and potentially achieve results comparable to those produced manually. Furthermore, the lower bound schedule length is based on the results that could be achieved by a solution that uses 100% of the available memory bandwidth, so in achieving a schedule length that is within 10% of this we can surmise that the solution found is using over 90% of the available memory bandwidth. This suggests that co-optimisation of the datapath (schedule) and memory has been successful in utilising the available memory resources which typically form the bottleneck in FPGA based systems.

7.2 Future Work

This thesis has presented the basis for a combined scheduling and memory optimisation approach that could be integrated into the toolboxes of future high level synthesis compilers, but there remains a great deal of scope to expand upon this work to improve the applicability of this work to real world algorithms and FPGA based platforms. In this section we give a brief overview of some of the possible extensions to this work.

7.2.1 Clock Frequency Effects

Throughout this thesis the focus has been on minimising the schedule length for a target algorithm, but overall execution time also depends on the maximum clock frequency that may be achieved. In Chapter 4 we showed that clock frequency may reduce as pipelining moves from the innermost loop to the outermost loop, but there is the potential for the scheduling parameters (cycles per stage, number of stages) to also affect the clock

frequency. Furthermore, in Chapter 5 it was shown that the inclusion of data reuse structures can reduce schedule length but also potentially reduce the clock frequency. The levels of unroll exploited and the partitioning of arrays also have the potential to reduce clock frequency and all of these factors could be investigated in future work.

7.2.2 Extensions for SDRAM

The current approach targets only SRAM memories, but modern FPGA based platforms typically include SDRAM modules [147, 148]. SDRAM offers higher storage densities than SRAM at lower costs and is often required to support the storage requirements of modern video processing and scientific computing algorithms. Hence extending the current approach to support SDRAM would greatly increase its utility, but such extensions are non-trivial. Firstly, the pipelining approach used assumes a fixed latency for operations to each memory, but the read and write latencies of SDRAM are variable, depending on whether the required memory row is currently open for reading or writing or not. To cope with the variable latency the pipelining algorithm could simply assume the worst case latency for every access, but this would be inefficient. Instead the methodology could be extended to support stalling, assuming the smallest latency during scheduling and pausing the pipeline if the data takes longer to arrive than expected. Another option would be to statically schedule all of the SDRAM commands for the target algorithm so that the latency for each access would be known.

Extensions to cope with the variable latency of SDRAMs could be relatively simple, but further work would be required to maximise the efficiency of the memory accesses. The word length of an SDRAM module is typically between 32 and 64 bits [149], but clocking on the positive and negative edge of the clock gives an effective word length of twice this value. Multiple elements of arrays with word lengths less than the memory word length must be packed into a single memory word to achieve high access efficiencies, and this adds another dimension to the solution space during memory optimisation. Furthermore, the spatial locality [47] of successive accesses must be considered when considering array to memory allocation and scheduling. If successive reads access data in the same row of

memory, the additional latency incurred when opening a new memory row will only be applied to the first read and the subsequent data will be returned in fewer cycles.

Also key to maximising the efficiency of accesses to SDRAM is the use of pipelined bursts, where a command is issued to read or write a long sequence of data elements to adjacent memory addresses. For a burst of N accesses (assuming all N addresses are in the same memory row) the access latency for the first access may be quite long (of the order of 10 or more clock cycles), but subsequent accesses will be pipelined and incur a latency of a single cycle. The use of FIFOs to buffer a sufficient number of successive reads or writes is typical when using SDRAM and could be integrated into the memory optimisation flow.

7.2.3 Extensions for Multiple Nested Loops

The current methodology can only be applied to a single, regularly nested loop. While loop transformations can convert some irregularly nested loops into regularly nested loops or merge multiple loops into a single loop, this will not always be possible. Extending the methodology presented in this work to schedule irregularly nested loops and to optimise the memory subsystem across multiple nested loops in an algorithm would increase the applicability of this approach.

To optimise the schedule and memory subsystem for two or more distinct nested loops, the optimisation algorithm would be required to minimise the sum of the execution times for loops which dependences constrain to execute in sequence, or the maximum execution time across loops which may execute in parallel. The search of the solution space would have to be modified to trade scheduling and memory decisions across the multiple loops to find the global optimum. It may be possible to generalise the transformations applied to the LU decomposition kernel in Appendix C to cope with irregular nesting in the majority of cases, but this would need to be investigated.

7.2.4 Investigation of Heuristics

Both the modulo scheduling and memory optimisation algorithms developed in this work are based around Integer Linear Programming. While the ILP formulations for the loops in this work were all solvable in less than five minutes on a single desktop computer, scalability and the potential exponential growth in run time with the number of variables in the ILP could limit the applicability of this work to larger algorithms. To counter this problem it is possible to set an upper limit on the time allowed to solve each ILP. If the optimal solution is not found within this time, the best solution found so far (if one has been found) is returned. In the worst case the ILP solver may have to evaluate the cost function at the majority (or even every) point in the search space to be certain that the optimal solution has been found, but what has finally been shown to be the optimal solution could have been found early in the search. In such cases the optimal solution may still be returned by the ILP if it is forced to terminate early, although the user could not be certain this is the optimal solution.

Future work could evaluate the effects of different time limits and how far the resulting solution deviates from that found by the full search. The performance of Simulated Annealing [121] for both modulo scheduling and memory optimisation could also be evaluated. Greedy algorithms have also been proposed in existing work for modulo scheduling [30], array to memory allocation [65] and data reuse selection [65] and these could be evaluated against the optimal ILP solutions. There is also the potential to investigate new greedy algorithms.

7.2.5 Minimising Resource Usage or Power Consumption

The methodology presented in this work seeks to minimise execution time within a fixed set of resources, which is a common approach for high performance algorithms in areas such as high definition video processing and scientific computing. However, lower cost embedded systems require the design space to be approached from the opposite direction, minimising the resources or power consumed for a specified performance target. Future work could investigate the possibility of rearranging the order of the scheduling search

and modifying the ILP cost functions to meet this goal. In general, exploiting a higher degree of parallelism will yield higher performance but require more physical resources, higher memory bandwidth and more power. Higher memory bandwidth will require more on-chip and off-chip memory blocks and/or more data reuse structures.

Generally higher degrees of loop unroll and lower pipeline stages will require more resources, bandwidth and power, so a search for the lowest cost solution within a given execution time would start with estimates of the minimum unroll and highest pipeline stage length such that the lower bound execution time for these parameters meets to specified performance. As the search progresses the degree of unroll may need to be increased and the pipeline stage length decreased if the specified execution time is not met. How to best trade the values of the remaining scheduling parameters would need to be investigated. Cost functions for the memory optimisation ILP could be based around minimising a weighted sum of the memory resources consumed, with the weights based on the relative cost/value of each memory resource, or the power consumed by each resource.

In summary, this thesis has essentially proposed a methodology for the co-optimisation of the datapath and memory subsystem for FPGA implementations of loops in target algorithms. This methodology can be built upon and developed along the lines proposed to increase its applicability to real world algorithms and FPGA based platforms.

Appendix A

Schedule Length Derivation For Outer Loop Pipelining

Given a perfectly nested loop with L levels of nesting (level L is the innermost loop while 1 is the outermost), which is pipelined at some level, p, the number of cycles to execute the loop can be derived as follows:

- The operations in the loop will ultimately be modulo scheduled into S stages, each of which lasts T cycles. For each single iteration of the innermost loop (level L) the S stages will run sequentially, one after another. Hence, each iteration of loop level L will take $S \cdot T$ cycles to execute.

- For every iteration of loop level p, the iterations of all of the loop levels below p will execute sequentially. Hence the loop body will execute $\left(\prod_{i=0}^{p-1} N_i\right)$ times for every iteration at level p. N_i is the number of iterations at loop level i and N_0 is defined as 1 (for uniformity in the case when $p = 1$). Each iteration at level p will therefore take $Cycles_{p_it}$ to execute, where:

$$Cycles_{p_it} = S \cdot T \cdot \left(\prod_{i=0}^{p-1} N_i\right) \tag{A.1}$$

- With S stages in the pipeline it will be possible to overlap the execution of up to S iterations at level p (*i.e.* if iteration 1 is started at time t, up to $S - 1$ subsequent

iterations may be initiated before iteration 1 completes). A new loop iteration will be issued every II cycles, where II is the initiation interval of the pipeline. Depending on the values of II and $Cycles_{p_it}$ one of two cases will occur:

1. II is small compared to $Cycles_{p_it}$ so that $S \cdot II < Cycles_{p_it}$. In this case, if iteration 1 begins at time 0, S consecutive periods of II cycles can pass before before the end of iteration 1 (at time $Cycles_{p_it} - 1$). As a result, iteration $S+1$ cannot be initiated after S initiation intervals as expected since this would cause its execution to overlap with that of iteration 1. Thus more than S iterations would be overlapped causing more than one iteration to require the same stage at once. The start time of iteration $S + 1$ must therefore be delayed until the cycle after iteration 1 completes, *i.e.* until time $Cycles_{p_it}$. This pattern of inserting a delay after each block of S iterations must repeat through all the iterations at loop level p such that every block of S iterations will be 'pushed down' by a further $(Cycles_{p_it} - S \cdot II)$ clock cycles.
2. II is large so that $S \cdot II \geq Cycles_{p_it}$. In this case iteration 1 will have completed before iteration $S + 1$ is due to start. Hence there will be no resource conflicts and no 'push down' is necessary.

Note also that if $S \geq N_p$ then there will also be no need for a 'push down'. To accommodate both of these cases in one equation for execution time, the scheduling of the loop can be considered in the following way. Regardless of whether or not 'push down' is necessary the iterations at loop level p will be executed in blocks of S consecutive loop iterations. Each iteration within a given block is started II cycles after the previous iteration begins. The first iteration in each block starts $\max(Cycles_{p_it}, S \cdot II)$ cycles after the start of the first iteration in the previous block. The first block (containing iteration 1) starts at time 0. The first term in the max function deals with case 1 while the second deals with case 2.

- There will be $\lceil N_p/S \rceil$ blocks of S iterations at level p (the last block may contain fewer than S iterations). The first $\lceil N_p/S \rceil - 1$ blocks will each effectively contribute $\max(Cycles_{p_it}, S \cdot II)$ cycles to the execution time since this is how much they

each delay the subsequent blocks. The last block will require $Cycles_{p_it}$ to complete its first iteration. A $flush$ time will also be required to complete the remaining iterations in the final block. Hence the total time taken to execute one complete run of the pipelined loop level (*i.e.* one iteration of loop level $p+1$) can be computed as:

$$Cycles_{p_loop} = \big(\lceil N_p/S \rceil - 1\big).\max(Cycles_{p_it}, S \cdot II) + Cycles_{p_it} + flush \quad \text{(A.2)}$$

- The length of the $flush$ can be determined as follows. There will be $\big((N_p - 1) \bmod S\big) + 1$ iterations in the final block. The time for the first iteration is accounted for so only the remaining $\big((N_p - 1) \bmod S\big)$ iterations fall into the $flush$. If a new iteration is initiated every II cycles then an iteration will complete every II cycles. Hence the length of the $flush$ can be defined as:

$$flush = \big((N_p - 1) \bmod S\big) \cdot II \quad \text{(A.3)}$$

- The iterations of all loop levels above p will be executed sequentially. Each of these iterations will entail a full execution of loop level p. Hence, the execution time for the full nest will be:

$$Cycles_{nest} = Cycles_{p_loop} \cdot \left(\prod_{z=p+1}^{L+1} N_z \right) \quad \text{(A.4)}$$

  N_z is the number of iterations at loop level z, with N_{L+1} defined as 1 (for uniformity in the case when $p = L$).

Combining equations (A.1) to (A.4) gives the total cycles required to execute the loop nest, as described in (A.5), in terms of the loop limits and the scheduling parameters (S, T, II).

$$Cycles_{nest} = \Big(\prod_{z=p+1}^{L+1} N_z \Big) \cdot \Bigg((\lceil N_p/S \rceil - 1) \cdot \max \Big(S \cdot T \cdot \big(\prod_{i=0}^{p-1} N_i \big), S \cdot II \Big) + S \cdot T \cdot \big(\prod_{i=0}^{p-1} N_i \big) + \big((N_p - 1) \,\mathbf{mod}\, S \big) \cdot II \Bigg) \quad \text{(A.5)}$$

Appendix B

Derivations

B.1 Derivation 1

Here we show that condition (B.1) is equivalent to condition (B.2). *gcd* is the greatest common divisor of ii and S.

$$\forall_{1\leq k<S}, \forall_{0\leq j<k} \qquad (k \cdot ii \,\mathbf{mod}\, S) \neq (j \cdot ii \,\mathbf{mod}\, S) \tag{B.1}$$

$$\big(gcd = 1\big) \tag{B.2}$$

The term $(k \cdot ii \,\mathbf{mod}\, S)$ can be expressed as $(k \cdot ii - \alpha_1 \cdot S)$, where α_1 is some integer greater than or equal to zero. Likewise $(j \cdot ii \,\mathbf{mod}\, S)$ can be expressed as $(j \cdot ii - \alpha_2 \cdot S)$, where α_2 is another integer greater than or equal to zero. Condition (B.1) can therefore be rewritten as shown in (B.3).

$$\forall_{1\leq k<S}, \forall_{0\leq j<k} \qquad (k \cdot ii - \alpha_1 \cdot S) \neq (j \cdot ii - \alpha_2 \cdot S) \tag{B.3}$$

Condition (B.3) can be rearranged to give (B.4).

$$\forall_{1\leq k<S}, \forall_{0\leq j<k} \qquad (k - j) \cdot ii/S \neq \alpha_1 - \alpha 2 \tag{B.4}$$

The value of $\alpha_1 - \alpha_2$ will be an integer greater than or equal to zero. This means that we require the value of $(k-j) \cdot ii/S$ to be non-integer for condition (B.4) to be satisfied. Since

k varies in the range $1 \leq k < S$ and j varies in the range $0 \leq j < k$, the highest value that the term $k-j$ can take is $S-1$ and the lowest value is 1. This gives condition (B.5), where $\mathbf{Z}^+$ is the set of positive integers.

$$\forall_{1 \leq (k-j) < S} \qquad (k-j) \cdot ii/S \ni \mathbf{Z}^+ \tag{B.5}$$

If $(k-j) \cdot ii/S$ is an integer this implies that the value ii/S can be expressed as $(\alpha_1 - \alpha_2)/(k-j)$. But the maximum value of $(k-j)$ is $S-1$, so for ii/S to be expressed as $(\alpha_1 - \alpha_2)/(k-j)$ then ii and S must have a common factor greater than one. This means that the value of $(k-j) \cdot ii/S$ can only be an integer over the given ranges of k and j if the greatest common divisor of ii and S is greater than one. Hence, if the greatest common divisor or ii and S is one, as stated in condition (B.2), condition (B.1) will be satisfied.

B.2 Derivation 2

Here we show that condition (B.6) is equivalent to condition (B.7) at values of k and j which fail condition (B.1). gcd is again the greatest common divisor of ii and S.

$$\forall_{1 \leq k < S}, \forall_{0 \leq j < k}$$

$$\left(k \cdot ii \geq j \cdot ii + S \cdot \prod_{i=0}^{p-1} N_i\right) AND \left(k \cdot ii + S \cdot \prod_{i=0}^{p-1} N_i \leq j \cdot ii + S \cdot ii\right) \tag{B.6}$$

$$\left(\frac{ii}{gcd} \geq \prod_{i=0}^{p-1} N_i\right) AND \left(\frac{(gcd-1) \cdot ii}{gcd} + \prod_{i=0}^{p-1} N_i \leq ii\right) \tag{B.7}$$

The left and right hand expressions of the 'and' term in (B.6) correspond directly to the left and right hand expressions of the 'and' term in (B.7) so we will deal with each separately. The left hand expression in (B.6) can be rearranged to give (B.8).

$$\forall_{1 \leq k < S}, \forall_{0 \leq j < k} \qquad (k-j) \cdot ii/S \geq \prod_{i=0}^{p-1} N_i \tag{B.8}$$

We could simply evaluate condition (B.8) for the minimum value of $(k-j)$ which, given the ranges of k and j, is 1. However, we only require the condition to hold at values of

k and j which fail condition (B.1), so we are instead concerned with the minimum value of $(k-j)$ such that (B.1) is not met. As stated in section B.1, condition (B.1) will fail where ii/S can be expressed as $(\alpha_1 - \alpha_2)/(k-j)$. The minimum value of $(k-j)$ for which this will be the case is S/gcd since dividing both ii and S by gcd reduces ii/S to a fraction in its lowest terms. Putting this value of $(k-j)$ back into condition (B.8) we get condition (B.9), which can be simplified to give the left hand term in condition (B.7) as required.

$$(S/gcd) \cdot (ii/S) \geq \prod_{i=0}^{p-1} N_i \tag{B.9}$$

Similarly, rearranging the right hand term in (B.6) gives condition (B.10).

$$\forall_{1 \leq k < S}, \forall_{0 \leq j < k} \qquad (k-j) \cdot ii/S + \prod_{i=0}^{p-1} N_i \leq ii \tag{B.10}$$

This condition need only be evaluated at the maximum value of $(k-j)$ in the given range such that $(k-j) \cdot ii/S$ is an integer. Since the value of $(k-j)$ will always be less than S we can express it as $S - \alpha$, where α is an integer, and finding the minimum value of α will yield the maximum value of $(k-j)$. $(S-\alpha) \cdot ii/S$ is equivalent to $1 - \alpha \cdot ii/S$. Hence we require the minimum value of α such that $\alpha \cdot ii/S$ is an integer, and we have already shown that this is S/gcd. Hence the maximum value of $(k-j)$ that we need to consider is $S - S/gcd$. Putting this value into condition (B.10) we get condition (B.11), which can be simplified to give the right hand term in condition (B.7) as required.

$$(S - S/gcd) \cdot ii/S + \prod_{i=0}^{p-1} N_i \leq ii \tag{B.11}$$

Appendix C

Original and Transformed C Code for LU Decomposition and Minimum Residual Algorithms

The C code for the LU Decomposition kernel is given in Figure C.1. This is simply the *dgefa* subroutine described in Chapter 3 with the calls to the three functions (*idamax*, *dscal* and *daxpy*) replaced by the contents of each function. In Figure C.1 it can be seen that 3 separate loops are executed within a single iteration of the outermost loop, making the loop irregularly nested. To allow the methodologies developed in this work to be applied to the LU decomposition kernel it must be transformed so only a single loop is executed within the outermost loop. Fortunately the loops for the *idamax* and *dscal* operations have the same bounds as the loop for the *daxpy* operations. This allows us to combine these three loops within the middle loop level (iterated over by the 'j' variable). Two extra iterations are added to the middle loop. On the first additional iteration of the middle loop, the new innermost loop performs the *idamax* operations, while it performs the *dscal* operations on the second additional iteration. The *daxpy* operations then execute on the remaining middle loop iterations as in the original code. The additional 'if' statements ensure the correct operations are performed on each iteration. The final transformed C code is shown in Figure C.2. The transformation of the LU Decomposition code into

```
for (k = 0; k < 999; k++){
  max = fabs(A[k][k]);                  //start idamax
  for (i = k+1; i < 1000; i++)
    if (fabs(A[i][k]) > max){
      max = fabs(A[i][k]);
      piv = i;
    }                                   //end idamax
  ipvt[k] = piv;
  if (A[piv][k] != 0){
    temp = A[piv][k];
    A[piv][k] = A[k][k];
    A[k][k] = temp;
    t = -1/(A[k][k]);
    for (i = k+1; i < 1000; i++)        //start dscal
      A[i][k] = t*A[i][k];              //end dscal
    for (j = (k+1); j < 1000; j++){
      temp = A[piv][j];
      A[piv][j] = A[k][j];
      A[k][j] = temp;
      t = A[piv][j];
      for (i = (k+1); i < 1000)         //start daxpy
        A[i][j]= t*A[i][k] + A[i][j];   //end daxpy
    }
  }
}
ipvt[999] =999;
```

Figure C.1: Original C code for the LU decomposition kernel with irregular loop nesting.

a suitable form was performed manually. The steps used in the transformation process were intuitive to a person. Automation of such steps might be possible, but has not been investigated in this thesis. Similar transformations were applied the the Minimum Residual algorithm which also features irregular nesting. The original and transformed C codes are shown in Figures C.3 and C.4 respectively.

```
for (k = 0; k < 999; k++){
  for (j = (k-1); j < 1000; j++){
    if (j > k){
      t = A[piv][j];
      g = A[k][j];
    }
    else{
      t = A[piv][k];
      g = A[k][k];
      t2 = -1/t;
    }
    for (i = k; i < 1000; i++){
      if (j == (k-1)){              //start idamax
        if ((i==k) | (fabs(A[i][k]> max)){
          max = fabs(A[i][k]);
          piv = i;
          ipvt[k] = i;
          if (max == 0)
            do = 0;
          else
            do = 1;
        }
      }                             //end idamax
      if ((j == k) && (do == 1)){  //start dscal
        if (i == k){
          A[i][k] = t;
          col_str[i] = t;
        }
        else
          if (i == piv){
            A[i][k] = t2*g;
            col_str[i] = t2*g;
          }
          else{
            col_str[i] = t2*A[i][k];
            A[i][k] = t2*A[i][k];
          }
      }                             //end dscal
      if ((j > k) & (do == 1)){   //start daxpy
        if (i == k)
          A[i][j] = t;
        else
          if (i == piv)
            A[i][j] = g + t*col_str[i];
          else
            A[i][j] = A[i][j] + t*col_str[i];
      }                             //end daxpy
    }
  }
}
ipvt[999] = 999;
```

Figure C.2: Transformed C code for the LU decomposition kernel with only regular nesting.

```
while (norm_rmr/norm_r0 > required){
    for (i = 0; i < N; i++){
        v_old[i] = v[i];
        v[i] = v_hat[i]/beta;
    }
    for (i = 0; i < N; i++){
        for (j = 0; j < N; j++){
            if (i == 0)
                Av[j] = A[j][i]*v[i];
            else
                Av[j] = Av[j] + A[j][i]*v[i];
        }
    }
    for (i = 0; i < N; i++){
        if (i == 0)
            alpha = v[i]*Av[i];
        else
            alpha = alpha + v[i]*Av[i];
    }
    for (i = 0; i < N; i++){
        v_hat[i] = Av[i] - alpha*v[i] - beta*v_old[i];
    }
    beta_old = beta;
    for (i = 0; i < N; i++){
        if (i == 1)
            beta = v_hat[i]*v_hat[i];
        else
            beta = beta + v_hat[i]*v_hat[i];
    }
    beta = sqrt(beta);
    c_oold = c_old;
    c_old = c;
    s_oold = s_old;
    s_old = s;
    r1_hat = c_old*alpha - c_oold*s_old*beta_old;
    r1 = sqrt(r1_hat*r1_hat + beta*beta);
    r2 = s_old*alpha + c_oold*c_old*beta_old;
    r3 = s_oold*beta_old;
    c = r1_hat/r1;
    s = beta/r1;
    w_oold = w_old;
    w_old = w;
    w = (v - r3*w_oold - r2*w_old)/r1;
    xmr = xmr + c*eta*w;
    norm_xmr = norm_xmr*abs(s);
    eta = s*eta;
}
x = xmr;
```

Figure C.3: Original C code for the Minimum Residual algorithm with irregular loop nesting.

```
while (norm_rmr/norm_r0 > required){
    beta_old = beta;
    for (i = 0; i < N+1; i++){
        for (j = 0; j < N; j++){
            if (i == 0){
                v_old[j] = v[j];
                v[j] = v_hat[j]/beta;
            }
            if (i < N)
                if (j == 0)
                    temp = A[i][j]*v[j];
                else
                    temp = temp + A[i][j]*v[j];
            else{
                v_hat[j] = Av[j] - alpha*v[j] - beta*v_old[j];
                if (j == 0)
                    beta = v_hat[j]*v_hat[j];
                else
                    beta = beta + v_hat[j]*v_hat[j];
            }
        }
        if (i < N){
            Av[i] = temp;
            temp2 = temp*v[i];
            if (i == 0)
                alpha = temp2;
            else
                alpha = alpha + temp2;
        }
    }
    beta = sqrt(beta);
    c_oold = c_old;
    c_old = c;
    s_oold = s_old;
    s_old = s;
    r1_hat = c_old*alpha - c_oold*s_old*beta_old;
    r1 = sqrt(r1_hat*r1_hat + beta*beta);
    r2 = s_old*alpha + c_oold*c_old*beta_old;
    r3 = s_oold*beta_old;
    c = r1_hat/r1;
    s = beta/r1;
    w_oold = w_old;
    w_old = w;
    w = (v - r3*w_oold - r2*w_old)/r1;
    xmr = xmr + c*eta*w;
    norm_rmr = norm_rmr*abs(s);
    eta = s*eta;
}
x = xmr;
```

Figure C.4: Transformed C code for the Minimum Residual algorithm with only regular nesting. Only the two innermost loop levels are pipelined in this work as our methodology does not target while loops with undefined iteration counts. The initialisation of the variables prior to the first outer loop iteration has been omitted for brevity.

Appendix D

C Code for Benchmark Algorithms

The C code for the benchmarks used in this work (other than the LU decomposition and Minimum Residual algorithms which are supplied in Appendix C) is supplied for the readers reference in Figures D.1 to D.7.

```
for (i=0; i < width; i++){
   for (j=0; j < height; j++){
      for (k=0; k < 2; k++){
         for (m=0; m < 2; m++){
            diff_max = 0;
            if ((i+k-1<0)||(i+k-1>width-1)||(j+m-1<0)||(j+m-1>height-1))
               diff = abs(im_in_1[i][j]);
            else
               diff = abs(im_in_1[i][j] – im_in_2[i+k-1][j+m-1]);
            if (diff > diff_max)
               diff_max = diff;
         }
      }
      im_out[i][j] = diff_max;
   }
}
```

Figure D.1: C code for the image edge detection kernel.

```
max_store = ceil(window*window/2);
half = window/2;
for (i = 0; i < width; i++){
   for (j = 0; j < height; j++){
      count = 0;
      for (k = 0; k < window; k++){
         for (m = 0; m < window; m++){
            if ((i+k-half<0)||(i+k-half>width-1)||(j+m-half<0)||(j+m-half>height-1))
               val = im_in[i][j];
            else
               val = im_in[i+k][j+m];
            for (n = 0; n <= max_store; n++){
               if ((val < str[n])||(n == count)){
                  temp = str[n];
                  str[n] = val;
                  val = temp;
               }
            }
            if (count < max_store-1)
               count++;
         }
      }
      im_out[i][j] = str[max_store/2];
   }
}
```

Figure D.2: C code for the 2D median filter kernel.

```
half = window/2;
for (i = 0; i <  blocks_h; i++){
   for (j = 0; j < blocks_v; j++){
      sad_min = 100000000;
      for (k = 0; k < window; k++){
         for (m = 0; m < window; m++){
            sad = 0;
            for (n = 0; n < 16; n++){
               for (p = 0; p < 16; p++){
                  if ((i+k-half<0)||(i+k-half>width-1)||(j+m-half<0)||(j+m-half>height-1))
                     sad += abs(fr_in[16*i+n][16*j+p] - ref[16*i+n+k-half][16*j+p+m-half]);
                  else
                     sad += fr_in[16*i+n][16*j+p];
               }
            }
            if (sad < sad_min){
               h_store = k - half;
               v_store = m - half;
            }
         }
      }
      h_vect[i][j] = h_store;
      v_vect[i][j] = v_store;
   }
}
```

Figure D.3: C code for the motion estimation kernel.

```
for (i = 0; i < N; i++){
   for (j = 0; j < N; j++){
      temp = 0;
      for (k = 0; k < N; k++){
         temp += in_1[i][k]*in_2[k][j];
      }
      out[i][j] = temp;
   }
}
```

Figure D.4: C code for the matrix-matrix multiplication kernel. 'in_1', 'in_2' and 'out' are NxN floating point matrices.

```
while (sigma > target){
    for (i = 0; i < N; i++){
        sigma = 0;
        for (j = 0; j < N; j++){
            if (j != i)
               sigma += A[i][j]*x_prev[j];
        }
        sigma = b[i] - sigma/A[i][j];
        x_next[i] = x_prev[i] + omega*(sigma - x_prev[i]);

    }
    for (i = 0; i < N; i++)
        x_prev[i] = x_next[i];
}
```

Figure D.5: C code for the Successive Over Relaxation kernel. 'A' is an NxN floating point matrix and 'b', 'x_prev' and 'x_next' are N element floating point vectors.

```
for (j = 0; j < N; j++){
   for (k = 0; k < N; k++){
      qa = za[j+1][k]*zr[j][k] + za[j-1][k]*zb[j][k]
           + za[j][k+1]*zu[j][k] + za[j][k-1]*zv[j][k]
           + zz[j][k];
      za[j][k] += 0.175*(qa - za[j][k]);
   }
}
```

Figure D.6: C code for the 2D hydrodynamics kernel. 'za', 'zb', 'zr', 'zu', 'zv' and 'zz' are NxN floating point matrices.

```
n = samples*2;
j=1;
i=1;
while (i < n){
    if (j > i){
        tempr = data[j];
        data[j] = data[i];
        data[i] = tempr;
        tempr = data[j+1];
        data[j+1] = data[i+1];
        data[i+1] = tempr;
    }
    m=n/2;
    while ((m >= 2) && (j > m)){
        j = j - m;
        m = m/2;
    }
    j = j + m;
    i = i + 2;
}
mmax=2;
for (m = 0; m < log_2(samples); m++);
    istep = mmax*2;
    theta = isign*(2*pi/mmax);
    wtemp = sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi = sin(theta);
    wr=1.0;
    wi=0.0;
    m = 1;
    i = 1;
    j = 1 + mmax;
    for (k = 0; k < (samples/2); k++){
        tempr = wr*data[j] - wi*data[j+1];
        tempi = wr*data[j+1] + wi*data[j];
        data[j] = data[i] - tempr;
        data[j+1] = data[i+1] - tempi;
        data[i] = data[i] + tempr;
        data[i+1] = data[i+1] + tempi;
        i = i + istep;
        j = j + istep;
        if (i > n){
            wtemp = wr;
            wr=wtemp*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
            i = m+2;
            m = m+2;
            j = i + mmax;
        }
    }
    mmax=istep;
}
```

Figure D.7: C code for the Complex FFT kernel. The second of the two nested loops is targeted by our methodologies and implemented on the FPGA. The first of the two loops is assumed to be implemented on a host microprocessor. 'samples' is the number of samples in the FFT. 'data' is a fixed point array of twice the sample length used to store both the input and output data. 'log_2(samples)' computes the base 2 logarithm of 'samples'.

Bibliography

[1] *Altera EP300 Datasheet*, Altera Corporation, 1984.

[2] *Xilinx XC2064 Datasheet*, Xilinx Inc., 1985.

[3] *Stratix II Device Family Datasheet*, Altera Corporation, 2007. [Online]. Available: www.altera.com/literature/hb/stx2/stx2_sii5v1_01.pdf

[4] *Virtex-4 Family Overview*, Xilinx Inc., 2007. [Online]. Available: www.xilinx.com/support/documentation/data_sheets/ds112.pdf

[5] A. Khan, "Recent Developments in High-performance System-on-chip IC Design," in *Proc. Int. Conf. Integrated Circuit Design and Technology*, 2004, pp. 151–158.

[6] D. Chen, J. Cong, and P. Pan, "FPGA Design Automation: A Survey," *Foundations and Trends in Electronic Design Automation*, vol. 1, no. 3, pp. 195–330, 2006.

[7] R. Camposano, "From Behavior To Structure: High-Level Synthesis," *IEEE Design and Test of Computers*, vol. 7, no. 5, pp. 8–19, 1990.

[8] *Catapult C Datasheet*, Mentor Graphics, 2006. [Online]. Available: www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/ upload/Catapult_DS.pdf

[9] Z. Guo, B. Buyukkurt, and W. Najjar, "Optimized Generation of Data-path from C Codes for FPGAs," in *Proc. Design Automation and Test in Europe*, 2005, pp. 112–117.

[10] M. McFarland, A. Parker, and R. Camposano, "High-Level Synthesis of Digital Systems," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 301–318, 1990.

[11] K. Turkington, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Pipeline Exploration for Reconfigurable Targets," in *Proc. Workshop on Application Specific Processors*, 2007, pp. 11–18.

[12] K. Turkington, G. A. Constantinides, K. Masselos, and P. Cheung, "Outer Loop Pipelining for Application Specific Datapaths in FPGAs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 10, pp. 1268–1280, 2008.

[13] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. Gao, "Single-Dimension Software Pipelining for Multi-Dimensional Loops," in *Proc. IEEE Int. Symp. Code Generation and Optimization*, 2004, pp. 163–174.

[14] K. Turkington, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Co-optimisation of Datapath and Memory in Outer Loop Pipelining," in *Proc. IEEE Int. Conf. Field-Programmable Technology*, 2008, pp. 1–8.

[15] J. Allen and K. Kennedy, "Automatic Loop Interchange," in *Proc. ACM SIGPLAN Symposium on Compiler Construction*, 1984, pp. 233–246.

[16] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and Z. F., "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Trans. Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.

[17] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK : A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations," in *Proc. IEEE Int. Conf. VLSI Design*, 2003, pp. 461–466.

[18] M. Wolfe, *High Performance Compilers for Parallel Computing.* AddisonWesley, 1995.

[19] J. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of Control To Data Dependence," in *Proc. ACM Symp. Principles of Programming Languages*, 1983, pp. 177–189.

[20] W. Pugh, "The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis," in *Proc. ACM/IEEE Conf. Supercomputing*, 1991, pp. 4–13.

[21] C. Fan and F. Chang, "A Novel Approach for Solving Diophantine Equations," *IEEE Trans. Circuits and Systems*, vol. 37, no. 11, pp. 1455–1457, 1990.

[22] V. Maslov, "Global Value Propagation Through Value Flow Graph and its Use in Dependence Analysis," University of Maryland at College Park, Tech. Rep. CS-TR-3310, 1994.

[23] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and S. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler," *IEEE Computer*, vol. 29, pp. 84–89, 1996.

[24] D. Bacon, S. Graham, and O. Sharp, "Compiler Transformations for High-Performance Computing," *ACM Computing Surveys*, vol. 26, no. 4, pp. 345–420, 1994.

[25] D. Kulkarni and M. Stumm, *Loop & Data Transformations: A Tutorial*, Technical Report CSRI-337, Computer Systems Research Institute, University of Toronto, 1993.

[26] U. Banerjee, *Loop Parallelization.* Kluwer Academic Publishers, 1994.

[27] M. Weinhardt and W. Luk, "Pipeline Vectorization for Reconfigurable Systems," in *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 1999, pp. 52–62.

[28] R. Jones and V. Allan, "Software Pipelining: A Comparison and Improvement," in *Proc. Symp. Microprogramming and Microarchitecture*, 1990, pp. 46–56.

[29] *Intel Core i7 Processor Extreme Edition and Intel Core i7 Processor Datasheet*, Intel Corporation, 2009. [Online]. Available: http://download.intel.com/design/processor/datashts/320834.pdf

[30] V. Allan, R. Jones, R. Lee, and S. Allan, "Software Pipelining," *ACM Proc. Computing Surveys*, vol. 27, no. 3, pp. 367–432, 1995.

[31] B. Su and J. Wang, "Loop-carried Dependence and the General URPR Software Pipelining Approach," in *Proc. Int. Conf. System Sciences*, 1991, pp. 366–372.

[32] S. Krishnamurthy, "Ultra-pipelining: An Efficient Software Pipelining Algorithm," in *Proc. Compcon*, 1993, pp. 586–591.

[33] A. Aiken and A. Nicolau, "Perfect Pipelining: A New Loop Parallelization Technique," in *Proc. European Symp. Programming*, 1988, pp. 221–235.

[34] R. Ramakrishna, "Iterative Modulo Scheduling," Hewlett Packard Laboratories, Tech. Rep. HPL-94-115 , 1995. [Online]. Available: http://www.hpl.hp.com/techreports/94/HPL-94-115.html, accessed Nov. 2006

[35] S. Gerez, S. Heemstra De Groot, and O. Herrmann, "A Polynomial-time Algorithm for the Computation of the Iteration-period Bound In Recursive Data-flow Graphs," *IEEE Trans. on Circuits and Systems-I: Fundamental Theory and Applications*, vol. 39, no. 1, pp. 49–52, 1992.

[36] M. Weinhardt and W. Luk, "Memory Access Optimization for Reconfigurable Systems," *IEE Proc. Computers and Digital Techniques*, vol. 148, no. 3, pp. 105–112, 2001.

[37] J. Wang and B. Su, "Software Pipelining of Nested Loops for Real-time DSP Applications," in *Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing*, 1998, pp. 3065–3068.

[38] D. Petkov, R. Harr, and S. Amarasinghe, "Efficient Pipelining of Nested Loops: Unroll-and-Squash," in *Proc Int. Symp. Parallel and Distributed Processing*, 2002, pp. 19–24.

[39] T. Yu, Z. Tang, C. Zhang, and J. Luo, "Control Mechanism for Software Pipelining on Nested Loop," in *Proc. Advances in Parallel and Distributed Computing*, 1997, pp. 345–350.

[40] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. Gao, "Single-Dimension Software Pipelining for Multi-Dimensional Loops," University of Delaware, Tech. Rep. CAPSL Technical Memo 049, 2005.

[41] J. Ramanujam, "Optimal Software Pipelining of Nested Loops," in *Proc. Int. Parallel Processing Symp.*, 1994, pp. 335–342.

[42] L. Lamport, "The Parallel Execution of DO Loops," *Communications of the ACM*, vol. 17, pp. 83–93, 1974.

[43] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, and F. Catthoor, "A Practical Dynamic Single Assignment Transformation," *ACM Trans. Design Automation for Electronic Systems*, vol. 12, 2007.

[44] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Dynamically Increasing the Scope of Code Motions During the High-level Synthesis of Digital Circuits," in *Proc. IEEE Int. Conf. Computers and Digital Techniques*, 2003, pp. 330–337.

[45] *Quartus II Handbook Version 9.0 Volume 4: SOPC Builder*, Altera Corporation, 2008. [Online]. Available: www.altera.com/literature/ hb/qts/qts_qii5v4.pdf

[46] M. Wolf and M. Lam, "A Data Locality Optimizing Algorithm," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1991, pp. 30–44.

[47] K. McKinley, S. Carr, and C. Tseng, "Improving Data Locality with Loop Transformations," *ACM Trans. Programing Languages and Systems*, vol. 18, pp. 424–453, 1996.

[48] P. Panda, F. Catthoor, N. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. Kjeldsberg, "Data and Memory Optimisation Techniques for Embedded Systems," *ACM Trans. Design and Automation of Electronic Systems*, vol. 6, no. 2, pp. 149–206, 2001.

[49] K. Masselos, F. Catthoor, C. Goutis, and H. DeMan, "System-level Power Optimizing Data-flow Transformations for Multimedia Applications Realized on Pro-

grammable Multimedia Processors," in *Proc. IEEE Int. Conf. Electronics, Circuits and Systems*, vol. 3, 1999, pp. 1733–1736.

[50] F. Franssen, L. Nachtergaele, H. Samsom, F. Catthoor, and H. De Man, "Control Flow Optimization for Fast System Simulation and Storage Minimization," in *Proc. European Design and Test Conference*, 1994, pp. 20–24.

[51] E. De Greef, F. Catthoor, and H. Man, "Memory Organisation for Video Algorithms on Programmable Signal Processors," in *Proc. IEEE Int. Conf. Computer Design*, 1995, pp. 552–557.

[52] D. Kulkarni, "Transformations for Improving Data Access Locality in Non-perfectly Nested Loops," in *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, 1998, pp. 314–321.

[53] *Stratix III Device Family Datasheet*, Altera Corporation, 2009. [Online]. Available: www.altera.com/literature/hb/stx3/stx3_siii52001.pdf

[54] S. Ang, G. Constantinides, P. Cheung, and W. Luk, "A Flexible Multiport Caching Scheme for Reconfigurable Platforms," *Reconfigurable Computing: Architectures and Applications*, pp. 205–216, 2006.

[55] O. Temam, C. Fricker, and W. Jalby, "Cache interference Phenomena," in *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1994, pp. 261–271.

[56] C. Aliagas, M. Valero, and C. Nord, "A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality," in *Proc. Int. Conf. Supercomputing*, 1995, pp. 338–347.

[57] W. Tang, A. Veidenbaum, A. Nicolau, and R. Gupta, "Conflict Miss Elimination by Time-stride Prefetch," University of California, Irvine, Tech. Rep., 2003.

[58] A. Shrivastava, I. Issenin, and N. Dutt, "Compilation Techniques for Energy Reduction in Horizontally Partitioned Cache Architectures," in *Proc. ACM Int. Conf. Compilers, Architectures and Synthesis for Embedded Systems*, 2005, pp. 90–96.

[59] P. Petrov and A. Orailoglu, "Performance and Power Effectiveness in Embedded Processors-customizable Partitioned Caches," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, pp. 1309–1318, 2001.

[60] J. Diguet, S. Wuytack, F. Catthoor, and H. De Man, "Formalized Methodology for Data Reuse Exploration in Hierarchical Memory Mappings," in *Proc. Int. Symp. Low Power Electronics and Design*, 1997, pp. 30–35.

[61] S. Wuytack, J. Diguet, F. Catthoor, and H. De Man, "Formalized Methodology for Data Reuse: Exploration for Low-Power Hierarchical Memory Mappings," *IEEE Trans. Very Large Scale Integration Systems*, vol. 6, no. 4, pp. 529–537, 1998.

[62] C. Kulkarni, F. Catthoor, and H. De Man, "Code Transformations for Low Power Caching in Embedded Multimedia Processors," in *Proc Int. Parallel Processing Symp. and Symp. Parallel and Distributed Processing*, 1998, pp. 292–297.

[63] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology.* Kluwer Academic Publishers, 1998.

[64] M. Weinhardt and W. Luk, "Pipeline Vectorization," *IEEE Trans Computer-aided Design of Integrated Circuits and Systems*, vol. 20, no. 2, pp. 234–248, 2001.

[65] P. Diniz and J. Park, "Automatic Synthesis of Data Storage and Control Structures for FPGA-based Computing Engines," in *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 2000, pp. 91–100.

[66] Z. Guo, B. Buyukkurt, and W. Najjar, "Input Data Reuse in Compiling Window Operations onto Reconfigurable Hardware," in *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2004, pp. 249–256.

[67] Q. Liu, K. Masselos, and G. Constantinides, "Data Reuse Exploration for FPGA Based Platforms Applied to the Full Search Motion Estimation Algorithm," in *Proc. IEEE Int. Conf. Field Programmable Logic and Applications*, 2006, pp. 389–394.

[68] Q. Liu, G. Constantinides, K. Masselos, and P. Cheung, "Automatic On-chip Memory Minimization for Data Reuse," in *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 2007, pp. 251–260.

[69] Q. Liu, G. Constantinides, K. Masselos, and P. Cheung, "Data Reuse Exploration Under Area Constraints for Low Power Reconfigurable Systems," in *Proc. Workshop on Application Specific Processors*, 2008.

[70] Q. Liu, G. Constantinides, K. Masselos, and P. Cheung, "Combining Data Reuse Exploitation with Data-level Parallelization for FPGA Targeted Hardware Compilation: A Geometric Programming Framework," in *Proc. IEEE Int. Conf. Field Programmable Logic and Applications*, 2008, pp. 179–184.

[71] P. Marwedel, "The MIMOLA System: Detailed Description of the System Software," in *Proc. IEEE/ACM Conf. Design Automation*, 1979, pp. 59–63.

[72] H. De Man, F. Catthoor, G. Goossens, J. Van Meerbergen, J. Rabaey, and J. Huisken, "Architecture-driven Synthesis Techniques for Mapping Digital Signal Processing Algorithms into Silicon," *Proc. IEEE*, vol. 78, no. 2, pp. 319–335, 1990.

[73] L. Ramachandran, D. Gajski, and V. Chaiyakul, "An Algorithm for Array Variable Clustering," in *Proc. European Design Test Conf.*, 1994, pp. 262–266.

[74] H. Schmit and D. Thomas, "Synthesis of Application-Specific Memory Designs," *IEEE Trans. Very Large Scale Integration Systems*, vol. 5, no. 1, pp. 101–111, 1997.

[75] H. Schmit and D. Thomas, "Array Mapping in Behavioral Synthesis," in *Proc. Int. Symp. System Synthesis*, 1995, pp. 90–95.

[76] D. Karchmer and J. Rose, "Definition and Solution of the Memory Packing Problem," in *Proc. Int. Conf. Computer Aided Design*, 1994, pp. 53–58.

[77] E. Ochotta and T. Mukherjee, "Programmers Guide to the Reconfigurable Simulated Annealing Library (anneal)," Carnegie Mellon Univ., Tech. Rep. CMUCAD-94-35, 1994.

[78] M. Gokhale and J. Stone, "Automatic Allocation of Arrays to Memories in FPGA Processors With Multiple Memory Banks," in *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 1999, pp. 63–69.

[79] M. Gokhale and J. Stone, "NAPA C: Compiling for a Hybrid RISC/FPGA Architecture," in *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 1998, pp. 126–135.

[80] H. Williams, *Model Building in Mathematical Programming (Fourth Edition)*. Wiley and Sons, 1998.

[81] *RCHTX–VX4 Datasheet*, Celoxica Limited, 2006. [Online]. Available: www.hypertransport.org/docs/tech/rchtx_datasheet_screen.pdf

[82] N. Baradaran and P. Diniz, "Memory Parallelism using Custom Array Mapping to Heterogeneous Storage Structures," in *Proc. Conf. Design, Automation and Test in Europe*, 2005, pp. 6–11.

[83] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal, "Memory Bank Disambiguation using Modulo Unrolling for Raw Machines," in *Proc. IEEE Int. Symp. Computer Archicture*, 1999, pp. 4–15.

[84] B. So, M. Hall, and H. Ziegler, "Custom Data Layout for Memory Parallelism," in *Proc. Int. Symp. Code Generation and Optimization*, 2004, pp. 291–302.

[85] W. Gong, G. Wang, and R. Kastner, "Storage Assignment during High-level Synthesis for Configurable Architectures," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 2005, pp. 3–6.

[86] N. Baradaran and P. Diniz, "A Register Allocation Algorithm in the Presence of Scalar Replacement for Fine-Grain Configurable Architectures," in *Proc. Conf. Field Programmable Logic and Applications*, 2006, pp. 383–388.

[87] A. Yamada, K. Nishida, R. Sakurai, A. Kay, T. Nomura, and T. Kambe, "Hardware Synthesis with the Bach System," in *Proc IEEE Int Symp. Circuits and Systems*, 1999, pp. 366–369.

[88] M. Gokhale, J. Stone, and J. Arnold, "Stream-Oriented FPGA Computing in the Streams-C High Level Language," in *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 2000, pp. 49–56.

[89] J. Coutinho, J. Jiang, and W. Luk, "Interleaving Behavioral and Cycle-Accurate Descriptions for Reconfigurable Hardware Compilation," in *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 2005, pp. 245–254.

[90] J. Tripp, P. Jackson, and B. Hutchings, "Sea Cucumber: A Synthesizing Compiler for FPGAs," in *Proc. Conf. Field Programmable Logic and Applications*, 2002, pp. 875–886.

[91] D. Soderman and Y. Panchul, "Implementing C Algorithms in Reconfigurable Hardware Using C2Verilog," in *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 1998, pp. 339–342.

[92] T. Maruyama and T. Hoshino, "A C to HDL Compiler for Pipeline Processing on FPGAs," in *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 2000, pp. 101–110.

[93] O. Mencer, D. Pearce, L. Howes, and W. Luk, "Design Space Exploration with A Stream Compiler," in *Proc. IEEE Int. Conf. Field-Programmable Technology*, 2003, pp. 270–277.

[94] M. Gokhale, J. Kaba, A. Marks, and J. Kim, "A Malleable Architecture Generator for FPGA Computing," *Proc. SPIE*, vol. 2914, pp. 208–217, 1996.

[95] G. Snider, B. Shackleford, and R. Carter, "Attacking the Semantic Gap Between Application Programming Languages and Configurable Hardware," in *ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays*, 2001, pp. 115–124.

[96] *Synplify Pro Datasheet*, Synplicity, 2008. [Online]. Available: www.synplicity.com/literature/pdf/syn_pro_ds.pdf

[97] *Leonardo Spectrum Datasheet*, Mentor Graphics Corporation, 2003. [Online]. Available: www.mentor.com/products/fpga/synthesis /leonardo_spectrum/upload/datasheet.pdf

[98] *DK Design Suite Datasheet*, Agility Design Solutions Inc., 2008. [Online]. Available: www.agilityds.com/literature/dk_datasheet_01000_hq_screen.pdf

[99] *Xilinx ISE 9.2i Software Manuals and Help - PDF Collection*, Xilinx Inc., 2007. [Online]. Available: www.xilinx.com/itp/xilinx92/books/manuals.pdf

[100] *Quartus II Handbook*, Altera Corporation, 2009. [Online]. Available: www.altera.com/literature/hb/qts/quartusii_handbook.pdf

[101] T. Callahan and J. Wawrzynek, "Adapting Software Pipelining for Reconfigurable Computing," in *ACM Int. Conf. Compilers, Architecture and Synthesis for Embedded Systems*, 2000, pp. 57–64.

[102] D. Galloway, "The Transmogrifier C Hardware Description Language and Compiler for FPGAs," in *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 1995, pp. 136–144.

[103] C. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.

[104] G. De Micheli, D. Ku, F. Mailhot, and T. Truong, "The Olympus Synthesis System," *IEEE Trans Design and Test of Computers*, vol. 7, no. 5, pp. 37–53, 1990.

[105] [Online]. Available: www.synopsys.com/products/logic/design_compiler.html, accessed Nov. 2006

[106] E. Lee and D. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[107] *DSP Builder Reference Manual*, Altera Cororation, 2009. [Online]. Available: www.altera.com/literature/manual/mnl_dsp_builder.pdf

[108] *System Generator for DSP User Guide*, Xilinx Inc., 2008. [Online]. Available: www.xilinx.com/support/documentation/sw_manuals/sysgen_user.pdf

[109] *Synplify DSP Datasheet*, Synplicity, 2008. [Online]. Available: www.synplicity.com/literature/pdf/syn_dsp_ds.pdf

[110] [Online]. Available: www.celoxica.com/techlib/files/CEL-W0410251JJ4-60.pdf, accessed Nov. 2006

[111] *Cynthesizer Datasheet*, Forte Design Systems, 2008. [Online]. Available: www.forteds.com/products/cynthesizer_datasheet_2008.pdf

[112] [Online]. Available: www.impulsec.com, accessed Nov. 2006

[113] *Pico Express Datasheet*, Synfora Inc., 2008. [Online]. Available: www.synfora.com/products/files/PicoExpress_datasheet_08.pdf

[114] "System C Website," Open System C Initiative. [Online]. Available: http://www.systemc.org/home/

[115] *Agility User Manual*, Agility Design Solutions Inc., 2007. [Online]. Available: http://agilityds.com/literature/Agility_User_Manual.pdf

[116] *AccelDSP Sythesis Tool User Guide*, Xilinx Inc., 2008. [Online]. Available: www.xilinx.com/support/documentation/sw_manuals/acceldsp_user.pdf

[117] *Nios II C2H Compiler User Guide*, Altera Corporation, 2008. [Online]. Available: www.altera.com/literature/ug/ug_nios2_c2h_compiler.pdf?GSA_pos=1&WT.oss_r=1&WT.oss=c2h user guide

[118] *Nios II Processor Reference Handbook*, Altera Corporation, 2009. [Online]. Available: www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf

[119] *Nios II Software Developer's Handbook*, Altera Corporation, 2009. [Online]. Available: www.altera.com/literature/hb/nios2/n2sw_nii5v2.pdf

[120] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[121] S. Kirkpatrick, J. Gellat, and M. Vecchi, "Optimization by Simulated Annealing," *Science*, pp. 671–680, 1983.

[122] R. Garfinkel and G. Nemhauser, *Integer Programming*. Wiley and Sons, 1972.

[123] *ILOG CPLEX Product Datasheet*, ILOG, 2007. [Online]. Available: www.ilog.com/download/docs/DS-CPLEX.pdf

[124] W. Ligon, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. Underwood, "A Re-evaluation of the Practicality of Floating Point on FPGAs," in *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*, 1998, pp. 206–215.

[125] J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: Past, Present and Future," *Proc. Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, 2003.

[126] *Handel-C Language Reference Manual*, Agility Design Solutions Inc., 2008. [Online]. Available: agilityds.com/literature/HandelC_Language_Reference_Manual.pdf

[127] *ModelSim Datasheet*, Mentor Graphics, 2008. [Online]. Available: www.mentor.com/products/fv/modelsim/upload/datasheet.pdf

[128] "Top 500 Supercomputing Sites Website," Top500.org. [Online]. Available: http://www.top500.org/

[129] R. Whaley and A. Petitet, "Minimizing Development and Maintainance Costs In Supporting Persistently Optimised BLAS," *Proc. Software: Practice and Experience*, vol. 35, no. 2, pp. 101–121, 2005.

[130] "Agility RC3000 Datasheet," 2008.

[131] "Agility RC2000 Datasheet," 2008. [Online]. Available: www.agilityds.com/literature/rc2000_datasheet_01000_hq_screen.pdf

[132] *Platform Developer's Kit: Pipelined Floating-point Library Manual*, Agility Design Solutions Inc.(formerly Celoxica), 2005. [Online]. Available: http://babbage.cs.qc.edu/courses/cs345/Manuals/Floating%20Point%20Library%20Manual.pdf

[133] G. Constantinides and G. Woeginger, " The Complexity of Multiple Wordlength Assignment," *Applied Mathematics Letters*, vol. 15, no. 2, pp. 137–140, 2002.

[134] *Nios Development Board User Guide*, Altera Corporation, 2004. [Online]. Available: www.altera.com/literature/manual /mnl_nios2_board_stratix_1s10.pdf

[135] J. Cong, M. Romesis, and M. Xie, "Optimality and Stability Study of Timing-driven Placement Algorithms," in *Proc IEEE Int. Conf. Computer Aided Design*, 2003, pp. 472–478.

[136] K. Underwood and K. Hemmert, "Closing the gap: CPU and FPGA trends in sustainable floating point BLAS performance," in *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 2004, pp. 219–228.

[137] G. Govindu, S. Choi, V. Prasanna, V. Daga, S. Gangadharpalli, and V. Sridhar, "A High-Performance and Energy-efficient Architecture for Floating-point based LU Decomposition on FPGAs," in *Proc. IEEE. Int. Symp. Parallel and Distributed Processing*, 2004, pp. 149–156.

[138] L. Zhuo and V. Prasanna, "Scalable Hybrid Designs for Linear Algebra on Reconfigurable Computing Systems," in *Proc. IEEE. Int. Conf. Parallel and Distributed Systems*, 2006, pp. 9–16.

[139] *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, Xilinx, 2007.

[140] W. Pugh, "Definitions of Dependence Distance," *ACM Letters on Programming Languages and Systems*, vol. 1, no. 3, pp. 261–265, 1992.

[141] *Stratix II Device Handbook*, Altera Corp., 2007. [Online]. Available: www.altera.com/literature/hb/stx2/stratix2_handbook.pdf

[142] F. McMahon, "The Livermore Fortran Kernels Test of the Numerical Performance Range," *Performance Evaluation of Supercomputers*, pp. 143–186, 1988.

[143] A. Hadjidimos, "Successive Over Relaxation (SOR) and related methods," *Journal of Computational and Applied Mathematics*, vol. 123, no. 1-2, pp. 177–199, 2000.

[144] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, 1994.

[145] S. Grossman, *Elementary Linear Algebra (Fourth Edition)*. Saunders College Publishing, 1994.

[146] Y. Liao and C. Wong, "An Algorithm to Compact a VLSI Symbolic Layout with Mixed Constraints," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 2, no. 2, pp. 63–69, 1983.

[147] *Stratix IV GX FPGA Development Board Reference Manual*, Altera Corp., 2009. [Online]. Available: http://www.altera.com/literature/manual/rm_sivgx_fpga_dev_board.pdf

[148] *Virtex-6 FPGA ML605 Evaluation Kit Product Brief*, Xilinx, 2009.

[149] *SODIMM Part Catalogue*, Micron Technology Inc., 2009. [Online]. Available: http://www.micron.com/products/modules/sodimm/partlist?Select=Select