Outer Loop Pipelining for Application Specific Datapaths in FPGAs

Kieron Turkington, George A. Constantinides, *Member, IEEE,* Konstantinos Massselos, *Member, IEEE,* and Peter Y.K. Cheung, *Senior Member, IEEE*

***Abstract*—Most hardware compilers apply loop pipelining to increase the parallelism achieved, but pipelining is restricted to the only innermost level in a nested loop. 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 nine 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 all but one of the loops occurs when pipelining is applied one to three levels above the innermost loop. Across the nine test loops we achieve an acceleration over the innermost loop solution of up to 7 times, with a mean speedup of 3.2 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.**

***Index Terms*—Pipelining, nested loop, field-programmable gate array (FPGA), integer linear programming (ILP).**

I. 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 [1]. To overcome this problem custom hardware coprocessors may be included in the system to implement one or more compute expensive 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 relative 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]–[5]. Of key importance when generating hardware is the exploitation of parallel execution in the algorithm as this is how acceleration is achieved. Loop pipelining [6] 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 [3]–[6]. 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 [7]–[9] and these can allow shorter schedules to be achieved. The Single Dimension Software Pipelining (SSP) approach [7] 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 [5].

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 shortest schedule available within the pipelining framework to maximise the gains in pipelining above the innermost loop. We also present a generic pipeline controller 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. Our 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 3.2 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 nine test loops and Section 8 summarises the conclusions of this work.

II. Background

Perhaps the most widely used loop pipelining methods are based around modulo scheduling [6]. In modulo scheduling the operations from a single iteration of the loop body are

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

K. Turkington, G. Constantinides and P. Cheung are with the Department of Electrical and Electronic Engineering, Imperial College London, Exhibition Road, London SW7 2BT. e-mail {kjt01, g.constantinides, p.cheung}@imperial.ac.uk

K. Masselos is with the Department of Computer Science and Technology, University of Peloponnese, Tripolis 22100, Greece. e-mail kmas@uop.gr

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

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 [6]. Single-dimension Software Pipelining (SSP) [7] 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 [7]. By assuming that the iterations from loop levels above and below the pipelined level execute sequentially, all dependence distance vectors [10] 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 [7]. 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.

III. 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 [7]. 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 [11]. 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 [7] an expression for the number of clock cycles required to execute a rectangular, perfectly nested loop with L levels of nesting[1] can be obtained by substituting the values for the last operation in the loop. This 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.

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 (please see the following section for more details) may allow smaller values of $ResMII$, such that $RecMII$ dominates. Thus we felt it 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 simply extend 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, we make no assumption about the relative values of $RecMII$ and $ResMII$ and the methodology presented in the remainder of this paper 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 offer less significant (if any) gains over the original SSP approach.

IV. 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 [12]. It may therefore be reasonable to assume

[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

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 [13]. 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 we assume 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 [12].

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 (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 (3).

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

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

V. 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 [14]. 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 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 (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[5]. 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 fewer 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 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.

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

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

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

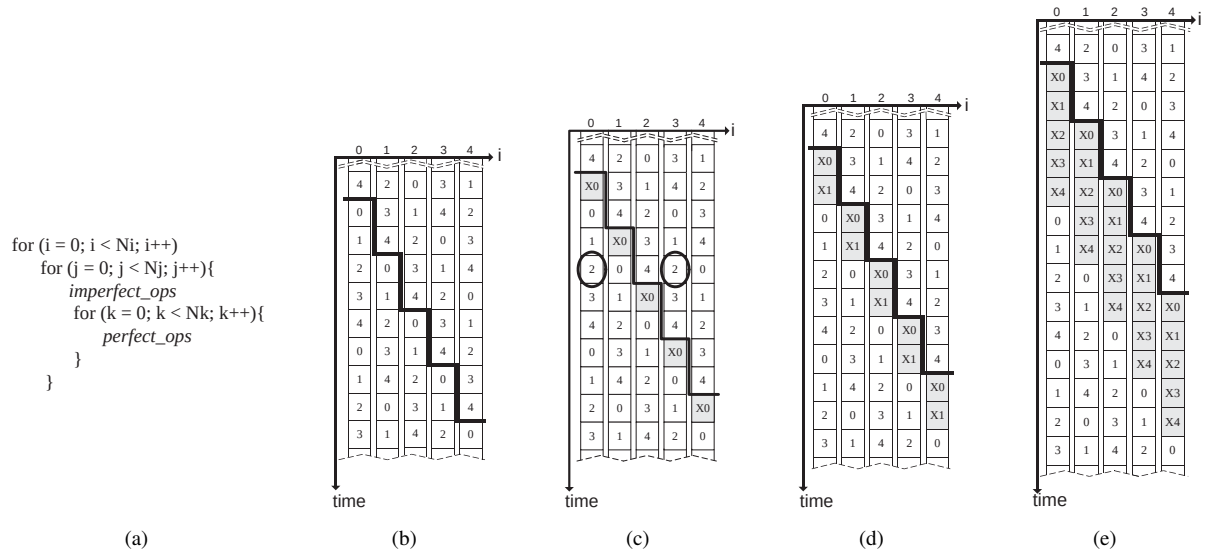

Fig. 1. Scheduling imperfectly nested operations. Each numbered box represents an execution of the stage denoted by the enclosed number. Each vertical column of stages represents part of the execution of one of the first five outer loop iterations. (a) A sample loop nest (b) A section of the perfectly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

VI. 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 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 [7]. 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.

A. 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[6]. Equation (4) is simply a weighted sum of the number of sets of stages nested at each level in the loop and equation (5) is simply the imperfect generalisation of equation (1). 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.

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

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

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

$$cycles = \left(\prod_{k=0}^{p-1} N_k \right) \cdot \left((N_p - 1) \cdot II + (\lfloor N_p - 1/S \rfloor + 1) \cdot S_{tot} - \lfloor N_p - 1/S \rfloor \cdot S \cdot II \right) \quad (5)$$

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
```

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 (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.

B. 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 an integer 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}.

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

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.

$$\forall v_n \in P, \quad x_n + l_n \leq S \cdot T \tag{7}$$

$$\forall v_n \in P, \quad x_n \geq 0 \tag{8}$$

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.

$$\forall v_n \in I, \quad x_n + l_n \leq Z_p \cdot S \cdot T \tag{9}$$

$$\forall v_n \in I, \quad x_n \geq -Z_p \cdot S \cdot T \tag{10}$$

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.

$$\forall v_n \in I_i, \quad Z_i \cdot S \cdot T \geq -x_n \tag{11}$$

$$\forall v_n \in I_i, \quad Z_i \cdot S \cdot T \geq x_n + l_n - S \cdot T \tag{12}$$

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}[7], 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}[8], 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.

[7] $(0 \leq t < T)$
[8] $(0 \leq s < S)$ and $(0 \leq t < T)$

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

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

For each imperfectly nested memory operation, v_n, assigned to memory m, a further $S \cdot T \cdot (Z_p + 1)$ binary variables, $d_{n\rho st}$[9], 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.

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

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

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.

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

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

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 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.

VII. 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 [3], [5] 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 circuit, such as those shown in Figures 2(a) and 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 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 drop in clock frequency is minimised. If this is not the case then any potential decrease in the schedule length with be canceled by the drop in clock rate.

In deriving a control scheme for outer loop pipelining we first consider the generation of the loop index vectors for

[9] $(0 \le \rho < Z_p)$, $(0 \le s < S)$ and $(0 \le t < T)$

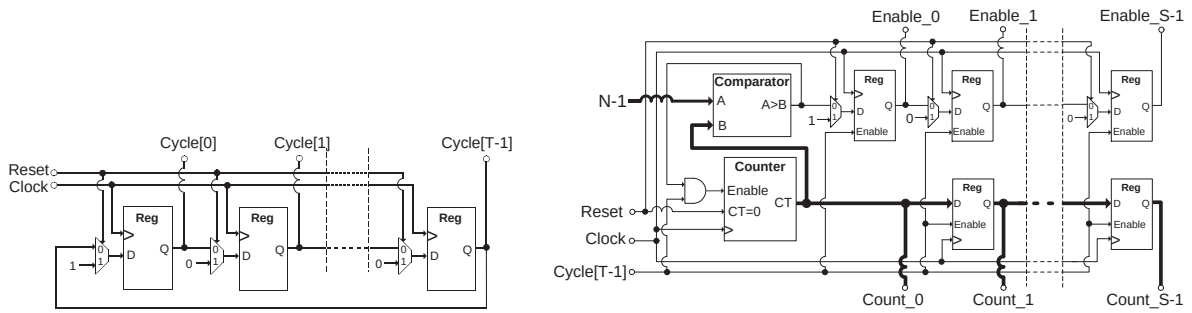

Fig. 2. A simple controller design for a loop pipelined at the innermost level (a) A cyclical shift register to indicate the current cycle in each stage (b) Control logic to generate the loop index and enable signal for each stage. The bold bus lines are of width $log_2 N$, where N is the number of loop iterations

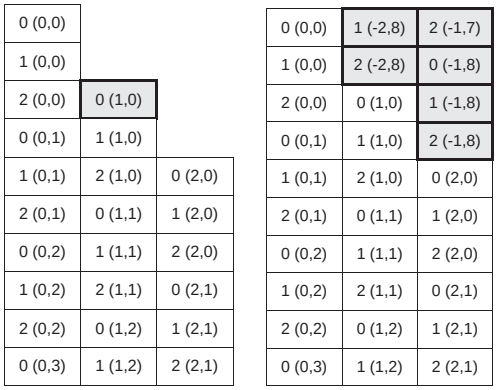

Fig. 3. A section of an example schedule for a double nested loop pipelined at the outermost level. It is assumed the loop has 9 iterations at each level and it has been scheduled into a pipeline with 3 stages and an initiation interval of 2 stages. These numbers are chosen arbitrarily. Each box represents 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.

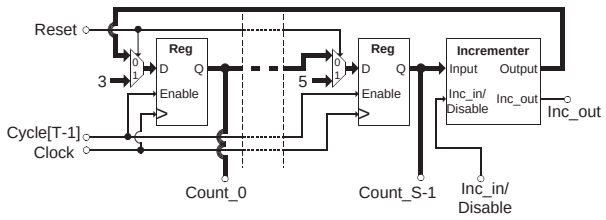

Fig. 4. A circuit to generate the loop index for each stage at one level in the loop. The bold bus lines are of width $log_2 N$, where N is the number of loop iterations at the given loop level. The 'Incrementer' block is detailed in Figure 5. The input values on the '1' input of the multiplexors (3 and 5 in this Figure) represent the initial reset values for the loop indices. The values shown were chosen arbitrarily and are not significant.

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 3(a). Starting from the highlighted stage, with time progressing vertically downwards, stage 0 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[10].

The circuit in Figure 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 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 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 the index vectors for each stage must be initialised correctly for the circuit to function as hoped. Fortunately 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 we extend the schedule back to the start of the loop execution as shown in Figure 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 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 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

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

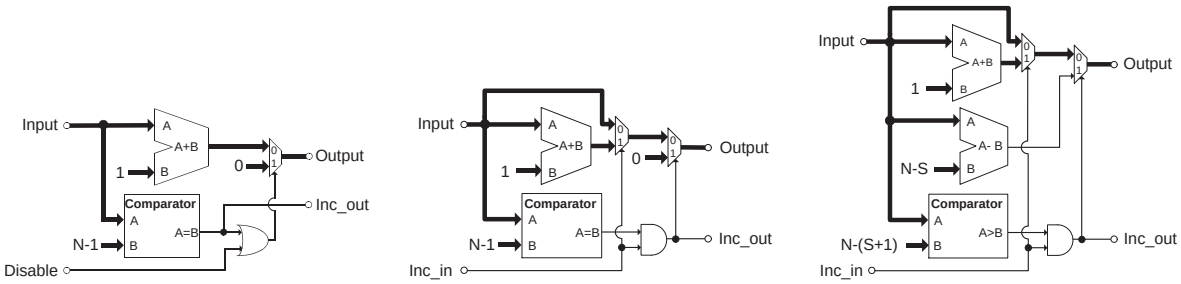

Fig. 5. The 'Incrementer' blocks used by the circuit in Figure 4 to update the loop indices. The bold bus lines are of width $log_2 N$, where N is the number of loop iterations at the given loop level (a) The 'Incrementer' for the innermost loop level (b) The 'Incrementer' for all loop levels between the innermost level and the pipelined level (a) The 'Incrementer' for the pipelined loop level

logic controlling the enable signals for a pipeline with only perfectly nested stages is shown in Figure 6 and Figure 7. The externsions to deal with imperfect nesting are relatively simple, requiring just additional shift registers and a small umber of lgoic gates, but these are not detailed here for brevity. 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 6 so that an additional '1' is input into the shift register for each new iteration[11], eventually enabling all S stages. The 'Enable/Disable' block, detailed in Figure 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 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.

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 greater than the number of innermost loop iterations in a single iteration at the pipelined level. Although the details for these cases cannot be detailed due to space constraints, 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

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

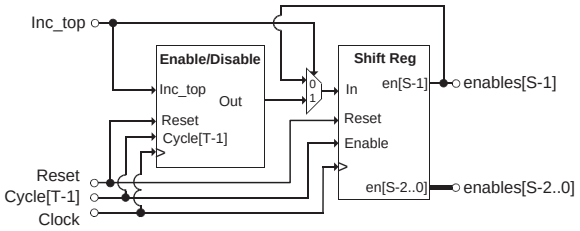

Fig. 6. The logic to control the enable signal for each pipeline stage. The 'Enable/Disable' block is detailed in Figure 7. The 'Shift Reg' block is a shift register with S bits. 'en[0]' is the output from the first register in the chain. 'en[S-1]' is the output from the last register in the chain. At 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

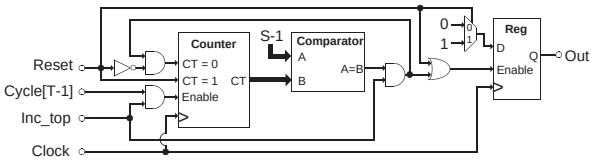

Fig. 7. The 'Enable/Disable' block. The 'Inc_top'input is connected to the 'Inc_out' from the 'Incrementer' block at the pipelined loop level

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.

VIII. 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 [16], a successive over relaxation kernel (SOR) [17], the MINRES

TABLE I
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

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

TABLE II
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

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

algorithm [18] and an LU decomposition kernel [19]. 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.

The scheduling results for each level in the four fixed point test loops are detailed in Table I and the results for the five floating point loops are detailed in Table II. In every case, aside from the FFT example, pipelining above the innermost loop level does yield a 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 [7] has already demonstrated the benefits of extending pipelining above the innermost loop. However, the results also demonstrate that our

TABLE III
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	124	108	457	2,129,028	0.0046	1.000
ED	3	127	110	438	1,548,384	0.0035	1.318
ED	2	266	181	431	1,297,940	0.0030	1.533
ED	1	288	233	399	1,295,410	0.0032	1.439
ME	6	91	142	489	3,891,200	0.0080	1.000
ME	5	115	163	483	3,328,000	0.0069	1.154
ME	4	191	271	347	3,289,600	0.0306	0.839
ME	3	205	281	335	3,279,360	0.0318	0.813
ME	2	305	269	384	3,277,568	0.0245	0.932
ME	1	326	280	406	3,277,328	0.0242	0.985
MED	5	51	58	500	52,428,800	0.1049	1.000
MED	4	163	216	500	36,372,480	0.0727	1.441
MED	2	243	474	496	21,369,088	0.0431	2.434
MED	1	317	539	500	21,364,753	0.0427	2.453
MMM	3	834	847	246	7,016,000,000	28.52	1.000
MMM	2	1067	1307	250	2,002,032,000	8.01	3.554
MMM	1	1175	1360	252	2,002,000,032	7.94	3.590
HD	2	3957	4449	233	42,008,000	0.180	1.000
HD	1	4088	4451	233	6,000,296	0.026	7.001
SOR	3	812	719	236	7,016,000	0.030	1.000
SOR	2	4600	8779	210	2,002,080	0.010	3.118
MIN	3	2445	4569	241	7,023,000	0.029	1.000
MIN	2	4071	7403	210	2,002,244	0.010	3.056
LU	3	805	1029	223	1,011,527,460	4.536	1.000
LU	2	1254	1722	219	1,008,293,931	4.604	0.985
LU	1	3156	4959	216	678,002,028	3.139	1.444

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 five so that it is again a factor of N_p (which is 5 in both cases).

Table III 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). 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 I and Table II to produce those shown in Tables III respectively. 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 III show that, in all eight 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 [5].

The degradation in the maximum clock frequency varies across the eight implemented 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 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. In fact, for both the matrix multiplication and hydrodynamics examples, the increase in resource usage when moving to the outer loop is less than would be expected bearing in mind the extra level of control added. This is most likely due to the heuristic nature of the optimisation algorithms used during the place and route process to map the design to the FPGA. These optimisations may duplicate logic and/or registers to increase the clock rate achieved, and this process may mask any actual increase in resource usage if the increase is small relative to the total. 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.

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. So what advantage does our approach offer over interchange and inner loop pipelining? Firstly, as pointed out in [7], 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 inner most 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 we are able 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 [7] and was shown to be of benefit.

IX. Conclusion

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 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 we achieved speedups over the innermost loop solution ranging from 1 (no speedup) to 7 times, with an average (root-mean-square) speedup of 3.22 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.

REFERENCES

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

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

[3] Z. Guo, B. Buyukkurt, and W. Najjar, "Optimized Generation of Datapath from C Codes for FPGAs," in *Proc. Design Automation and Test in Europe*, 2005, pp. 112–117.

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

[5] M. Weinhardt and W. Luk, "Pipeline Vectorization for Reconfigurable Systems," in *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 1999, pp. 52–62.

[6] V. Allan, R. Jones, R. Lee, and S. Allan, "Software Pipelining," *ACM Proc. Computing Surveys*, vol. 27, no. 3, pp. 367–432, 1995.

[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] 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.

[9] 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.

[10] W. Pugh, "Definitions of Dependence Distance," *ACM Letters on Programming Languages and Systems*, vol. 1, no. 3, pp. 261–265, 1992.

[11] R. Ramakrishna, "Iterative Modulo Scheduling," Hewlett Packard Laboratories, Tech. Rep. HPL-94-115 , 1995.

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

[13] 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.

[14] 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.

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

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

[17] A. Hadjidimos, "Successive Over Relaxation (SOR) and related methods," *Journal of Computational and Applied Mathematics*, vol. 123, no. 1-2, pp. 177–199, 2000.

[18] 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.

[19] S. Grossman, *Elementary Linear Algebra (Fourth Edition)*. Saunders College Publishing, 1994.

Kieron Turkington received the M.Eng (Hons) degree in Electrical and Electronic Engineering from Imperial College London, U.K., in 2005. He is currently a PhD student in the Circuits and Systems research group at Imperial College London, with research interests based around behavioural synthesis and memory optimisations for FPGAs.

George A. Constantinides (S96-M01) received the M.Eng. (Hons) in Information Systems Engineering and the Ph.D. degree from Imperial College London, London, U.K., in 1998 and 2001, respectively. In 2002, he joined the faculty at Imperial College London and is currently a Senior Lecturer.

Dr. Constantinides is a member of the IEEE, the ACM, and SIAM. He is associate editor of the IEEE Transactions on Computers and the Journal of VLSI Signal Processing. He was programme co-chair of the IEEE International Conference on Field-Programmable Technology in 2006 and Field Programmable Logic and Applications in 2003, and serves on the technical program committees of several conferences, including DATE, where he is chair of the Architectural Synthesis track in 2009.

Kostantinos Masselos (S92-M94) received a first degree in Electrical Engineering from University of Patras, Greece in 1994 and an MSc degree in VLSI Systems Engineering from University of Manchester Institute of Science and Technology, United Kingdom in 1996. In April 2000 he got a PhD degree in Electrical and Computer Engineering from University of Patras, Greece.

From 1997 until 1999 he was associated as a visiting researcher with the Inter-university Micro Electronics Centre (IMEC) in Leuven, Belgium, where he was involved in research related to the ACROPOLIS multimedia compiler. Until 2004 he was with INTRACOM S.A, Greece where he was involved in the realization of wireless communication systems. In 2005 he joined as a lecturer the Department of Electrical Engineering and Electronics of Imperial College London. Since 2006 he is an Assistant Professor in the Department of Computer Science and Technology of University of Peloponnese, Greece and a visiting lecturer at Imperial College. His main interests include compiler optimizations and high level synthesis, high level power optimization, FPGAS and reconfigurable hardware, and efficient implementations of DSP algorithms. He is a member of the IEEE.

Peter Y.K. Cheung (M85-SM04) received the B.S. degree with first class honors from Imperial College of Science and Technology, University of London, London, U.K., in 1973. Since 1980, he has been with the Department of Electrical Electronic Engineering, Imperial College, where he is currently a Professor of digital systems and deputy head of the department. He runs an active research group in digital design, attracting support from many industrial partners. Before joining Imperial College he worked for Hewlett Packard, Scotland. His research interests include VLSI architectures for signal processing, asynchronous systems, reconfigurable computing using FPGAs, and architectural synthesis.

Prof. Cheung was elected as one of the first Imperial College Teaching Fellows in 1994 in recognition of his innovation in teaching.