Pipeline Vectorization for Reconfigurable Systems*

Markus Weinhardt and Wayne Luk
Department of Computing, Imperial College, London, UK
{mw8, wl}@doc.ic.ac.uk

Abstract

This paper presents pipeline vectorization, a method for synthesizing hardware pipelines in reconfigurable systems based on software vectorizing compilers. The method improves efficiency and ease of development of reconfigurable designs, particularly for users with little electronics design experience. We propose several loop transformations to customize pipelines to meet hardware resource constraints, while maximizing available parallelism. For run-time reconfigurable systems, we apply hardware specialization to increase circuit utilization. Our approach is especially effective for highly repetitive computations in DSP and multimedia applications. Case studies using FPGA-based platforms are presented to demonstrate the benefits of our approach and to evaluate trade-offs between alternative implementations. The loop tiling transformation, for instance, has been found to improve performance by 30 to 40 times above a PC-based software implementation, depending on whether run-time reconfiguration is used.

1 Introduction

Many application developers recognize that the key to effective use of reconfigurable systems is to maximize their available parallelism. This task, which has to be achieved while meeting specific hardware resource constraints, is difficult to perform by hand.

Vectorizing compilers have proved successful in detecting and exploiting parallelism for conventional processors with a fixed architecture. A vector execution unit adapted for DSP and multimedia processing has also been identified as an important component of novel computer architectures, such as the vector IRAM [1]. This paper presents an approach for automatically producing optimized pipelined circuits from a high-level program, using techniques derived from software vectorizing compilers. The compile-time and run-time reconfigurability of FPGAs can also be efficiently exploited.

Our approach, which we call *pipeline vectorization*, involves essentially the synthesis of pipelined coprocessors which execute inner loops of programs. Data dependence analysis similar to software vectorization is performed, which determines if a pipeline can be generated for a loop. In contrast to software vectorization, we do not explicitly generate vector instructions. Instead, all instructions of the loop body are vectorized and chained by pipelining input data through the entire dataflow graph synthesized from the loop body.

2 Overview and Related Work

This paper is organized as follows. Section 3 presents the pipeline vectorization design flow and its core components. Next, we devise several loop transformations which widen the applicability of our technique by adjusting the amount of hardware used in vectorized loops to the given FPGA resources in Section 4. In Section 5, we explore methods to increase pipeline circuit utilization by run-time circuit specialization and run-time reconfiguration. Finally we provide case studies evaluating these new optimizations in Section 6, and conclude the paper in Section 7.

An earlier version of our approach, which covers some of the ideas in Section 3, has been reported [2]. Some restrictions in that work have been overcome in our current approach.

Several research projects address the synthesis of pipelined circuits from program loops. The closest to our approach is the NAPA C compiler [3]. However, it targets specifically the NAPA1000 chip and considers only innermost loops. No optimizing transformations similar to ours are reported. The RaPiD-B compiler [4], too, is architecture specific for the RaPiD chip and

*This work is supported by a European Union training project financed by the Commission in the TMR programme, the UK Engineering and Physical Sciences Research Council (Grant Number GR/L24366, GR/L54356 and GR/L59658), Embedded Solutions Ltd., and Xilinx Inc.

requires manual parallelisation and partitioning. Our techniques can be used as a frontend for RaPiD-B. Loop parallelization based on the ALPHA system [5] is restricted to linear systolic arrays, whereas our techniques can vectorize more general programs.

3 Pipeline Vectorization

Figure 1 shows the pipeline vectorization design flow. We first present the core components on the direct path III from an input program to an executable application. Later sections will cover the optimizing transformations (paths I and II in Figure 1).

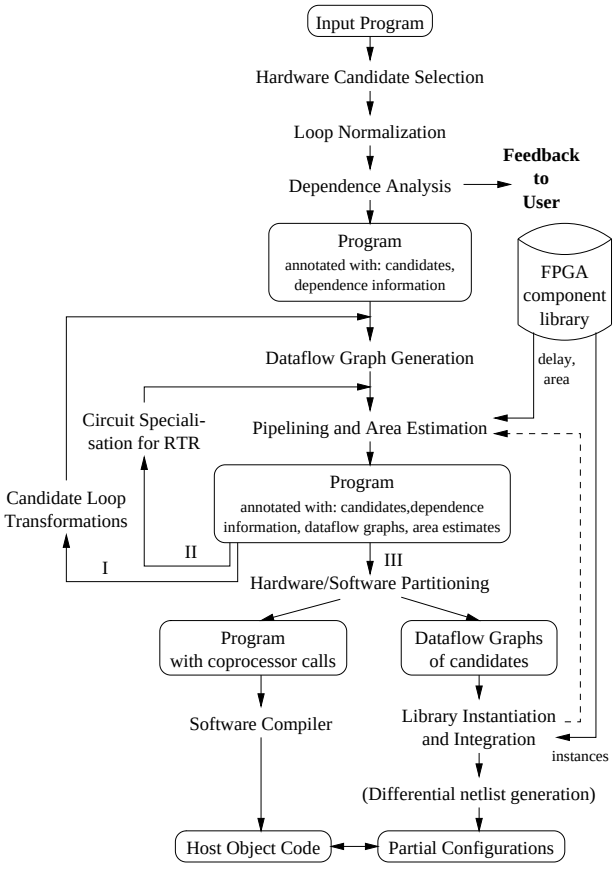

Figure 1: Pipeline vectorization.

Hardware Candidate Selection Regular, iterative computations which perform identical operations on a large set of data are likely to benefit from hardware acceleration. Hence loops are natural candidates for hardware coprocessors. We only generate pipelines for innermost loops, since outer loops have a smaller speedup potential and require more complicated control circuitry, and are better handled in software. However, our procedure will initially consider all loops since the loop transformations presented in Section 4 rearrange loop nests. We concentrate on FOR-loops for vectorization, since efficient pipelined coprocessors cannot be synthesized for general WHILE-loops. FOR-loops have predetermined loop counts and can thus be handled by efficient control circuitry.

There are some additional restrictions for the candidate loops: they must not contain recursive function calls, external operating system or library calls.[1]

The candidate loop of the example program in Figure 2 will be used to synthesize a pipeline circuit.

```
unsigned short x[N];
...
unsigned short rand = 0x1; /* 16-bit */
for(i=0; i<=N-2; i++) { /* CANDIDATE */
   if (rand >> 15)  /* bit 15 is set */
      rand = (rand << 1) ^ 0x7549;
   else
      rand = rand << 1;
   x[i] = x[i+1] + x[i+2] + rand; }
```

Figure 2: Example program.

Loop Normalization For vectorization, we *normalize* the candidate loops by the following transformations [6]. First, we remove all additional induction variables and normalize the loop's lower bound to zero and its step to one. Next, the index expressions are reduced to linear expressions of the induction variable if possible. Note that the candidate loop in Figure 2 is already normalized.

Dependence Analysis The next processing step analyzes candidate loops for dependences. In a loop nest, we determine for each loop hierarchy the dependences carried by each loop: dependences between statements in different iterations of this loop. Only these *loop-carried dependences* affect the loop level parallelism. However, since the pipeline execution overlaps the loop iterations but maintains their order, memory writes are never out of order. Hence we only have to consider *true* dependences, but not *anti-* or *output-dependences*.[2] Therefore pipeline vectoriza-

[1] Non-recursive function calls can be inlined. Therefore we assume — without loss of generality — that no function calls exist in the candidates.

[2] *True* or *flow dependence* occurs when a variable is assigned or defined in one statement and used in a subsequently executed statement. *Anti-dependence* occurs when a variable is used in one statement and reassigned in a subsequently executed state-

tion applies to more loops than software vectorization. We utilize standard dependence analysis methods [6] to detect these dependences. As in software vectorization [6], only array index expressions linear in the induction variable can be analyzed. Other cases, especially indirect array accesses, are assumed dependent.

Next, we check if the detected true loop-carried dependences occur in all loop iterations with the same dependence distance. We call these dependences *regular*. All dependences stemming from scalar variables and from array accesses with the same stride are regular. They can be implemented by feedback paths in the circuit. In contrast to software vectorization, regular dependences do not prevent pipeline synthesis, although they reduce parallelism because the feedback paths restrict the speedup achieved by pipelining in a later processing stage.

Irregular dependences can be handled, provided that the original order of read and write accesses of the arrays involved are maintained. However, this requires many sequential memory accesses and is only feasible with very fast memories, such as on-chip memories. Since this extension requires synthesis of an additional controller for states within a pipeline cycle, it is not supported by our current prototype.

In Figure 2, there is a dependence stemming from the scalar variable `rand`. In every iteration, its value from the previous iteration is read. Since this dependence is regular, it can be realized by a sequential feedback path, resulting in a linear feedback shift register generating random numbers (see Figure 3). Additionally, there are two more loop-carried dependences stemming from the assignment to array `x`. Since they are anti-dependences (only an out-of-order execution of the assignments would lead to real dependences), we can disregard them. Consequently, the loop in Figure 2 can be pipeline vectorized though it could not be executed on a parallel or vector computer.

Dataflow Graph Generation For those candidate loops which pass the dependence test, pipeline circuits are synthesized. We employ a simple storage allocation scheme: scalar variables are held in FPGA registers, and arrays are stored on off-chip memory.[3] Array elements are fed to the pipeline as continuous data streams through vector inputs, and output streams are written back to local memory through vector outputs. In this way one loop iteration is executed every pipeline cycle. Element addresses for linear array accesses can be precomputed. However, for arbitrary accesses, address computation logic must be generated and synchronized with the local memory. Pointer accesses are indirect accesses to the entire host memory space and could be treated similarly. Since most reconfigurable systems do not have direct access to host memory, we do not handle pointers currently. When targeting tightly-coupled architectures with direct memory access, this restriction is not necessary.

Synthesis starts with generating an acyclic combinational *dataflow graph* (DG) for the loop body by using multiplexers to select the correct values of conditionally assigned variables. We treat array accesses and scalar variables uniformly. Index-shifted accesses to the same array are then combined and realized by shift registers. Using these delayed values of the input stream avoids accessing the same value in memory more than once and reduces the number of required vector inputs. This reduction is crucial since all vector input streams must be read and all output streams written once for every loop iteration. Thus the pipeline throughput directly depends on the number of vector inputs and outputs.

Figure 3 shows the DG for the example program from Figure 2. There is a vector input and output for `x`, and a scalar input and output for `rand`. Registers are represented by delay elements `D`. Note that both branches of the `if`-statement are implemented and the correct value for `rand` is selected by a multiplexer. The condition is evaluated by selecting bit 15 of the input register. The purpose of the multiplexer on the left-hand side will be explained next.

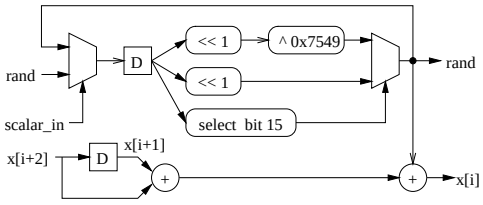

Figure 3: Dataflow graph for Figure 2.

If a loop has regular loop-carried dependences, corresponding *feedback cycles* need to be inserted in its DG. This is simple for dependences stemming from scalar variables: since one loop iteration is executed every clock cycle, the input register of such a variable (which is read *and* written in the loop) must always contain the value computed in the previous clock cycle. To achieve this, a multiplexer is added at the register's

ment. *Output dependence* occurs when a variable is assigned in one statement and reassigned in a subsequently executed statement [6].

[3] On some FPGA families, small arrays of data can be stored in very fast on-chip memory. However, this requires synthesis of access logic for these arrays.

input. It selects the input value during initialization and the feedback value during normal operation, depending on an external control signal (`scalar_in` in Figure 3) which is provided by the environment.

Dependences stemming from array accesses are treated similarly. An output value, however, cannot be directly fed back into an input register if its dependence distance is greater than one. In this case additional registers are inserted to account for the greater delay. Figure 3 shows a feedback path for the dependent variable `rand` and a multiplexer selecting the correct input.

Pipelining and Area Estimation The DGs generated so far may not be very efficient because the combinational delays of chained operators may accumulate to a long critical path. The critical path delay can be reduced by pipelining, thereby improving the performance. Although the latency is also increased, it often has only a minimal effect since the time for filling and flushing the pipeline is normally negligible.

We use a standard retiming technique [7] to insert the minimal number of FPGA flipflops necessary to achieve the cycle time determined by the given I/O bandwidth. The technique is extended to take into account that in many FPGAs combinational gate outputs can be latched in the same cell. We use an FPGA technology specific component library to determine operators' delays. The components are parametrized by operator bitwidth to provide accurate estimates. However, routing delays cannot be estimated accurately. The same component library is used to estimate the pipeline's area (or resource usage) by summing up the area used by all components. These estimates are used in the partitioning step described later.

These techniques will reduce the critical path of the dataflow graph in Figure 3 by inserting additional pipeline registers at the inputs of the adder on the right side.

Hardware/Software Partitioning Partitioning determines which part of a program will be executed in software and hardware. If loops are to be executed in hardware, their DGs' area estimations must not exceed the given hardware resources. Of course, partitioning must also consider the expected speedup achieved by the coprocessor. This estimation problem has been addressed elsewhere [2, 8] and is not the subject of this paper. Partitioning extensions related to the optimizing transformations will be covered in the respective sections.

However, automatic partitioning is not always desirable. The user might want to influence the result. Therefore, our prototype compiler produces explanations about whether a loop is a hardware candidate or not. Hence the user can change the program accordingly, for instance by substituting floating point with fixed point data, or by eliminating dependences. For the candidates, area and speed estimations are given as well. Thus an experienced user can assess the chances of improving the generated circuit manually and partition the program himself.

The program running on the host is generated by substituting the chosen loops by runtime library calls for configuring and executing the pipeline as well as copying data between host and coprocessor. They reconfigure the FPGAs if a new coprocessor is needed.

Library Instantiation and Integration The DGs are transformed into an FPGA specific netlist by instantiating all operators with macros from the component library also used for estimation. These netlists have to be combined with control circuitry which provides access to the host and local memory, and clocks the circuit. They are then further processed with off-the-shelf vendor tools, resulting in a configuration bitstream. Differential netlist generation applies only for partially reconfigurable systems, see Section 5. In case the implemented circuit does not meet the estimated area or delay targets, the dotted design flow cycle in Figure 1 back-annotates the DGs with accurate values, and the subsequent steps are repeated. For instance, more pipeline stages are inserted to reduce the delay. Alternatively, an experienced user can review and optimize the generated circuits manually.

The control circuitry can be a *Pipeline Control Unit* [2, 8, 9] which is initialized by the host, but accesses vector data from local memory and runs the pipeline without host interaction. The length of a pipeline cycle is determined by the number of memory accesses per cycle, given by the number of vector inputs and outputs. Alternatively, the pipeline can directly communicate with the host or with external data sources and sinks. Pipeline vectorization is not restricted to a specific system and interface architecture.

4 Loop Transformations

The core design flow discussed so far is limited to programs with suitable innermost loops. If the loop body is too small to warrant the hardware overheads or too large to fit in the given hardware, no copro-

cessor can be synthesized. This section shows how loop unrolling, loop tiling, loop merging, loop distribution, and loop interchange — transformation techniques known from parallelizing software compilers — can be adapted to overcome these problems and widen the applicability of pipeline vectorization. Since the transformations naturally involve the part of the application remaining in software, they are more systematic and comprehensive than just optimizing the hardware parts *after* partitioning and hardware generation. We apply unrolling and tiling wherever possible since they influence the resulting performance significantly. The other transformations are only used under specific circumstances.

The transformations generate new variations of the candidates and add them to the internal program representation. Then DG generation, pipelining and area estimation are repeated for the new loops (see path I in Figure 1). Finally, the best suited among the original and alternative coprocessors are implemented (path III, Figure 1). Since the transformations only manipulate the internal program and high-level DG representations, all interesting alternatives can be generated quickly. Only the implementation of the selected coprocessors involves running slow hardware design tools, such as place and route tools.

Loop Unrolling In software compilers, loop unrolling is an important technique to increase basic block sizes, extending the scope of local optimizations. Unrolling inner loops results in larger loop bodies. For pipeline vectorization, this means larger coprocessors and therefore more potential parallelism. However, the size of the coprocessors must match the available FPGA resources.

We gain the most by completely unrolling a candidate loop. This is possible if its bounds are constant. This situation occurs in many programs, for instance in image processing applications with loops over small, constant-size templates [10]. Specific examples include the skeletonization program used in Section 6, or filters with a constant number of taps.

The inner loop is completely removed so that the outer loop can be vectorized. Since the new loop body might be too large for the given FPGA resources, or the coprocessor might be too slow due to too many vector inputs and outputs, unrolling might not lead to a feasible pipeline coprocessor for the outer loop. Therefore partitioning will decide if the original or the unrolled candidate is selected.

Let us now consider partial unrolling. If an inner loop is only partially unrolled, the next outer loop cannot be vectorized and remains in software. This means that unrolling n iterations also increases the number of vector inputs and outputs and the length of a pipeline cycle n-fold, thus annihilating the speedup gained by fewer loop iterations.[4] Hence partial loop unrolling is not useful for pipeline vectorization. However, it would be useful if the outer loop was vectorized. This is achieved by loop tiling.

Loop Tiling Loop tiling is an alternative transformation for cases where complete unrolling is not applicable due to variable loop bounds or resulting coprocessors becoming too large. In these cases it is very beneficial to *partially* unroll a loop, thereby adjusting the circuit size to the given hardware resources, *and* vectorize the next outer loop. Loop tiling achieves this by combining loop partitioning and interchange. We adjust this technique for pipeline vectorization.

Transformation steps (1) and (2) in Figure 4 show loop tiling in the general form used here. The transformation works on two nested loops where `PRE(i)` and `POST(i)` do not contain loops themselves. The inner loop is partitioned in *tiles* which will eventually be unrolled. The tile size *tsize* is chosen as the maximum number of "processing elements" (instances of the loop body `F(i,j)`) fitting in the given hardware resources along with the operations in `PRE(i)` and `POST(i)` which are executed before the first tile and after the last tile, respectively. Hence *tsize* is estimated as $tsize = (area_{HW} - area_{PRE} - area_{POST})/area_F$ where $area_{HW}$ is the size of the hardware resources, $area_{PRE}$, $area_{POST}$ and $area_F$ are the estimated sizes of `PRE(i)`, `POST(i)` and `F(i,j)`, and `/` denotes integer division. Loop tiling will then result in a coprocessor which is approximately *tsize* times larger and *tsize* times faster than the coprocessor generated from the original loop.

Transformation step (1) partitions the loop for a given `tsize` and renormalizes the bounds and steps. Rather than unrolling the inner loop, step (2) interchanges the outer loop with the tile loop. This allows us to vectorize the former outer `i`-loop and to unroll the reduced inner `j`-loop without considering the (now outermost) tile loop. `PRE(i)` and `POST(i)` were first "sunk" in the tile loop (by adding guards) since interchange is only possible for perfectly nested loops (with no statements between the inner and outer loop).

However, loop tiling is not possible if the bounds of the inner loop depend on the outer loop index or if

[4] Note that this is not automatically the case for completely unrolled loops since the outer loop (with another index variable) is vectorized.

```
 1  for (i=0; i<M; i++) {
 2    PRE(i);
 3    for (j=0; j<n; j++)
 4      F(i,j);
 5    POST(i); }
```

```
 1  for (i=0; i<M; i++) {
 2    PRE(i);
⇒3    for (jt=0; jt<(n-1)/tsize+1; jt++)
(1)4      for (j=0; j<min(tsize,n-jt*tsize); j++)
 5        F(i,j+jt*tsize);
 6    POST(i); }
```

```
 1  for (jt=0; jt<(n-1)/tsize+1; jt++)
 2    for (i=0; i<M; i++) {
 3      if (jt==0) /* first tile */
⇒4         PRE(i);
(2)5      for (j=0; j<min(tsize,n-jt*tsize); j++)
 6        F(i,j+jt*tsize);
 7      if (jt==(n-1)/tsize) /* last tile */
 8        POST(i); }
```

```
 1  for (jt=0; jt<(n-1)/tsize+1; jt++) {
 2    for (i=0; i<M; i++) {
 3      if (jt==0) /* first tile */
⇒4         PRE(i);
(3)5      for (j=0; j<tsize; j++)
 6        if (jt!=(n-1)/tsize || j<(n-1)%tsize+1)
 7          F(i,j+jt*tsize);
 8      if (jt==(n-1)/tsize) /* last tile */
 9        POST(i); } }
```

```
 1  for (jt=0; jt<(n-1)/tsize+1; jt++) {
 2    for (i=0; i<M; i++) {
 3      if (jt==0) /* first tile */
 4         PRE(i);
 5      F(i,jt*tsize); /* no guard necessary */
⇒6      if (jt!=(n-1)/tsize || 1<(n-1)%tsize+1)
(4)7        F(i,1+jt*tsize);
 8      ...
 9      if (jt!=(n-1)/tsize ||
10          tsize-1<(n-1)%tsize+1)
11        F(i,tsize-1+jt*tsize);
12      if (jt==(n-1)/tsize) /* last tile */
13        POST(i); } }
```

```
 1  guard_1 = 1<(n-1)%tsize+1;
 2  guard_2 = 2<(n-1)%tsize+1;
 3  ...
 4  guard_last = tsize-1<(n-1)%tsize+1;
 5  for (jt=0; jt<(n-1)/tsize+1; jt++) {
 6    first_tile = jt==0;
 7    last_tile  = jt==(n-1)/tsize;
 8    for (i=0; i<M; i++) {
⇒9      if (first_tile)
(5)10        PRE(i);
11      F(i,jt*tsize); /* no guard necessary */
12      if (!last_tile || guard_1)
13        F(i,1+jt*tsize);
14      ...
15      if (!last_tile || guard_last)
16        F(i,tsize-1+jt*tsize);
17      if (last_tile)
18        POST(i); } }
```

Figure 4: Hardware specific loop tiling.

data dependences prevent the loop interchange. Fortunately this can be checked before starting the entire transformation since step (2) is legal iff the original loops are *fully permutable* [11]. This is the case if all dependences carried by these loops have non-negative distances. This condition can be tested during dependence analysis. It means that no dependence on an earlier iteration of the inner loop is allowed. In terms of the generated pipelines, no backward dataflow between "processing elements" is allowed, but non-local forward flow is.

The output of step (2) cannot directly be vectorized. Thus we devise additional hardware specific transformations extending software loop-tiling. The non-constant upper bound $min(tsize, n - jt \times tsize)$ prevents unrolling the inner loop. Since $tsize$ is constant, the upper bound can never be larger, and we substitute it by $tsize$. To maintain correctness, the loop body `F` has to be guarded by $j < n - jt \times tsize$ for the case that $n - jt \times tsize$ is the actual minimum. This guard can only be wrong for the last tile $jt = (n-1)/tsize$, so we can rewrite it to

$$jt \neq (n-1)/tsize \vee j < (n-1)\%tsize + 1$$

where % denotes the modulo operator. Step (3) shows this transformation.

Now the inner loop can be unrolled in step (4). Unfortunately the guards have to be replicated, too.[5] Implementing them in hardware would increase the pipeline area considerably. Fortunately this is not necessary, since their values do not depend on the index variable `i`. We can assign flags outside the vectorized loop (in software) and pass them to the hardware. Step (5) in Figure 4 shows this final transformation. Note that `guard_1` to `guard_last` need only be computed once since they do not change in the tile loop, whereas `first_tile` and `last_tile` need to be adjusted in the tile loop. The resulting program generates a DG adjusted to the given hardware resources. An example will be given in Section 6.

Loop Merging Loop merging is another means of increasing parallelism in loop bodies. Its scope is, however, rather limited since loops (or loop nests) must traverse exactly the same index space to allow merging. Moreover, all dependences of the original loops must be preserved in the merged loop. A simple example is given in Figure 5. Merging is legal if `F` and `G` only depend on `p1[v][h]` and `p2[v][h]` respectively. However, a more realistic program where `F` and `G` are

[5] Only for $j = 0$, the condition is always true because every tile performs at least one inner loop iteration. Hence it can be omitted.

for instance linear image processing operators depending on a 3×3 neighborhood[6] has dependences which prevent direct merging. This is because new `p2` values have to be computed for the entire neighborhood before a new `p3` value can be computed. Fortunately this dependence does not mean the loops cannot overlap; they can if the second operator only starts when the first has finished computing the first row. We develop a new transformation which merges loops systematically even if there are local dependences like these.

```
for (v=1; v<vlen-2; v++)
 for (h=1; h<hlen-2; h++)
  p2[v][h] = F(p1); /*A*/
for (v=1; v<vlen-2; v++)
 for (h=1; h<hlen-2; h++)
  p3[v][h] = G(p2); /*B*/
```

$\Rightarrow$

```
for (v=1; v<vlen-2; v++)
 for (h=1; h<hlen-2; h++)
 {
  p2[v][h] = F(p1); /*A*/
  p3[v][h] = G(p2); /*B*/
 }
```

Figure 5: Loop merging.

Our strategy is to delay the second loop body B (with respect to the outermost loop) by d iterations where d is large enough to preserve all dependences. d is determined by directly merging the loops and checking for *anti-dependences* carried by the outermost loop from a statement in B to a statement in A. They indicate a dependence in the original loops from iteration i in A to iteration j in B with $j < i$. This dependence has been violated by merging. We determine d as the largest occurring anti-dependence distance. For instance, in Figure 5, a right-hand-side access to `p2[v-1][h]` in B and a left-hand-side access to `p2[v][h]` in A establish an anti-dependence with distance 1. If no anti-dependences exist, $d = 0$ and direct merging is possible. However, if irregular anti-dependences exist (no distance can be determined), merging is not possible.

Let us consider the case $d > 0$. We can delay loop body B by subtracting d from all occurrences of the outer loop index in B. Additionally, the upper bound of the outer loop has to be increased by d to execute the delayed iterations on B. A and B have to be guarded so that the first d iterations are only executed by A and the last d iterations only by B. Figure 6 shows the result for the operators depending on a 3×3 neighborhood. In this case $d = 1$; Figure 7 (a) shows the corresponding DG. We see that the merged pipeline requires five vector inputs and two outputs. This might slowdown the pipeline considerably and make merging not worthwhile. It must be checked by the final coprocessor selection in the hardware/software partitioning phase. On the other hand, we can efficiently implement these vector inputs and outputs on architectures with several concurrently accessible memory banks by allocating `p1`, `p2` and `p3` to different banks. We discuss a detailed case study on this in Section 6.

[6] That is, `p2[v][h] = F(p1[v-1][h-1], p1[v-1][h], ..., p1[v+1][h], p1[v+1][h+1]);` and analogous for `G`.

```
for (v=1; v < vlen-2+d; v++)
  for (h=1; h < hlen-2; h++) {
    if (v < vlen-2)                                  /* A */
      p2[v][h] = F(p1[v-1], p1[v], p1[v+1]);
    if (v >= 1+d)                                    /* B */
      p3[v-d][h] = G(p2[v-1-d], p2[v-d], p2[v+1-d]); }
```

Figure 6: Generalized loop merging result.

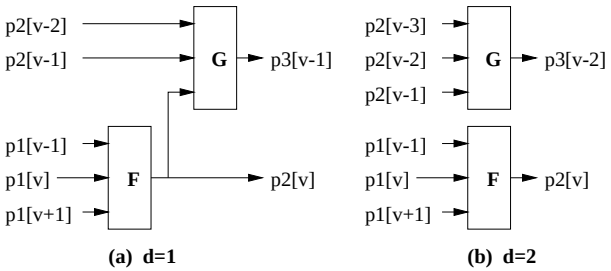

Figure 7: Dataflow graphs for merged loop.

By slightly changing this transformation, it becomes suitable for multi-FPGA systems. Figure 7 (b) shows the resulting DG if we "overdelay" B by one ($d = 2$ in this example). In this case, the pipelines of the original loop bodies become completely independent and communicate only via memory. Hence they can easily be allocated to separate FPGAs which share access to a memory bank for array `p2`. In this case we do not really merge the loops but determine how two (or more) pipelines can overlap forming a composite pipeline. For n pipelines, a speedup factor up to n can be achieved compared to sequential execution. This would not be possible without the analysis information. The minimal or overdelayed d-value is chosen depending on the given architecture.

Other Loop Transformations *Loop distribution* is the opposite of loop merging. It results in smaller pipelines and thus can be applied if a loop body is too large to fit on the given hardware. A loop cannot be distributed if dependences of the original loop are violated. As in loop tiling — which is a form of loop distribution — pipeline feedback paths must not be cut. *Loop interchange* swaps perfectly nested loops. As discussed for loop tiling, it is legal if the interchanged loops are fully permutable. This trans-

formation does not change the size of the generated hardware, but can increase the length of the vectorized loop, thereby reducing the overhead for setting up, filling and flushing the pipeline. Furthermore, it can increase the locality of data accesses by changing the index variable relevant for vectorization. This is necessary for some memory access models. Finally, *strip mining* (the first step of loop tiling) can reduce local memory requirements if combined with array region analysis and applied to the vectorized loop.

We do not attempt to transform entire loop nests as in [11] since it is difficult to define a strategy for such a global transformation in the context of pipeline vectorization. This is an area of future research. In addition, our compiler provides feedback to the user to allow manual improvements of the program. Changing an entire loop nest would make the compiler less understandable and predictable, and would therefore limit the ability of the user to improve the program.

Partitioning Extensions Automatic hardware/software partitioning is extended by a recursive algorithm which selects the transformed loop which results in the largest feasible coprocessor. Alternatively, the user selects the applied transformations. He can also select parameters as the tile size. This is especially useful if the area targets are not met and the dotted design flow cycle in Figure 1 is activated.

5 Run-time Circuit Specialization

Constant propagation has long been used in software and hardware compilers to optimize programs or circuit designs. The advent of reconfigurable hardware has opened the opportunity to propagate values which are not constant, thereby reducing a design's delay and area [12]. Whenever a value changes, the circuit is reconfigured. Rather than changing the input of flexible operators, a design which exploits run-time reconfiguration (RTR) uses smaller operators obtained by constant propagation. Hence more of a program's operators can be implemented on a given hardware area. Because of the reconfiguration overhead, only values changing infrequently should be considered. We therefore only consider those variables for value propagation which do not change inside the loops to be vectorized. The hardware/software partitioning must evaluate the trade-off between design improvement and reconfiguration overhead.

We distinguish two cases of RTR. First, the number of propagated values is limited and the values themselves are known at compile time. Second, there is an arbitrary number of values unknown at compile time. We present methods for exploiting these cases for pipeline vectorization next.

Limited Value Propagation If the number of possible values is limited, the hardware candidate can be reproduced for all values. Consider the transformation of the example in Figure 8. The program is a string pattern matcher where PM(x, i, pat) computes a boolean value indicating if the input string `x` contains the pattern `pat` at position `i`. The original version uses the variable input `pat` in the FOR-loop. By standard definition-use analysis, the conditional assignment to `pat` can be propagated to its use in the FOR-loop (step (1) in Figure 8). The next step (2) moves the evaluation of `sel` out of the FOR-loop. The loop is duplicated, but each instance now has a constant input to `PM` which results in smaller and faster hardware. This transformation can easily be extended for more than two values or more than one variable being considered. It performs constant propagation in software and effectively produces several independent loops. Standard hardware generation is applicable, and the design flow path I in Figure 1 is used. As with the other loop transformations the original program code is retained, since only the partitioning phase decides if the propagated version will be used.

```
1  pat = (sel) ? "new" : "not";
2  ...
3  for (i=0; i<N-2; i++)
4    y[i] = PM(x, i, pat);

⇒  1  for (i=0; i<N-2; i++)
(1)2    y[i] = PM(x, i, ((sel) ? "new" : "not"));

1  if (sel)
2    for (i=0; i<N-2; i++)
⇒  3      y[i] = PM(x,i,"new");
(2)4  else
5    for (i=0; i<N-2; i++)
6      y[i] = PM(x,i,"not");
```

Figure 8: Limited value propagation.

We can also generate independent loops for tiled loops if the tiling is necessary due to limited hardware resources, while the inner loop length (and therefore the number of tiles) stays constant. Unrolling the tile loop (which is the outermost loop considered) generates an independent vectorizable loop for every tile with constant values for `jt` and for all guards (cf. Figure 4). Note however that the tiling transformation should be repeated if RTR is considered since value propagation reduces the area of a "processing

element". Hence more elements fit on the available hardware, and the tile size can be increased.

This case of RTR is suitable for chip-level and partially reconfigurable systems. However, the trade-offs will be different. If partial reconfiguration is not supported, the reconfiguration time will be large, regardless of how small the difference between two configurations is. So chip-level RTR will not be useful for examples like the pattern matcher in Figure 8 where only three comparators can be simplified. The gain will be negligible compared with the reconfiguration overhead.

On the other hand, for partially reconfigurable devices the reconfiguration time is proportional to the amount of logic altered. We use tools like *ConfigDiff* [13] to determine the fastest partial configuration to switch between two similar designs. Hence small changes can be performed very quickly.

Arbitrary Value Propagation The second case of run-time reconfiguration occurs if a variable can assume any value at run-time. Then we cannot prepare separate configurations for each of them at compile time. Since it is prohibitive to run the entire design tool suite for new values at run-time, this case cannot be handled with FPGAs which can only be configured completely. It is only suitable for partially reconfigurable FPGAs which allow to adapt an operator to any constant input values within a few cycles at run-time. Therefore a circuit "skeleton" is synthesized which reserves area for the largest possible constant input operator. At run-time all these operators are adapted to the given values. Doing this also requires a special component library which provides the operator "skeletons" along with information on how to generate the configuration instructions for a given input value and a given position of the operator on the chip.

Generating such a circuit "skeleton" adds an alternative implementation for a given hardware candidate, but the candidate loop itself remains unchanged. Since the constant input operators have smaller delays than their flexible counterparts, their pipelined versions might contain less registers. Therefore pipelining and area estimation — but not DG generation — is repeated for these new implementations (path II in Figure 1). As for limited value propagation, the tile size for partially unrolled loops is increased. Thus tiling should be repeated.

This is the most flexible approach to RTR. Unfortunately, generating such designs has not yet been completely automatized. However, we present a manually implemented case study in Section 6.

RTR Partitioning and Integration In RTR systems, the original or the specialized circuit must be selected automatically (unless only the specialized circuit fits on the given hardware). There is a trade-off between the reconfiguration time and the amount of computation performed in one configuration. The reconfiguration time depends on the FPGA technology (partial or complete reconfiguration) and on the reconfiguration frequency. The latter depends on the overall control flow of the program. Its analysis involves estimating loop and branch execution counts and must be addressed in the context of the overall speedup estimation, cf. [2, 8]. Alternatively, an implementation can be selected manually.

For partially reconfigurable systems, differential netlists can be generated. This additional step replaces complete configurations by differential configurations which just change the differences between two consecutive configurations. Thereby even the configuration times of unrelated coprocessors are reduced, especially if they share the same control circuitry.

6 Implementation and Case Studies

We have implemented a simple version of the core design flow in the *Modula Pipeline Compiler* prototype [8]. Here we present case studies which demonstrate the new techniques presented in this paper. The results have been produced with the assistance of a prototype compiler based on the SUIF framework [14], which provides C and Fortran frontends, and powerful loop analysis and transformation libraries.

String Pattern Matcher This case study evaluates the benefits of loop tiling and run-time circuit specialization. We implement a string pattern matcher on a PC-based Xilinx 6200DS board using a XC6216 FPGA. This program, shown in Figure 9, is the same as that in Figure 8, but with arbitrary pattern lengths and values. Therefore the inner loop cannot be unrolled. However, the inner loop can be vectorized and the tiling transformation can be applied. The resulting pipeline circuit is a linear datapath of comparators and registers. Both compile-time reconfigurable (CTR) and run-time reconfigurable (RTR) versions are possible. The CTR version contains generic comparators and the XC6200's protected registers so that pattern bytes can be loaded directly from the host, whereas the specialized RTR version contains constant comparators. The pipelines have been placed automatically. The XC6216 is large enough to implement

```
for (i=0; i<N-P+1; i++) {
  y[i] = 1;
  for (j = 0; j<P; j++)
    if (pat[j] != x[i+j])
      y[i] = 0; }
```

Figure 9: String pattern matcher program.

the Pipeline Control Unit (about 25% of the chip area) and 54 CTR processing elements or 90 smaller, specialized RTR processing elements.

	Soft-ware	Inner loop vectorizat.	Tiled vect. CTR	Tiled vect. RTR
Performance	24.8	12.5	671	1,032
Speedup	—	0.5	27	42

Table 1: Analysis of string pattern matcher: raw performance in 10^6 comparisons/s and speedup over software.

Table 1 shows the raw performance of the implementations, including speedups over software on a 300 MHz Pentium II PC. All values are actual measurements, except those related to inner loop vectorization which are estimated. The values for the tiled implementations include the times for changing a tile, amortized over 100,000 pipeline cycles. The pipeline cycle is 80 ns for all circuits.[7]

However, the hardware performance data do not include the overheads for initialising the FPGA configuration and data transfer since their significance depends on the overall number of tiles. The CTR and RTR performance numbers only concern the case when all processing elements are used. Figure 10 shows the overall execution times including configuration and data transfer times, which are indicated by two additional lines in the graph. Since the execution time of a tiled implementation only depends on the number of tiles, their graphs are step functions.

We conclude that loop tiling is a transformation which *enables* a considerable speedup for string pattern matching in the first place, and run-time reconfiguration further *improves* the performance by approximately 50% for large patterns.

Morphological Skeletonization We now analyze the morphological skeletonization algorithm from [10]. This example evaluates loop unrolling and generalized loop merging. Figure 11 shows the algorithm's

[7]The Pipeline Control Unit can access one local memory word in 40 ns. Therefore a pipeline cycle with two accesses takes 80 ns on our system.

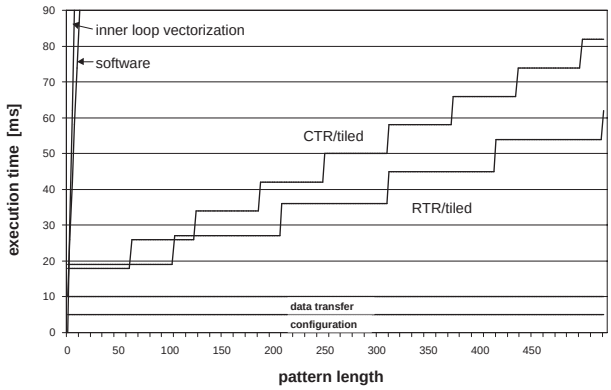

Figure 10: Execution times for string pattern matcher for N = 100,000.

structure. `IMAGE` is initialized with the input image, and `SKELETON` with an empty image. Then the operators erosion, dilation and difference/union are repeatedly performed on the data until `IMAGE` is completely eroded. The dotted arrows indicate which operators' outputs are used for the next repetition.

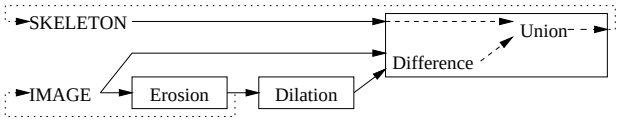

Figure 11: Morphological skeletonization.

The erosion operator consists of two nested inner loops which iterate over a constant 5×5 template. Pipelining the innermost loops would not be beneficial since it only contains one operator computing the minimum of two inputs. However, after completely unrolling both inner loops, a pipeline containing 20 minimum operators can be generated. It can compute one output pixel every pipeline cycle.

The upper part of Table 2 gives raw performance, pipeline frequency and execution time data (for a 512×512 pixel image), as well as the total time for the independent execution of all skeletonization operators, based on 50 ns memory accesses. Dilation is similar to erosion, but the combined difference and union operator loop is not very efficient, since it contains only two operations and no inner loops to unroll. Note that the frequency is higher for architectures with two memory banks, since concurrent read and write accesses are possible.

The performance can be improved by merging all operators to produce one large pipeline. The last line

	1 memory bank			2 memory banks		
	F	P	T	F	P	T
Erosion	3.3	66.7	79	4.0	80.0	66
Dilation	3.3	66.7	79	4.0	80.0	66
Diff./Union	5.0	10.0	52	10.0	20.0	26
Total			210			158
Merged	1.7	70.0	157	3.3	140.0	79

Table 2: Analysis of skeletonization operators: pipeline frequency F in MHz, raw performance P in 10^6 operations/s, execution time T for a 512×512 image in ms.

in Table 2 shows that the advantage of loop merging is limited for one memory bank, since too many memory accesses have to be performed sequentially in one cycle. For two banks, however, merging is effective. It halves the execution time.

We have implemented in the Handel-C language the merged pipeline on an ESL RC1000-PP board [15]. The design, running at 20 MHz, completes one skeletonization iteration for a 512×512 pixel image in 97 ms. Even including data transfer (8 ms for the image data using packed DMA, amortized over 15 to 30 iterations), the hardware was measured to be 11 times faster than software (1,045 ms on the 300 MHz PC).[8]

To summarize, loop unrolling is an *enabling transformation* for the erosion and dilation loops, whereas generalized loop merging further *improves* the entire skeletonization program.

7 Conclusion

This paper presents a framework based on pipeline vectorization for producing optimized pipelined circuits from high-level programs. The framework includes new optimizing transformations which customize hardware coprocessors to meet specific FPGA resource constraints and exploit run-time reconfiguration. The case studies show that some transformations result in hardware acceleration which cannot be achieved easily by hand. Others improve the performance of coprocessors significantly. Our framework can select, generate and integrate coprocessors automatically while retaining the flexibility to allow users to influence the synthesis process. Future work will combine our fine-grain vectorization with coarse-grain, task-level parallelism for large multi-FPGA systems. Strategies to transform entire loop nests will also be studied. We are interested in supporting various input languages, particularly parallel ones, in order to optimize existing parallel programs. Further extensions will allow users to include manually designed hardware blocks and to synthesize digit-serial designs.

[8] The RC1000-PP's Xilinx XC4085XL FPGA also has to be configured once during program execution. Though we utilize only 30%, the chip must be reconfigured completely. This takes 780 ms on our board (despite only 240 ms pure configuration time). We expect much faster configuration for the Virtex chip.

References

[1] C.E. Kozyrakis and D.A. Patterson. A new direction for computer architecture research. *IEEE Computer*, Nov. 1998.

[2] M. Weinhardt. Compilation and pipeline synthesis for reconfigurable architectures. In *Reconfigurable Architectures Workshop RAW'97*, 1997.

[3] M.B. Gokhale and J.M. Stone. NAPA C: compiling for a hybrid RISC/FPGA architecture. In *Proc. FCCM'98*. IEEE Computer Society Press, 1998.

[4] D.C. Cronquist, P. Franklin, S.G. Berg and C. Ebeling. Specifying and compiling applications for RaPiD. In *FCCM'98*. IEEE Computer Society Press, 1998.

[5] E. Fabiani, D. Lavenier and L. Perraudeau. Loop parallelization on a reconfigurable coprocessor. In *Proc. WDTA'98: Workshop on Design, Test and Applications*, Dubrovnik, Croatia, June 1998.

[6] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[7] C.E. Leiserson and J.B. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, 1:41–67, 1983.

[8] M. Weinhardt. *Übersetzungsmethoden für strukturprogrammierbare Rechner (Compilation techniques for structurally programmable computers, in German)*. PhD thesis, Universität Karlsruhe, July 1997.

[9] M. Weinhardt. Portable pipeline synthesis for FCCMs. In *Proc. FPL'96*. Springer, 1996.

[10] H.R. Myler and A.R. Weeks. *Computer Imaging Recipes in C*. P T R Prentice Hall, 1993.

[11] M.E. Wolf and M.S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distr. Systems*, Oct. 1991.

[12] M.J. Wirthlin and B.L. Hutchings. Improving functional density through run-time constant propagation. In *Proc. FPGA'97*. ACM Press, February 1997.

[13] W. Luk, N. Shirazi and P.Y.K. Cheung. Compilation tools for run-time reconfigurable designs. In *Proc. FCCM'97*. IEEE Computer Society Press, 1997.

[14] The Stanford SUIF Compiler Group. Homepage `http://suif.stanford.edu`.

[15] Embedded Solutions Limited. Homepage `http://www.embedded-solutions.ltd.uk`.