Co-optimisation of Datapath and Memory in Outer Loop Pipelining

Kieron Turkington,
George A. Constantinides, Peter Y.K. Cheung
Department of Electrical and Electronic Engineering,
Imperial College London,
London SW7 2BT, UK.
Email {kjt01, g.constantinides, p.cheung}@imperial.ac.uk

Konstantinos Masselos
Department of Computer Science
and Technology,
University of Peloponnese,
Tripolis 22100, Greece.
Email kmas@uop.gr

Abstract

When targeting algorithms to FPGAs both the array to memory assignment and the selection of data reuse structures should be considered to maximise performance. In this work we present an Integer Linear Programming formulation for the combined problem of array to memory assignment and data reuse selection. We include a number of cost functions to minimise during memory optimisation and show how these optimisations can be integrated into a loop pipelining framework to iteratively update the memory subsystem during scheduling. By co-optimising the datapath and memory subsystem we are able to produce near optimal (fastest) solutions, with an upper bound on the distance from the optimal. Our results show an average speedup of up to 4x over a non-optimised memory subsystem when integrated into an existing outer loop pipelining framework.

1 Introduction

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 we 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, 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 [1]–[6]. 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 [1] and [3] integrates array to memory assignment with scheduling, data reuse decisions are not included. [2] and [4] 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. [5] presents a scheme combining array to memory allocation and data reuse, but this work targets low power, relatively sequential DSP systems and is not closely linked to fine grain scheduling. [6] produces autonomous buffers for 'windows' of data, but does not support other forms of reuse. The contribution of this work is to provide an Integer Linear Programming formulation to simultaneously place arrays in the available memories and select the appropriate data reuse options from the available set. We offer three cost functions to be used when optimising the memory and show how these can be integrated into an existing loop pipelining framework to co-optimise the schedule and the supporting memory subsystem.

The remainder of the paper is divided into seven further sections. In Section 2 we give a basic description of the outer loop pipelining framework into which the memory optimisation approach is integrated. Sections 3 and 4 describe our ILP formulation for memory optimisation, while Sections 5 and 6 show how the memory subsystem is iteratively updated during scheduling. Section 7 presents our results and we conclude in Section 8.

2 Background

We have previously presented an approach for extending an existing outer loop pipelining approach [7], originally developed for VLIW processors, to FPGAs [8]. This approach works by overlapping the executions of iterations at

Algorithm 1 : 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
```

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), outer loop pipelining presents a more complicated solution space where a number of scheduling parameters may be traded. In [8] we presented a search scheme 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 pseudo code for this search is given in Algorithm 1. 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 given T.

In [8] it is assumed that an array to memory map is supplied as an input, and there is no data reuse. Ideally we 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, all of the scheduling parameters can be affected by the properties of the memory subsystem, so as the scheduling search progresses the desired properties from the memory subsystem will change. For instance, at first we require the memory subsystem such that T is minimised at the cost of all else, but as the search progresses the value of T may be increased in favour of minimising S or II. Hence we propose that the memory subsystem is continually updated during scheduling. The commented lines in Algorithm 1 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.

In this work we are looking to find the fastest possible pipelined implementation for a nested loop. However, there are currently a number of restrictions to our methodology. Firstly, we have not yet automated any loop unrolling strategies, though the user may unroll the loop manually and input this unrolled version. We also do not consider array partitioning or duplication, though these may again be specified manually by the user. We currently also do not support data reuse for imperfectly nested operations. Removing these restrictions remains the basis for future work. This work targets SRAM memories as their constant, relatively short latencies are well suited to our underlying pipelining framework. Currently we can only include DRAM or SDRAM memories by assuming the worst case access latency in all cases, which is far from optimal.

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 [2, 3]. 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 then assign each logical memory to one of the types of memory resource, rather than any particular instance of a given type.

We have formulated this problem using Integer Linear Programming (ILP) [9]. 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) [10]. Each t_k is therefore considered to have con_k configurations. The arrays are mapped to a set of N logical memories, $\mathbf{L} = \{L_1, ..., L_N\}$, 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, we introduce a set of integer variables, $banks_{jkm}$, 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 (1) to (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 we allow multiple elements of an array to be packed into a single (larger) memory word if the array is only ever read during the loop.

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

$$\forall l_j \in \mathbf{L}, \qquad \sum_{k=1}^{M} \sum_{m=1}^{con_k} b_{jkm} \; \leq \; 1 \tag{2}$$

$$\forall l_j \in \mathbf{L}, \qquad N \cdot \sum_{k=1}^{M} \sum_{m=1}^{con_k} b_{jkm} \; \geq \; \sum_{i=1}^{N} d_{ij} \tag{3}$$

$$\forall t_k \in \mathbf{T}, \qquad \sum_{j=1}^{N} \sum_{m=1}^{con_k} banks_{jkm} \; \leq \; num_k \tag{4}$$

$$\begin{aligned} \forall l_j \in \mathbf{L}, \forall t_k \in \mathbf{T}, \forall m \in (1 : con_k), \\ banks_{jkm} \cdot W_{Mkm} \; \geq \; \sum_{i=1}^{N} (d_{ij} \cdot W_{Aikm}) \\ + \quad (b_{jkm} - 1) \cdot \sum_{i=1}^{N} W_{Aikm} \end{aligned} \tag{5}$$

Equation (1) constrains each array to be assigned to single logical memory. The constraints represented by (3) ensure that any logical memory with one or more arrays assigned to it is assigned to at least one memory resource type, while (2) ensures that each logical memory is assigned to only a single resource type. (4) ensures that no more resources are used than are available. The constraints represented by (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) 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) 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) will be equal to the maximum possible value of term one. As a result the RHS of (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.

4 Integrating Data Reuse with Array to Memory Placement

In this work we attempt to reuse array data through buffers (scratch pad memories) [6, 11] and FIFOs [2, 4] as this can reduce the number of accesses to off-chip memories. Data reuse will compete with the array to memory placement for the available memory resources, while the reuse options selected will alter the number of accesses to some arrays and affect how they should be placed in memories to maximise parallel access. Furthermore, the FIFOs and buffers may need to be filled with data at the start of each loop level. This will add extra imperfectly nested operations to the loop if the data reuse structure is selected for implementation, and the time taken for this fill process must be traded against the potential scheduling gains.

To integrate the process of selecting the optimum set of data reuse options into the array to memory placement formulation, the ILP presented in Section 3 is updated as follows: We are given a set of P buffers, $\mathbf{E} = \{e_1, ..., e_P\}$, and a set of Q FIFOs, $\mathbf{F} = \{f_1, ..., f_Q\}$. 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 3 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 logical memories. A further set of binary variables, y_{nkm}, is created to assign the FIFOs to memory resource types. y_{nkm} is one if FIFO f_n is assigned to configuration m of type t_k and zero otherwise. The constraints represented by (6) to (9) must be added to those in Section 3, and equation (4) must be replaced by equation (10) to ensure that the combined usage of memory resources across the arrays and data reuse options does not exceed those available. In equation (8), $\mathbf{R}$ represents the set of read operations at the innermost level in the target loop. $\mathbf{Z_i}$ represents the set of reuse options that serve read operation r_i. In equation (9), 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. B_{nkm} in (10) is a constant

representing the banks of type t_k used if FIFO f_n is targeted to configuration m of that type.

$$\forall l_j \in \mathbf{L_B}, \qquad \sum_{k=1}^{M} \sum_{m=1}^{con_k} b_{jkm} \leq 1 \tag{6}$$

$$\forall l_j \in \mathbf{L_B}, \qquad P \cdot \sum_{k=1}^{M} \sum_{m=1}^{con_k} b_{jkm} \geq \sum_{i=1}^{P} x_{ij} \tag{7}$$

$$\forall r_i \in \mathbf{R}, \sum_{f_n \in \mathbf{Z_i}} \sum_{k=1}^{M} \sum_{m=1}^{con_k} y_{nkm} + \sum_{e_k \in \mathbf{Z_i}} \sum_{j=N+1}^{N+P} x_{kj} \leq 1 \tag{8}$$

$$\forall l_j \in \mathbf{L_B}, \forall t_k \in \mathbf{T}, \forall m \in (1 : con_k), \quad banks_{jkm} \cdot W_{Mkm} \geq \sum_{i=1}^{P} (x_{ij} \cdot W_{Bikm}) + (b_{jkm} - 1) \cdot \sum_{i=1}^{P} W_{Bikm} \tag{9}$$

$$\forall t_k \in \mathbf{T}, \qquad \sum_{j=1}^{N+P} \sum_{m=1}^{con_k} banks_{jkm} + \sum_{n=1}^{Q} \sum_{m=1}^{con_k} (B_{nkm} \cdot y_{nkm}) \leq num_k \tag{10}$$

The constraints represented by equations (6), (7) and (9) constrain the buffer logical memories just as equations (2), (3) and (5) constrain the array logical memories. Equation (8) 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 or buffer can be assigned to (at most) one memory resource type or logical memory respectively.

The cost functions employed with this formulation during the scheduling search are discussed in Section 5, 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. 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 equation (11). $\mathbf{Z_i}$ in equation (11) 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 equation (11) 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 equation (11) 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 is the sum of the Acc_{ij} variables for all arrays that may be assigned to it.

$$\forall a_i \in \mathbf{A}, \forall l_j \in \mathbf{L}, \qquad Acc_{ij} \geq d_{ij} \cdot Acc_i - \sum_{f_n \in \mathbf{Z_i}} \sum_{k=1}^{M} \sum_{m=1}^{con_k} y_{nkm} - \sum_{e_n \in \mathbf{Z_i}} \sum_{k=N+1}^{N+P} x_{nk} \tag{11}$$

5 Cost Functions

Algorithm 1 (in Section 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. For the minimisation of T this is relatively simple since 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 [2]. Creating a (real) variable for T, its minimum value can be determined using the constraints represented by equation (12). 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 (12) 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.

$$\forall l_j \in \mathbf{L}, \forall t_k \in \mathbf{T}, \qquad T \cdot ports_k \geq \sum_{i=1}^{N} Acc_{ij} \cdot Iss_k + \Big(\sum_{m=1}^{con_k} e_{jkl} - 1 \Big) \cdot X_j \cdot Iss_k \tag{12}$$

With the T variable bound by equation (12), 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. Unfortunately it is not so simple to generate linear cost functions to minimise the values of S, II or Z_p as all of these factors depend on how exactly the memory operations may be scheduled once the memory subsystem has been set. Essentially the lengths of the longest paths through the dependence graph must be minimised. If the minimisation of any path 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 [1], but the methods involved are complicated and of exponential time complexity. We have chosen not to implement such a scheme due to its complexity relative to the limited gains it is likely to achieve. In [8] 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 we use estimates of the lower bound values of S and II to bound how far the solution we find is from the most optimistic lower bound. Further details of this are given in Section 6.

We cannot guarantee to find the memory subsystem for the minimum number of imperfect stages, Z_p, without complex methods for examining critical paths. However, we can provide a simple, linear heuristic 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 [8], 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 (13), where Lat_{imp} is the latency of the longest path through the imperfectly nested operations.

$$Z_p \geq \left\lceil \frac{Lat_{imp}}{S \cdot T} \right\rceil \tag{13}$$

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 (14). Note that acc_k is a constant value for each resource type.

$$acc_k = \left\lfloor \frac{T}{iss_k} \right\rfloor \cdot ports_k \tag{14}$$

Z_p is added to the ILP formulation as an integer variable and its value is constrained by (13) and (15). 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. We use $Z_p + 1$ on the left hand side of (15) 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.

$$\forall l_j \in \mathbf{L}, \forall t_k \in \mathbf{T}, \qquad acc_k \cdot (Z_p + 1) \geq \sum_{i=1}^{N} Acc_{ij} + \sum_{i=1}^{N} (d_{ij} \cdot Imp_i) + \Big(\sum_{m=1}^{con_k} e_{jkl} - 1 \Big) \cdot Y_j \tag{15}$$

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 level in the loop 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 equation (13), 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 equation (15), 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 also 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 and buffer filled at level n. When filling a FIFO, 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 (imperfectly nested) 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 equation (16). FF_{ni} is the number of writes required to initialise FIFO f_n at level n.

$$\forall f_i \in \mathbf{F}, \qquad I_n \ \geq \ \sum_{k=1}^{M} \sum_{m=1}^{con_k} y_{ikm} \cdot FF_{ni} \qquad (16)$$

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 (17), where FB_{ni} is the number of writes required to initialise buffer e_i at level n.

$$\forall e_i \in \mathbf{E}, \qquad I_n \ \geq \ \sum_{j=1}^{P} x_{ij} \cdot \left\lceil \frac{FB_{ni}}{R_i} \right\rceil \qquad (17)$$

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 (18).

$$\forall l_j \in \mathbf{L}, \forall t_k \in \mathbf{T}, \qquad acc_k \cdot I_n \ \geq \ \sum_{i=1}^{Q} \sum_{k=1}^{M} \sum_{m=1}^{con_k} y_{ikm} \cdot FF_{ni} + \sum_{i=1}^{P} \sum_{j=1}^{P} x_{ij} \cdot \left\lceil \frac{FB_{ni}}{R_i} \right\rceil \qquad (18)$$

6 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, we cannot guarantee 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, if the minimum latencies of the critical paths and cyclic paths in the dependence graph (ignoring resource constraints) are known, then lower bounds can be placed on the values on the values of S and II. Polynomial time algorithms exist to solve both of these problems [13, 14] so these values can be found relatively quickly. Thus, with some modifications to the search algorithm (Algorithm 1) we can either produce the optimal pipelined solution (within the restrictions described in Section 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 the revised search algorithm. Firstly, not only can we not optimise the memory to minimise S or II, but we 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), we must check that the target loop may still be scheduled within the given S stages. 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 onto the next S value. Since the optimisation and bounding of the Z_p value is heuristic based, we 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}, another function is used to schedule the target loop for the given value of Z_p (and S). If this fails we find the new lower bound schedule length and revert to the memory subsystem optimised for minimum Z_p and recheck. If this also fails we revert to the memory for minimum T and recheck once more. If this final attempt fails then the search skips onto the next Z_p.

7 Results

The combined memory optimisation and pipelining methodology has been automated and applied to seven test loops. The results are presented in Table 1. In each case, along with the dependence graph for the test loop, we input a list of the memory resources, plus a list of buffers that may be inferred using other methodologies [6, 11]. The tool generates a VHDL controller for the pipeline, along with a schedule for the datapath and a description of the memory subsystem. The datapath must be written manually, but there is scope to automate this. The on-chip memory resources targeted are those on the Altera Stratix II EP2S30. We have included a single 16MByte bank of off-chip SRAM for each loop, except the hydrodynamics kernel. In this case we included two 16MByte banks of off-chip SRAM as it uses six 1000x1000 element floating point matrices. In each case we pipelined the loop with no automated memory optimisations (the 'NONE' option in Table 1) 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. For most of the loops we include results for two forms of memory optimisation. The 'UNBOUND' option in Table 1 refers to memory optimisation with no user-supplied bindings for specific arrays to be placed in specific memories. We have also included 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 1. There is no 'BOUND' option for the matrix multiply and hydrodynamics kernels as all the arrays are too large to store on chip.

Run time and scalability are always concerns for ILP based solutions. In our formulation the number of binary variables scales quadratically with the number of arrays and the number of buffers, and linearly with the number of FI-

Table 1. Results for edge detection (ED), motion estimation (ME), matrix-matrix multiply (MMM), hydrodynamics (HD) [12], successive over relaxation (SOR), MINRES (MIN) 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. We define the access ratio 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.

Loop	Memory optimisation	No. FIFOs (used/total)	No. buffers (used/total)	Off-chip accesses reduced by:	Access ratio	T	Cycles	Dev (cycles)	Speedup
ED	NONE	-	-	-	5.00	10	655,370	-	4.70
ED	UNBOUND	7/8	0/8	90.0%	1.00	1	69,905	0	44.06
ED	BOUND	7/8	1/8	79.9%	1.02	1	136,225	0	22.61
ME	NONE	-	-	-	25.0	2	3,277,326	-	3.99
ME	UNBOUND	0/28	0/8	50.0%	12.3	1	1,638,415	0	7.98
ME	BOUND	0/28	1/8	48.0%	13.0	1	1,704,207	0	7.67
MMM	NONE	-	-	-	667	2	2,002,000,030	-	8.49
MMM	UNBOUND	0/2	1/5	50.0%	334	1	1,002,039,000	0	16.96
HD	NONE	-	-	-	1.571	6	6,000,294	-	13.67
HD	UNBOUND	4/5	0/7	27.2%	1.143	4	4,060,392	8000	20.20
SOR	NONE	-	-	-	1.998	2	2,004,118	-	9.51
SOR	UNBOUND	2/2	0/4	50.1%	1.000	1	1,000,039	0	19.03
SOR	BOUND	2/2	1/4	50.0%	1.000	1	1,003,059	0	18.98
MIN	NONE	-	-	-	1.996	2	2,002,242	-	8.50
MIN	UNBOUND	0/0	0/3	50.0%	1.000	1	1,000,273	0	17.01
MIN	BOUND	0/0	1/3	49.9%	1.000	1	1,003,218	0	16.96
MED	NONE	-	-	-	5.00	10	655,370	-	5.30
MED	UNBOUND	7/8	0/8	90.0%	1.00	1	70,415	510	49.32
MED	BOUND	8/8	0/8	79.9%	1.02	2	136,718	493	25.40

FOs. For six of the seven test loops the complete scheduling search completed in under a minute, with the hydrodynamics kernel taking around 5 minutes. This is possibly due to the large number of equivalent solutions in this case.

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 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.

We define the access ratio for each example 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. 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 we achieved accesses ratios that were close to or exactly one. 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 unrolling the loop to create extra parallelism. Extending the methodology presented here to include array partitioning and loop unrolling is future work.

The 'DEV' column in Table 1 lists how far each solution is from the the lower bound solution derived during the

Table 2. 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	204	250	0 / 0 / 0	390
ED / BOUND	851	809	1 / 1 / 6	331
MMM / NONE	1175	1360	0 / 0 / 0	252
MMM / UNBOUND	1198	1755	0 / 8 / 0	247

search. Note that the lower bound is estimated using 'best case' results and will never be overestimated. In most cases we were 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 we 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 2 shows the implementation results on a Stratix II EP2S30 device for the edge detection and matrix multiply kernels. The 'BOUND' edge detection example uses 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, we achieved a 4.8x scheduling speedup over the 'NONE' case, giving an overall acceleration of 4x. The matrix multiply indicates the tradeoff in a case where fewer reuse options are used. There is a relatively small increase in resource usage, and only a 2% drop in clock rate compared to a scheduling speedup of 2x. In each case the designer must decide whether the increase in speed warrants the extra resources.

8 Conclusion

We have presented a combined array to memory assignment and data reuse selection approach that may be combined with loop pipelining to provide co-optimised schedules and memory subsystems. Our initial results have shown an average speedup of up to 4x over non-optimised results. 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 usefulness of this approach.

Acknowledgment

This work was partially funded by the EPSRC (EP/C549481/1).

References

[1] N. Baradaran and P. Diniz, "Memory Parallelism Using Custom Array Mapping to Heterogeneous Storage Structures," in *Proc. IEEE Int. Conf. Field Programmable Logic and Applications*, 2006, pp. 1–6.

[2] 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.

[3] 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.

[4] 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.

[5] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology*. Kluwer Academic Publishers, 1998.

[6] 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.

[7] 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.

[8] K. Turkington, G. Constantinides, K. Masselos, and P. Cheung, "Outer Loop Pipelining for Application Specific Datapaths in FPGAs," vol. 16, no. 10, 2008.

[9] H. Williams, *Model Building in Mathematical Programming (Fourth Edition)*. Wiley, 1998.

[10] *Statix II Device Handbook*, Altera Corp., San Jose, CA, 2007.

[11] 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.

[12] F. McMahon, "The Livermore Fortran Kernels Test of the Numerical Performance Range," *Performance Evaluation of Supercomputers*, pp. 143–186, 1988.

[13] 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.

[14] 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.