Pipeline Exploration for Reconfigurable Targets

Kieron Turkington
Imperial College London
kjt01@imperial.ac.uk

George A. Constantinides
Imperial College London
g.constantinides@imperial.ac.uk

Konstantinos Masselos
University of Peloponnese
k.masselos@imperial.ac.uk

Peter Y.K. Cheung
Imperial College London
p.cheung@imperial.ac.uk

ABSTRACT

The ability to compile software into an efficient, synthesisable hardware description can dramatically reduce the design times for embedded systems that require the acceleration offered by hardware coprocessors. Most hardware compilers apply loop pipelining to increase the parallelism achieved, but pipelining is restricted to the only innermost level in a nested loop. Methods have been developed for software compilers that extend loop pipelining to levels above the innermost loop but they have yet to be extended for hardware.

In this work we extend and adapt an existing outer loop pipelining approach known as Single Dimension Software Pipelining to generate schedules for FPGA hardware coprocessors. Each loop level in four test loops is pipelined and the resulting schedules are implemented in VHDL and targeted to an Altera Stratix II FPGA. The results show that the fastest solution for each loop occurs when pipelining is applied one or two levels above the innermost loop. Across the four test loops we achieve an acceleration over the innermost loop solution of between 1.15 and 7 times, with a mean speedup of 2.56 times. The results suggest that inclusion of outer loop pipelining in future hardware compilers may be worthwhile as it can allow significantly improved results to be achieved at the cost of a small increase in compile time.

1. INTRODUCTION

The real-time demands of embedded applications, especially those related to image, video and signal processing, often exceed the processing capabilities of embedded microprocessors [5]. To overcome this problem custom hardware coprocessors may be included in the system to implement one or more compute expensive (nested) loops within the target application, allowing the microprocessor to implement the less demanding sections of the algorithm. While hardware coprocessors can dramatically increase the performance of an embedded system, their development can be a lengthy and complicated process when compared to software development. To address this problem a number of tools have been developed that allow hardware coprocessors to be generated automatically from existing software descriptions [2, 4, 7, 13]. Of key importance when generating hardware is the exploitation of parallel execution in the algorithm as this is how acceleration is achieved. Loop pipelining [1] techniques 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 [1, 4, 7, 13]. 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 [10, 12, 16] and these can allow shorter schedules to be achieved. The Single Dimension Software Pipelining (SSP) approach [10] 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, unroll and tiling 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 [13].

In this work we extend and adapt the existing SSP approach to better suit the generation of schedules for hardware, specifically FPGAs. We also introduce a search scheme to find the optimal (fastest) schedule available within the pipelining framework to maximise the gains in pipelining above the innermost loop. Our scheduling approach is applied to four 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 one or two levels above the innermost loop. When compared to inner loop pipelining a maximum speedup of 7 times is achieved along with an average speedup across the four loops of 2.56 times.

The remainder of the paper is split into seven further sections. In Section 2 we provide a brief description of Modulo Scheduling and the existing SSP methodology. Sections 3 and 5 describe our extension and adaptation of this existing methodology to improve results when targeting FPGAs, and Section 4 describes the FPGA resource constraints. In Section 6 we present details of our scheme to search the solution space, along with details of our modulo scheduling formulation. Section 7 details the results achieved for four test loops and Section 8 summarises the conclusions of this work.

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

2. BACKGROUND

Perhaps the most widely used loop pipelining methods are based around modulo scheduling [1]. 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 [1]. Single-dimension Software Pipelining (SSP) [10] 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 [10]. By assuming that the iterations from loop levels above and below the pipelined level execute sequentially, all dependence distance vectors [8] 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 [10]. 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.

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 work [10]. In this section we show 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. The determination of $ResMII$ is discussed in Section 4 while the calculation of $RecMII$ is described in [9]. 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.

From the scheduling function presented in [10] an expression for the number of clock cycles required to execute a rectangular, perfectly nested loop with L levels of nesting[1] can be derived. Although the derivation is not included for brevity, the length of the schedule is defined by equation (1). N_i represents the number of loop iterations at level i in the nest[2] and p is the loop level selected for pipelining. 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.

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 [3]. Often it is not the physical resources on the device that limit performance, but the memory bandwidth available to access the arrays used by the application [14]. In this work we assume that the number of available memory ports will limit the parallelism that may be exploited before the physical 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 [3].

With these assumptions and a valid array to physical memory map (which is must be supplied by the designer in this work) 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 (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[3] 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 (3).

$$cyc_m = \left\lceil \frac{Iss_{rm} \cdot R_m + Iss_{wm} \cdot W_m}{ports_m} \right\rceil \quad (2)$$

$$ResMII = \max_m(cyc_m) \quad (3)$$

[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

[3] In this description we consider only read/write ports, but the methodology is easily extended to deal with dedicated read and write ports

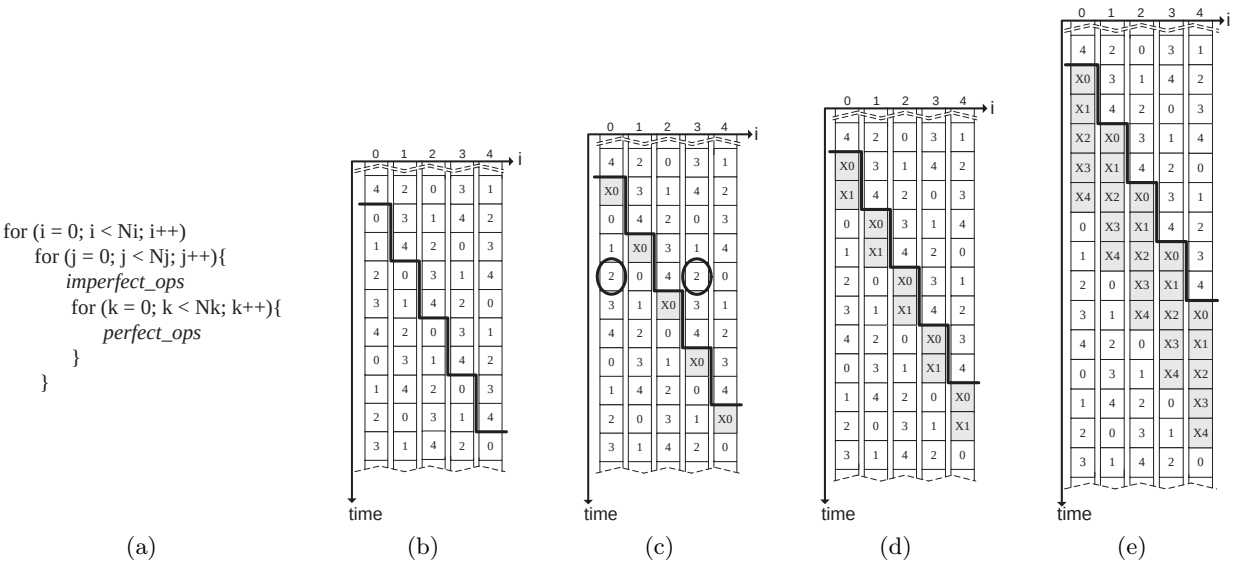

Figure 1: Scheduling imperfectly nested operations. Each numbered box represents an execution of the stage denoted by the enclosed number. Each vertical column of stages represents part of the execution of one of the first five outer loop iterations. (a) A sample loop nest (b) A section of the perfectly 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

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 [11]. This methodology requires the use of multiple stage lengths to achieve maximum efficiency. The implementation of multiple stage lengths in FPGA hardware would require complex control that would ultimately reduce the clock frequency of the final pipeline. To avoid such complications a simpler method for scheduling imperfectly nest 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 (or sets) of S, where S is the number of perfect stages. The reasoning behind this is demonstrated using the simple example in Figure 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[4]. Once the pipeline has been filled the perfect schedule follows the pattern shown in Figure 1(b). The stepped bold line in Figure 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 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 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 less than S (five in this case). Figure 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 matches that of the perfect stage it replaces, 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 we move outwards from the innermost loop the number of imperfectly nested instructions may increase. Hence, to improve the efficiency of our approach, we allow an increasing number of sets of stages to be included for each level in the loop above the innermost loop. We define Z_i to be the number

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

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.

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 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, the number of sets of imperfect stages, Z_p, and the initiation interval, II. The second part is an Integer Linear Programming (ILP) [15] 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 or the total number of stages executed is minimised.

The two inputs to the scheduling process are the data dependence graph for the nested loop and a valid 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 [10]. 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.

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) and (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[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. 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 (5) is $\left(T \cdot \prod_{i=1}^{L} N_i\right)$ cycles. 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.

[5] Z_i is defined to be zero for all $i \geq L$

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
```

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

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

$$S_{tot} = S \cdot \left(\sum_{i=p+2}^{L} \left(Z_i \cdot (N_i - 1) \cdot \prod_{j=p+1}^{i-1} N_j \right) + (N_{p+1} - 1) \cdot Z_{p+1} + Z_p + \prod_{i=p+1}^{L+1} N_i \right) \quad (4)$$

$$cycles = \left(\prod_{k=0}^{p-1} N_k \right) \cdot \left(\left(\lceil N_p / S \rceil - 1 \right) \cdot \max \left(T \cdot S_{tot}, S \cdot II \right) + (T \cdot S_{tot}) + \left((N_p - 1) \bmod S \right) \cdot II + (S \cdot Z_p \cdot T) \right) \quad (5)$$

(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 (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.

The *schedule*, *find_S_tot_min* and *find_II_min* functions all make use of the ILP modulo scheduling formulation described described in Section 6.2. The functions set different variable bounds and minimise different cost functions which are discusses at the end of Section 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). 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 7 the search was found to bound after far fewer iterations.

6.2 Modulo Scheduling

Modulo scheduling of the simplified dependence graph is implemented using ILP as this provides a simple method for obtaining schedules with minimal cost functions. The cost functions used are discussed at the end of this section. 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 a real constant l_n. Each edge in E_s produces a linear constraint in the scheduling formulation, as described by equation (6). II is an integer variable representing the initiation interval. 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}.

$$\forall (v_{n1}, v_{n2}) \in E_s, \quad x_{n2} + d_{n1n2} \cdot II \geq x_{n1} + l_{n1} \quad (6)$$

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 equations (7) and (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.

$$\forall v_n \in P, \quad x_n + l_n \leq S \cdot T \quad (7)$$

$$\forall v_n \in P, \quad x_n \geq 0 \quad (8)$$

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 equations (9) and (10) must be met during scheduling. I is the set of all imperfectly nested operations in the dependence graph.

$$\forall v_n \in I, \quad x_n + l_n \leq Z_p \cdot S \cdot T \quad (9)$$

$$\forall v_n \in I, \quad x_n \geq -Z_p \cdot S \cdot T \quad (10)$$

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 equations (11) and (12). I_i is the set of operations nested imperfectly up to and including loop level i.

$$\forall v_n \in I_i, \quad Z_i \cdot S \cdot T \geq -x_n \quad (11)$$

$$\forall v_n \in I_i, \quad Z_i \cdot S \cdot T \geq x_n + l_n - S \cdot T \quad (12)$$

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 (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}[6], 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 (13) and (14). The constraints defined by equation (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.

$$\forall v_n \in P_m, \quad x_n = st_n \cdot T + \sum_{t=0}^{T-1} t \cdot d_{nt} \tag{13}$$

$$\forall v_n \in P_m, \quad \sum_{t=0}^{T-1} d_{nt} = 1 \tag{14}$$

$$\forall 0 \leq t \leq T, \quad \sum_{v_n \in P_m} d_{nt} \leq ports_m \tag{15}$$

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. The resource constraints required for multiple sets of imperfect stages ($Z_p > 1$) differ from those for a single set of stages ($Z_p = 1$). Due to space constraints only the constraints for a single set of imperfect stages are described since the optimal solution for the loops we target normally requires only a single set of imperfect stages. For each perfectly nested memory operation, v_n, assigned to memory m, an extra $S \cdot T$ binary variables, d_{nst}[7], 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 equations (16) and (17). P_m again represents the set of perfect memory accesses to memory m.

$$\forall v_n \in P_m, \quad x_n = \sum_{s=0}^{S-1} \sum_{t=0}^{T-1} (s \cdot T + t) \cdot d_{nst} \tag{16}$$

$$\forall v_n \in P_m, \quad \sum_{s=0}^{S-1} \sum_{t=0}^{T-1} d_{nst} = 1 \tag{17}$$

[6] $(0 \leq t < T)$

[7] $(0 \leq s < S)$ and $(0 \leq t < T)$

For each imperfectly nested memory operation, v_n, assigned to memory m, a further $S \cdot T \cdot (Z_p + 1)$ binary variables, $d_{n\rho st}$[8], 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 (18) and (19). I_m represents the set of imperfect access operations assigned to memory m.

$$\forall v_n \in I_m,$$
$$x_n = \sum_{\rho=0}^{1} \sum_{s=0}^{S-1} \sum_{t=0}^{T-1} \Big((-\rho \cdot S \cdot T + s \cdot T + t) \cdot d_{n\rho st} \Big) + 2 \cdot ba_n \cdot S \cdot T \tag{18}$$

$$\forall v_n \in I_m, \quad \sum_{\rho=0}^{1} \sum_{s=0}^{S-1} \sum_{t=0}^{T-1} d_{n\rho st} = 1 \tag{19}$$

The resource constraints for memory m in the imperfect system are defined by equations (20) and (21). Equation (20) defines the resource constraints for the perfect stages while equation (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.

$$\forall\ 0 \leq t < T,$$
$$\sum_{s=0}^{S-1} \Big(\sum_{v_n \in P_m} d_{nst} + \sum_{v_n \in I_m} d_{n0st} + slack_{mst} \Big) \leq ports_m \tag{20}$$

$$\forall\ 0 \leq s < S,\ \forall\ 0 \leq t < T,$$
$$\sum_{v_n \in I_m} d_{n1st} \leq \sum_{v_n \in P_m} d_{nst} + \sum_{v_n \in I_m} d_{n0st} + slack_{mst} \tag{21}$$

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). The *schedule* function also places an upper bound on the value of II. The *find_II_min* function uses II as the cost function.

The II value returned by the ILP formulation may cause resource conflicts in certain cases. Although a proof is not included due to space constraints it can be shown that, when II and T have different values, the constraints that II is an integer multiple of T and that the greatest common divisor (gcd) of (II/T) and S equals one must be met. When these

[8] $(0 \leq \rho < Z_p)$, $(0 \leq s < S)$ and $(0 \leq t < T)$

Table 1: Scheduling results for the edge detection (ED), motion estimation (ME), matrix-matrix multiply (MMM) and hydrodynamics (HD) 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

Loop	Level	Cycles	Average Parallelism	Speedup
ED	4	1,935,480	2.40	1.000
ED	3	1,419,352	3.27	1.364
ED	2	1,297,432	3.68	1.492
ED	1	1,295,408	3.69	1.494
ME	6	11,943,936	3.55	1.000
ME	5	10,741,248	3.95	1.112
ME	4	10,653,696	3.99	1.121
ME	3	10,620,928	4.00	1.125
ME	2	10,617,568	4.00	1.125
ME	1	10,617,358	4.00	1.125
MMM	3	7,014,000,000	2.42	1.000
MMM	2	2,002,030,000	8.49	3.503
MMM	1	2,002,000,030	8.49	3.503
HD	2	42,006,000	1.95	1.000
HD	1	6,000,294	13.67	7.001

constraints are not met by the II value returned from the ILP the minimum legal II for the given values of T and S may be found by increasing II to the next integer multiple of T such that the *gcd* of (II/T) and S is one.

7. RESULTS

Our extended Single Dimension Software Pipelining algorithm has been used to pipeline each level in four 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 a set of parameterised component blocks. The loops used are an image edge detection kernel, a motion estimation kernel, a matrix-matrix multiply and a 2D hydrodynamics fragment taken from the Livermore Loops [6]. The edge detection and motion estimation algorithms act upon 256x256 pixel images (8 bit fixed point) and the search window for the motion estimator is +/-4 pixels. The matrix multiply and hydrodynamics kernels operate on 1000x1000 element (single precision) floating point matrices. 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.

The scheduling results for each level in the four test loops are detailed in Table 1. In each case pipelining above the innermost loop level does yield a shorter schedule, but the original Single Dimension Software Pipelining work [10] has already shown this to be the case. 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 in first increased to 4 to accommodate extra imperfectly nested instructions (allowing Z_p to be zero), and then increased to nine so that it is again a factor of N_p (which is 9 in both cases).

Table 2: Hardware implementation results for the four test loops. The speedup figure is relative to the innermost loop solution.

Loop	Level	Fmax (MHz)	Cycles	Time (s)	Speedup
ED	4	457	2,129,028	0.0046	1.000
ED	3	438	1,548,384	0.0035	1.318
ED	2	431	1,297,940	0.0030	1.533
ED	1	399	1,295,410	0.0032	1.439
ME	6	497	12,607,488	0.0254	1.000
ME	5	489	10,782,720	0.0221	1.149
ME	4	348	10,658,304	0.0306	0.830
ME	3	334	10,621,440	0.0318	0.799
ME	2	434	10,617,600	0.0245	1.036
ME	1	439	10,617,360	0.0242	1.050
MMM	3	186	7,016,000,000	37.72	1.000
MMM	2	186	2,002,032,000	10.76	3.504
MMM	1	182	2,002,000,032	11.00	3.430
HD	2	179	42,008,000	0.235	1.000
HD	1	179	6,000,296	0.034	7.001

Table 2 details the performance results the four test loops when the pipelined hardware accelerator for each loop level is targeted to an Altera Stratix II FPGA. Only the control and data path 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 1 to produce those shown in Table 2. For each loop 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 2 show that, in all four cases, the optimum (fastest) solution occurs when pipelining above the innermost loop. However, we notice 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 or two loop levels above the innermost loop as this these levels offer the

best tradeoff between the scheduling gains and the clock frequency. This is contrary to the claims in existing work that extending pipelining above the innermost is rarely worthwhile [13].

The degradation in the maximum clock frequency varies across the four target loops. The edge detection data path is a small, simple circuit and so the critical path through the complete design lies within the controller. Hence we see a steady decline in the clock frequency as the pipelining level moves up through the loop and the controller becomes more complex. The motion estimator also has a relatively simple data path 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 data path complexity, moving the critical path into the data path. 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 data paths for the matrix multiplication and hydrodynamics kernels include floating point units and, for each loop level in both loops, the critical path lies within these units and not the controller. Hence there is virtually no degradation in clock frequency.

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 LUT and register usage roughly double that for the innermost loop. The more complex data paths of the matrix multiplication and hydrodynamics kernels lead to smaller increases in the resource usage. The fastest solution for the matrix multiplication uses 1.3 times the resources of the pipeline for the innermost loop while the hydrodynamics kernel uses no more resources when the outer loop is pipelined. In every case the designer must decide whether the increase in speed is necessary or warrants the extra resource usage.

8. CONCLUSIONS

In this work 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. We have also introduced a simplified method for dealing with imperfectly nested instructions that 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.

Our scheduling tool has been applied to four test loops. In each case, when the resulting coprocessors are targeted to an Altera Stratix II FPGA, the fastest solution is found when the loop is pipelined one or two 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 we achieved speedups over the innermost loop solution ranging from 1.15 times to 7 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.

9. ACKNOWLEDGMENTS

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

10. REFERENCES

[1] V. Allan, R. Jones, R. Lee, and S. Allan. Software Pipelining. *ACM Proc. Computing Surveys*, 27(3):367–432, 1995.

[2] Altera Corp. *Nios II C2H Compiler User Guide*, 2007.

[3] Altera Corp. *Statix II Device Handbook*, 2007.

[4] 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*, pages 112–117, 2005.

[5] B. Hounsell and R. Taylor. Co-processor Synthesis: A New Methodology for Embedded Software Acceleration. In *Proc. Conf. Design, Automation and Test in Europe*, pages 682–683, 2004.

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

[7] Mentor Graphics. *Catapult C Datasheet*, 2006.

[8] W. Pugh. Definitions of Dependence Distance. *ACM Letters on Programming Languages and Systems*, 1(3):261–265, 1992.

[9] R. Ramakrishna. Iterative Modulo Scheduling. Technical Report HPL-94-115 , Hewlett Packard Laboratories, 1995.

[10] 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*, pages 163–174, 2004.

[11] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. Gao. Single-Dimension Software Pipelining for Multi-Dimensional Loops. Technical Report CAPSL Technical Memo 049, University of Delaware, 2005.

[12] 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*, pages 3065–3068, 1998.

[13] M. Weinhardt and W. Luk. Pipeline Vectorization for Reconfigurable Systems. In *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, pages 52–62, 1999.

[14] M. Weinhardt and W. Luk. Memory Access Optimization for Reconfigurable Systems. *IEE Proc. Computers and Digital Techniques*, 148(3):105–112, 2001.

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

[16] 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*, pages 345–350, 1997.