Pipeline Vectorization

Markus Weinhardt and Wayne Luk, *Member, IEEE*

***Abstract*—This paper presents pipeline vectorization, a method for synthesizing hardware pipelines based on software vectorizing compilers. The method improves efficiency and ease of development of hardware 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 runtime reconfigurable systems, we apply hardware specialization to increase circuit utilization. Our approach is especially effective for highly repetitive computations in digital signal processor (DSP) and multimedia applications. Case studies using field programmable gate arrays (FPGAs)-based platforms are presented to demonstrate the benefits of our approach and to evaluate tradeoffs between alternative implementations. For instance, the loop-tiling transformation, has been found to improve vectorization performance 30–40 times above a PC-based software implementation, depending on whether runtime reconfiguration (RTR) is used.**

***Index Terms*—High-level synthesis, parallelization, pipelining, reconfigurable computing, vectorization.**

I. Introduction

MANY application developers recognize that the key to effective use of custom computing 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 digital signal processor (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 runtime reconfigurability of field programmable gate arrays (FPGAs) can also be efficiently exploited.

Our approach, which we call *pipeline vectorization* [2]–[4], essentially involves the synthesis of pipelined processors that 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. Therefore, it generates circuits that exhibit more parallelism than many other automatic high-level hardware design tools.

Manuscript received February 29, 1999; revised April 6, 2000. This work was supported by a European Union training project financed by the Commission in the TMR program, the U.K. Engineering and Physical Sciences Research Council, Embedded Solutions Ltd., and Xilinx Inc. This paper was recommended by Associate Editor R. Camposano.

The authors are with the Department of Computing, Imperial College, London SW7 2BZ, U.K. (e-mail: m.weinhardt@computer.org; wl@doc.ic.ac.uk).

Publisher Item Identifier S 0278-0070(01)00943-5.

There are significant differences between pipeline vectorization and software vectorization. For instance, our approach covers a wider range of loops since it does not consider out-of-order execution. It can be used with a variety of storage allocation methods. 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. To widen the applicability of our technique, we devise several loop transformations that adjust the amount of hardware used in vectorized loops to the available hardware resources. For reconfigurable implementations, we explore methods to increase circuit utilization by runtime circuit specialization and runtime reconfiguration (RTR).

Our approach includes the synthesis of nonpipelined circuitry for nonvectorizable loops and conditional and sequential program code. It can be used in two modes—hardware mode and codesign mode. In *hardware mode*, a processor is generated for the entire program (which includes only synthesizable operations as defined in Section III-A1), rendering descriptions from a high-level sequential programming language into an efficient hardware description language (HDL). Alternatively, in *codesign mode*, parts of the program (such as nonsynthesizable or highly irregular parts) remain in software to be executed on a host microprocessor. This mode results in a hardware-software codesign system with data and control transfer between host processor and custom hardware automatically being implemented.

This paper is organized as follows. First, we discuss relevant previous work. Section III then presents the core pipeline vectorization design flow, the main contribution of this paper. Next, Sections IV and V describe the other important contributions: optimizing loop transformations and runtime circuit specialization. Section VI reports on a prototype compiler implementation and Section VII provides case studies and results evaluating pipeline vectorization. Finally, Section VIII presents conclusions and future work.

II. Background

Increasingly, system description is written in a high-level software language [5]. This method simplifies algorithm development and facilitates experiments to map different components into hardware. Our approach not only supports this method, but it goes a step further by providing a framework in which efficient hardware can be produced automatically from a software description. It is particularly suited for reconfigurable systems, which have been shown to be useful in various application domains [6]. We attempt to simplify the programming of these systems, which typically consist of a host processor and FPGA hardware boards [7], [8]. Programming difficulties

include analyzing the tradeoffs between software and hardware, designing software and hardware parts using different languages and tools and debugging the interface between these parts.

Some tools approach these problems by using a programming language input to specify both software and hardware in a uniform manner. This enables systematic analysis of the tradeoffs as well as automatic synthesis of the interface between software and hardware. However, while a sequential program is a natural specification for the host processor, hardware coprocessors synthesized from sequential code often fail to exploit the hardware's parallelism sufficiently. For instance, this is the case in the PRISM system [9]. The configuration and communication overhead is often larger than the achieved speedup itself. Better results can be obtained for systems that integrate a microprocessor core and reconfigurable hardware on a chip. An example of this approach is the Garp chip [10] and its C-based compiler [11].

Guccione adopts *data-parallel* C vector operations on data streams to describe pipelined circuits [12], [13]. However, this method requires the user to learn a new programming language that is only used for the hardware part of an application. The same is true for Transmogrifier C [14], a research compiler allowing only task-level parallelism.

Hardware programming systems based on communicating sequential processes such as OCCAM [15] and Handel-C [16] are suitable for control intensive applications. However, the user has to specify parallel operations explicitly. As for data-parallel C, the software parts of an application must be written in a different language and interfaced manually to the hardware parts.

In the application-specific integrated circuit domain, *high-level synthesis* systems generate register-transfer structures from behavioral (algorithmic) specifications [17]–[19]. These methods employ sophisticated scheduling, resource allocation, and binding techniques for general processor architectures. They perform an exhaustive design-space exploration, which makes the tools very slow, especially if compared to a software compiler. A commercial high-level synthesis system is Synopsys' Behavioral Compiler [20], which can handle array data and generate memory accesses. However, to pipeline loops, the user has to analyze loop-carried dependences manually (as defined in Section III-A3) and specify a safe initiation interval, which preserves the dependences and the original order of memory reads and writes. Another system, C2Verilog, [21] uses ANSI C to produce a Verilog register-transfer structure using high-level synthesis techniques. However, it does not perform loop pipelining.

Several research projects address the automatic synthesis of pipelined circuits from program loops. The closest to our approach is the NAPA C compiler [22]. However, that system targets specifically the NAPA processor [23] and considers only innermost loops. No automatic vectorization or optimizing transformations similar to ours are reported. The scheduling of instructions is performed on an atomic basis and, thus, is less flexible than hardware pipelining, which can also use internally pipelined operators. Finally, a loop parallelization method is also included in the Alpha System [24]. However, it is restricted to producing linear systolic arrays whereas our techniques can vectorize more general programs.

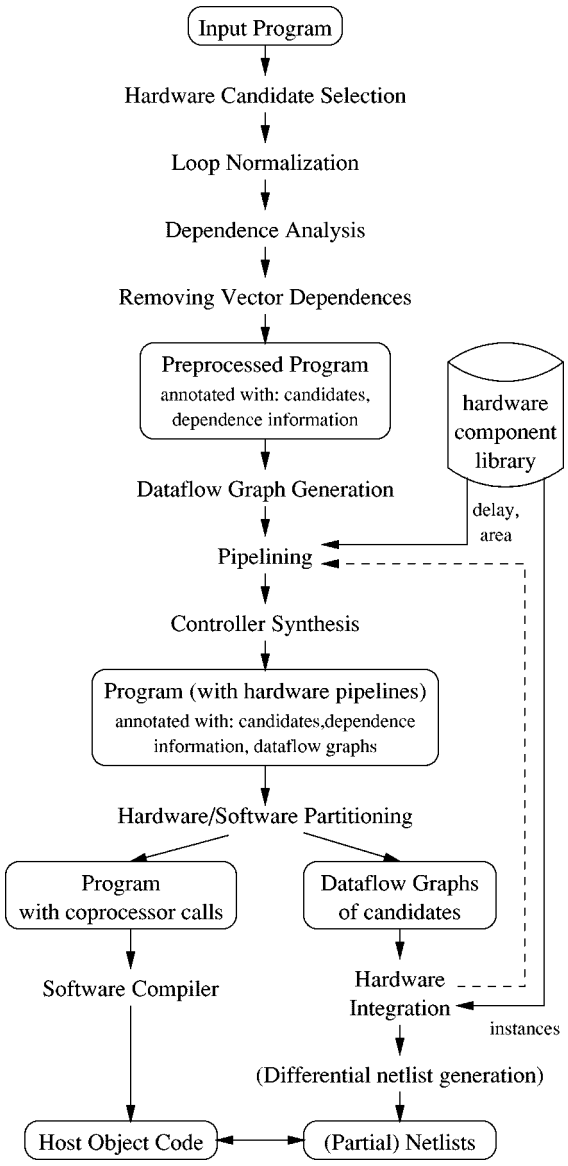

Fig. 1. Core design flow.

III. Core Design Flow

We first present the core pipeline-vectorization design flow, as shown in Fig. 1. Most parts of the figure are relevant for both the codesign and the hardware mode. Only the software branch on the bottom left side and the hardware–software partitioning phase do not exist for the hardware mode. The core design flow consists of three major phases: preprocessing, hardware synthesis, and partitioning and integration. They are described in the following sections. The extension of the core design flow to include loop transformations will be covered in Section IV.

A. Preprocessing

Preprocessing consists of four steps: hardware candidate selection (Section III-A1), loop normalization (Section III-A2),

dependence analysis (Section III-A3), and removing vector dependences (Section III-A4). These steps are necessary to perform hardware synthesis effectively.

1) Hardware Candidate Selection: This step selects program parts suitable for hardware synthesis. Regular iterative computations, which perform identical operations on a large set of data. are likely to achieve high performance in hardware. Hence, loops are natural candidates for hardware processors. We attempt to vectorize innermost FOR loops and generate pipelines for them. FOR loops have an induction (index) variable necessary for normalization and vectorization and have predetermined loop counts. Thus, they can be handled by efficient control circuitry. It is possible to transform some WHILE loops to FOR loops by induction variable detection [25]. Other WHILE loops, outer loops, and other program constructs are nonpipeline candidates and, therefore, are unlikely to result in fast efficient hardware. They should only be considered in combination with pipeline candidates or left for software execution. However, our procedure will consider all loops since the loop transformations presented in Section IV rearrange loop nests.

There are some additional restrictions for the candidates: they must not contain nonsynthesizable operations such as recursive function calls, external operating system calls, or library calls.[1]

In hardware mode, programs containing any nonsynthesizable operations are not considered legal input. Thus, the entire legal input program is a candidate and candidate selection only distinguishes between pipeline and nonpipeline candidates.

The example edge detector program in Fig. 2 will be used to synthesize a pipeline circuit. Its inner loop is a pipeline candidate and its outer loop (by definition) a nonpipeline candidate. We use C language syntax here because our compiler prototype (Section VI) uses a C front end. Our approach is valid for any sequential imperative programming language.

2) Loop Normalization: For vectorization, we have to normalize the pipeline candidate loops by the following transformations.

- Remove all additional induction variables and normalize the loop's lower bound to zero and its step to one (induction variable substitution [26]).
- Normalize the index expressions to linear expressions of the induction variable (subscript normalization [26]). For induction variable I, the resulting expression has the form $S \times I + C$. S is called the access' *stride*.

If one or more index expressions cannot be normalized, the loop is only a nonpipeline candidate. In particular, indirect array accesses, where array elements are accessed by intermediate results prevent vectorization as in software vectorizing compilers [26].

Normalizing the inner candidate loop in Fig. 2 will create the loop header

$$\texttt{for}\ (\texttt{h} = 0;\ \texttt{h} < \texttt{hlen} - 3;\ \texttt{h}{+}{+})$$

and substitute h by $\texttt{h} + 1$ in the loop body.

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

```
char p1[vlen][hlen], p2[vlen][hlen];
...
/* Vertical and horizontal edge detection       */
/* in both directions                           */
for (v=1; v<vlen-2; v++) /* non-pipeline cand.  */
   for (h=1; h<hlen-2; h++) { /* pipeline cand. */
      int vedge = (p1[v-1][h+1] - p1[v-1][h-1]) +
                  2 * (p1[v][h+1] - p1[v][h-1]) +
                  (p1[v+1][h+1] - p1[v+1][h-1]);
      int hedge = (p1[v+1][h-1] - p1[v-1][h-1]) +
                  2 * (p1[v+1][h] - p1[v-1][h]) +
                  (p1[v+1][h+1] - p1[v-1][h+1]);
      int tmp = abs(vedge) + abs(hedge);
      if (tmp > 255)
         tmp = 255;
      p2[v][h] = (char) tmp;
   }
```

Fig. 2. Edge detector program.

3) Dependence Analysis: The next processing stage analyzes pipeline candidate loops for dependences. There are three general types of dependences [27]. *True* or *flow dependence* occurs when a variable is assigned or defined in one statement and used in a subsequently executed statement. *Antidependence* occurs when a variable is used in one statement and reassigned in a subsequently executed statement. *Output dependence* occurs when a variable is assigned in one statement and reassigned in a subsequently executed statement. General dependences are either *loop independent* or *loop carried*. The former occurs between statements in the same loop iteration and the latter between statements in different iterations. For loop-carried dependences, the *dependence distance* is the number of iterations between the statements that cause the dependence. In a loop nest, we determine for each loop hierarchy the loop-carried dependences since only these affect the loop-level parallelism.

Since 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. Therefore, pipeline vectorization applies to more loops than software vectorization. We utilize standard dependence analysis methods [27] to detect these dependences. Unfortunately, these methods are not completely accurate. In some cases, they cannot determine the absence of dependences, thus failing to detect potential parallelism.

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 [3]. It is possible to synthesize hardware obeying these dependences, but the resulting circuits may contain feedback cycles. In contrast to software vectorization, regular dependences do not prevent pipeline synthesis, although they can 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 usually requires many sequential memory accesses and is only feasible with very fast memories such as on-chip memories.

In the program in Fig. 2, there are no dependences from array accesses: array p1 is only read and array p2 is only written.

```
int x[N];
...
x[0] = 0;
x[1] = 1;
for(i=0; i<N-2; i++)
   x[i+2] = x[i] + x[i+1];
```

Fig. 3. Fibonacci numbers program.

The following example shows that our approach can deal with loops not usually vectorized by software vectorizing compilers. The loop

$$\texttt{for}\,(\texttt{i} = 0;\ \texttt{i} < \texttt{N};\ \texttt{i++})\quad \texttt{x[i]} = \texttt{x[i]} + \texttt{x[i} + 1\texttt{]};$$

has a loop-carried dependence stemming from the assignment to array x but no value written to memory is read in a subsequent loop iteration. Only out-of-order execution of the assignments would lead to a real dependence. Hence, it is an antidependence and we can disregard it and vectorize the loop.

To the contrary, the program computing the Fibonacci numbers in Fig. 3 contains true loop-carried dependences. The assignment to $\texttt{x[i} + 2\texttt{]}$ depends on the two previous assignments. Since both dependences are regular (dependence distances 1 and 2), we can generate a pipeline circuit for this program. But the dependences have to be handled as described in the next paragraph.

4) Removing Vector Dependences: The hardware synthesis technique presented in the next section cannot handle true loop-carried dependences stemming from array (vector) accesses, as shown in Fig. 3. Therefore, pipeline candidate loops have to be transformed to remove them in the following way. First, vector accesses depending on earlier iterations are substituted by new scalar variables in the candidate loop body. Next, at the end of the loop body, instructions are inserted that assign these variables to the values they depend on in the original program. For dependences with dependence distances larger than one, additional variables and assignments are inserted. Finally, assignments to initialize the variables are added before the loop. For the Fibonacci number program, accesses $\texttt{x[i]}$ and $\texttt{x[i} + 1\texttt{]}$ are substituted. Fig. 4 shows the resulting transformed program. It only contains dependences stemming from the new scalar variables new0 and new1. In every iteration, their values from the previous iteration are read. These dependences will be handled by the hardware synthesis phase.

```
int x[N];
...
x[0] = 0;
x[1] = 1;
int new0 = x[0];
int new1 = x[1];
for(i=0; i<N-2; i++) {
   x[i+2] = new0 + new1;
   new0 = new1;
   new1 = x[i+2];
}
```

Fig. 4. Transformed Fibonacci program.

B. Hardware Synthesis

The hardware synthesis phase contains three steps: dataflow graph generation (Section III-B1) and extension (Section III-B2), pipelining (Section III-B3), and controller synthesis (Section III-B4).

For those candidate loops that pass the dependence test, independent pipeline circuits are synthesized. They are later integrated in larger designs or instantiated in separate configurations (see Section III-C).

Various storage allocation schemes can be used. For instance, scalar variables can be held in hardware registers and arrays can be stored on off-chip memory. On some FPGA families, small arrays of data can also be stored in very fast on-chip memory. 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. All element addresses for the linear array accesses can be computed in parallel with the loop computations. Thus, they do not slow down the application. However, for arbitrary accesses, address computations depend on loop computation results and must be scheduled accordingly, thereby slowing down the circuits. Pointer accesses are indirect accesses to the entire host memory space and are only possible in tightly coupled architectures with direct memory access (DMA).

We first generate an acyclic *dataflow graph* for the loop body. Next, regular loop-carried dependences are resolved, possibly introducing feedback cycles. Finally, the circuit is pipelined and a controller is synthesized.

1) Dataflow Graph Generation: We generate an acyclic combinational *dataflow graph* for the loop body by analyzing its internal dependences and allocating a new operator for each operation in the program's expressions. This simple "direct compilation" shares no resources within the loop body but later allows overlapping loop iterations by pipelining. We treat array accesses and scalar variables uniformly. Since the loop body can only contain linear code and conditional statements, we can use a control flow/data flow transformation [19] to generate one combined dataflow graph for the entire loop body. It computes all program branches in parallel and uses multiplexers to select the correct values of conditionally assigned variables. Resources in these mutually exclusive paths can be shared without interfering with pipelining (cf. the PISYN system [18]). For instance, an adder and a subtractor with the same inputs can be replaced by a combined adder/subtractor if their outputs are not required concurrently. Our dataflow graph generation is similar to the method used in the Transmogrifier C compiler [14], but our method avoids unnecessary memory accesses: when an input value remains unchanged in one branch of a conditional statement, we do not read the old value in and write the unchanged value back. Instead, write-enable signals are generated for the RAM accesses to write values only if the appropriate conditions are met.

To further reduce redundant memory accesses, index-shifted accesses to the same array are combined and realized by shift registers [28]. 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

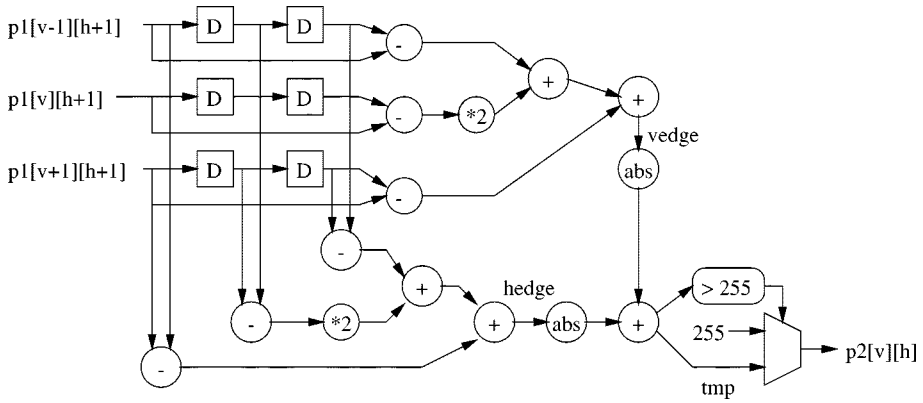

Fig. 5. Edge detector dataflow graph.

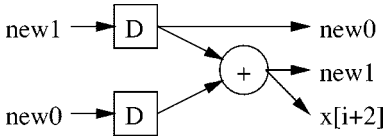

Fig. 6. Incomplete Fibonacci dataflow graph.

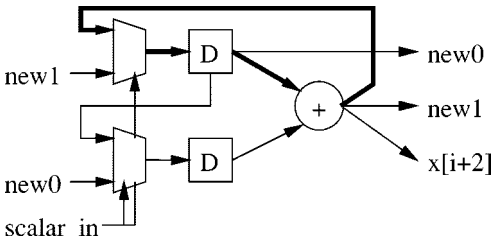

Fig. 7. Complete Fibonacci dataflow graph.

throughput directly depends on the number of vector inputs and outputs.

Fig. 5 shows the dataflow graph for the edge detector program from Fig. 2. There are three vector inputs for `p1[v − 1]`, `p1[v]`, and `p1[v + 1]` and a vector output for `p2[v]`. The shift registers are represented by delay elements `D`. Note that the `if` statement of the loop body is implemented by a multiplexer, which either selects the conditionally assigned value (`255`) or the unchanged value of `tmp` generated in the previous statement.

The dataflow graph generated for the transformed Fibonacci program in Fig. 4 is shown in Fig. 6. Obviously, this circuit does not produce correct outputs since the loop-carried dependences stemming from variables `new0` and `new1` are not accounted for. The registers are only initialized but never updated. The next section shows how the circuit can be altered to produce correct output.

2) Dependences and Feedback Cycles: If a loop has regular loop-carried dependences, the dataflow graph must be extended to use the correct values upon which a computation depends. The transformation in Section III-A4 substituted dependent array accesses by scalar variables. Thus, all loop-carried dependences remaining in a pipeline candidate stem from scalar variables. Such dependences are treated in the following way. Since one loop iteration is executed every pipeline 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 pipeline cycle. To achieve this, a multiplexer is added at the register's input. It selects the input value during initialization and the feedback value during normal operation depending on an external control signal provided by the environment.

Fig. 7 shows the result for the Fibonacci program in Fig. 4. For `new0` and `new1`, multiplexers have been inserted between the inputs and the registers storing the variables. During initialization, the control signal `scalar_in` selects the input values that are used for the first loop iteration. All subsequent iterations select the other inputs of the multiplexers that are connected to the output values of the variables in the previous iteration. In this example, a *feedback cycle* from the output of the adder to the register holding `new1` is created (bold lines in Fig. 7). However, not all dependences result in feedback cycles.

This example shows that the feedback operators synthesized by pipeline vectorization are more general than those available in single-instruction multiple-data (SIMD) parallel programming languages [27]. Such languages feature special REDUCE or SCAN operations but they are limited to single operators with direct output feedback to one input (for instance, ADD-SCAN for an accumulator). The same is true for software vectorizing compilers, which extract these operations. Arbitrarily, customized feedback units, as the one shown in Fig. 7, are only possible when a pipeline is implemented in hardware.

3) Pipelining and Timing: So far, we have generated a dataflow graph that computes one loop iteration once all input registers are set. It 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, effectively overlapping different loop iterations, and 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.

Theoretically, it is possible to pipeline an acyclic dataflow graph very deeply and run it at a very high clock speed. In a practical implementation, however, the system clock cycle T_C is restricted by the combinational delay of the controller (cf. Section III-B4). Since most pipelines are fed by data from external memory, we also require that an external memory access (assuming synchronous RAM) completes in one cycle. Hence, we choose an appropriate value for T_C for each target architecture, cf. Section VI-C.

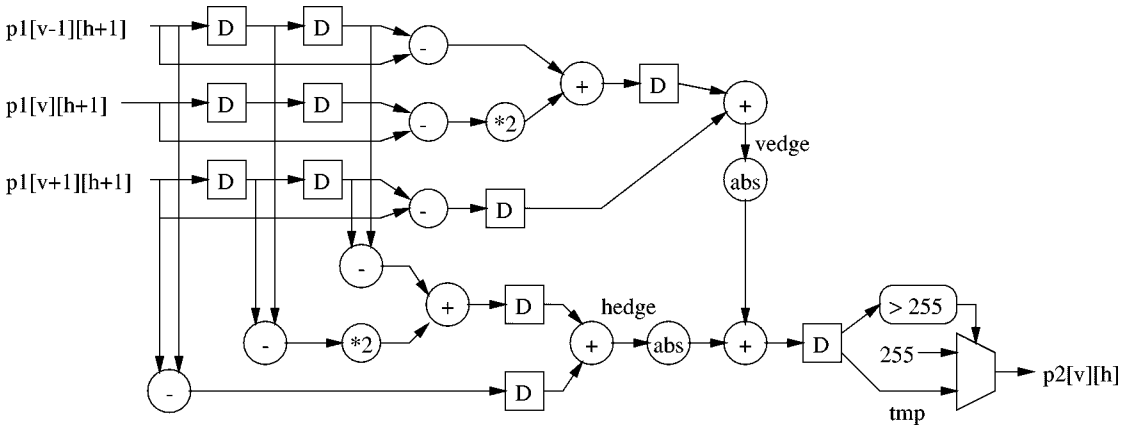

Fig. 8. Pipelined edge detector dataflow graph.

For correct operation, pipelining has to insert the same number of registers on all noncyclic paths from an input register to an output. Register insertion, however, is forbidden in feedback cycles because this would change the circuit's behavior. If the maximum delay within a feedback cycle T_{fb} is larger than T_C, several system clock cycles are required for one pipeline cycle. Then the pipeline cycle time T_P is a multiple of T_C such that $T_P \geq T_{fb}$. As mentioned above, vector accesses restrict the throughput, too. The pipeline cycle must contain at least N_{mem} clock cycles, where N_{mem} is the number of clock cycles needed to perform all vector accesses to external memory required for one loop iteration.

Every access to the same memory bank takes one cycle but accesses to different banks can occur concurrently. Hence, all accesses are sequential for architectures with one memory bank and N_{mem} equals the number of accesses. However, on architectures with several memory banks the allocation of the program arrays to the memory banks determines how many accesses can occur concurrently. This allocation can be performed manually by the user or automatically [28], which aims to minimize N_{mem}.

The resulting pipeline cycle time is $T_P = N_{PC} \times T_C$, where the number of clock cycles per pipeline cycle N_{PC} is the smallest number such that T_P meets the above stated requirements $T_P \geq T_{fb}$ and $T_P \geq N_{\text{mem}} \times T_C$. It is computed as the following:

$$N_{PC} = \max\left(\left\lceil \frac{T_{fb}}{T_C} \right\rceil, N_{\text{mem}}\right).$$

Neglecting the time required for filling and flushing the pipeline, we can now easily predict that a loop with n iterations executes in $n \times N_{PC}$ cycles or in time $n \times N_{PC} \times T_C$.

We use a standard retiming technique [29] to insert the minimal number of flip-flops necessary to achieve T_P. For reconfigurable devices, the technique is extended to take into account that in many FPGAs, combinational gate outputs can be latched in the same cell. Pipelining requires estimates of all operators' delays. They are provided by a technology-specific component library parametrized by operator bit width. The same 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.

Pipelining reduces the critical path of the edge detector in Fig. 5 by inserting additional registers. Fig. 8 shows the circuit after two pipeline stages have been inserted. The Fibonacci dataflow graph in Fig. 7 cannot be improved since it contains only one operator. Nevertheless, wherever possible, the hardware computation and the writing of output values are pipelined in candidate loops.

4) Controller Synthesis: Finally, control circuitry for the pipeline is generated. For FPGA implementations where there are abundant latches, we generate a one-hot controller triggered by an external *START* signal. As mentioned above, let T_C be the cycle time of the controller. First, the controller initializes the pipeline loop's index variable and then repeatedly loops through N_{PC} cycles to complete a pipeline cycle. At the beginning of a new pipeline cycle, the loop index is incremented.

All memory accesses of a loop iteration are scheduled in one of the N_{PC} cycles with memory writes (for results produced in a previous iteration) before memory reads (for input values of the next iteration). By storing them in registers, all input and output values are presented to the pipeline synchronously at the beginning of a pipeline cycle. The registers within the pipeline are only clock enabled at the beginning of a pipeline cycle. Thus, the pipeline is effectively clocked with the period T_P and it contains multicycle operators.

A validity bit is used to control the filling and flushing of the input shift registers and the pipeline stages. It also guarantees that only valid output values are written to external memory. When all computations are completed, a *STOP* signal is raised. It can be used by external circuitry or to notify the calling host program in codesign mode about completion of coprocessor tasks. Along with the controller states, we generate operators to compute memory addresses for the external memory banks and multiplex them to the memories' address buses.

C. Partitioning and Integration

The last pipeline vectorization phase performs hardware–software partitioning as well as hardware integration.

1) Hardware–Software Partitioning: Partitioning determines which parts of a program will be executed in software and which in hardware and it does not exist in hardware mode. The partitioning depends on several properties of the program. Obviously, only hardware candidates can be allocated to

hardware. The combined area estimations of the corresponding circuits of all chosen candidates must not exceed the given hardware area. For reconfigurable systems, separate configurations can be generated for each loop and the FPGA resources are reused by reconfiguration.

The main partitioning criterion, however, is the expected speedup achieved by the coprocessor. This estimation problem and an automatic partitioning procedure have been addressed elsewhere [3], [4] and are beyond the scope 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. Our methodology supports producing explanations and helps the user in partitioning a program manually.

Partitioning also determines if hardware is synthesized for nonpipeline candidates. Though they are not likely to significantly speed up computations, they can, for instance, initialize variables directly in hardware and, therefore, reduce the need for slow data transfers from the host. Generating hardware for the outer loop in Fig. 2 sets variable `v` for each execution of the inner loop and thus allows us to compute the memory addresses for the array accesses completely in hardware. Hence, the entire program can be executed in hardware without host interaction. The hardware integration phase discussed in the next section synthesizes circuitry for the selected nonpipeline candidates.

On the software side, the program running on the host is generated by substituting the chosen loops by runtime library calls for executing the pipeline as well as copying data between host and coprocessor. For reconfigurable hardware implementation, appropriate library functions reconfigure the FPGAs if a new coprocessor is needed.

2) Hardware Integration: Hardware integration first synthesizes dataflow graphs for the selected nonpipeline candidates and integrates them with the selected pipeline circuits. We use a simple syntax directed synthesis technique [15]. It generates a controller executing instructions in their original order. Only assignments within a basic block (a linear piece of program code) are performed concurrently if there are no local dependencies between them. If the resulting delay of an assignment becomes larger than T_C, the clock-cycle time of the pipelined circuit, the assignment is performed in several cycles. Thus, the performance-critical pipelined part of the design is never slowed down. Since nonpipelined circuitry is only intended for outer control loops and initialization, we do not attempt to share operators amongst instructions as in high-level synthesis systems [19]. This avoids time-consuming optimizations and complicated control. The new controllers are easily combined with the pipeline controllers by combining their *START* and *STOP* signals.

Next, the dataflow graphs are transformed into a device-specific netlist by instantiating all operators with macros from the component library also used for estimation. These netlists have to be combined with interface circuitry for host and local memory access and clock signals on the system used. Differential netlist generation applies only for partially reconfigurable systems (see Section V-C). The netlists are then further processed with off-the-shelf vendor tools. The fixed clock cycle T_C can normally be achieved. Though it is difficult to estimate routing delays, it suffices to include reasonable slack for routing in the delay estimates used in pipelining and syntax directed synthesis. The controllers do not contain deep combinational logic since we generate one-hot controllers.

In case the implemented circuit does not meet the estimated area or delay targets, a step indicated by the dotted line in Fig. 1 back annotates the dataflow graphs with more accurate values and the subsequent steps are repeated. For instance, more pipeline stages can be inserted to reduce the delay. Alternatively, an experienced user can review and optimize the generated circuits manually.

IV. Loop Transformations

The core design flow discussed so far is limited to programs with innermost loops of suitable size: problems occur when the loop body is too small to warrant the hardware overheads or too large to fit in the given hardware. The method is also sensitive to programming styles. For instance, the edge detector program in Fig. 2 could not be vectorized if small inner loops had been used to compute `vedge` and `hedge`. The `h` loop would no longer be an innermost loop and, therefore, would not be considered a pipeline candidate. This section shows how transformation techniques known from parallelizing software compilers such as loop unrolling, loop tiling, loop merging, loop distribution, and loop interchange 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 in a codesign system. We apply all transformations when applicable, giving priority to unrolling and tiling since they have the biggest influence on the resulting performance.

The hardware synthesis part of the core design flow is extended, as shown in Fig. 9. The transformations generate new variations of the candidates and add them to the internal program representation. Then, the hardware synthesis is repeated for the new loops (see path I in Fig. 9). Finally, the best suited among the original and alternative processors are implemented (path III, Fig. 9).[2] Since the transformations only manipulate the internal program and high-level dataflow graph representations, all interesting alternatives can be generated quickly. Only the implementation of the selected processors involves running slow hardware design tools, such as place and route tools.

A. 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 processors and, therefore, more potential parallelism. However, the size of the processors must match the available hardware resources.

A candidate loop can be completely unrolled if its bounds are constant. This situation occurs in many programs; for instance,

[2]Path II in Fig. 9 refers to the circuit specializations discussed in Section V.

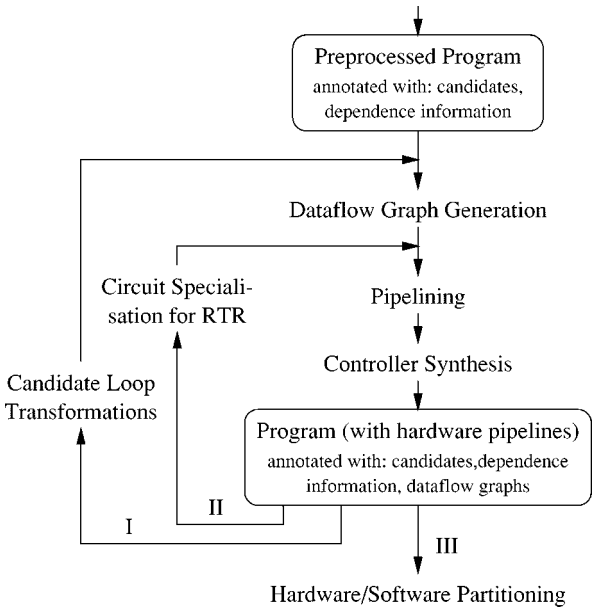

Fig. 9. Extended hardware synthesis design flow.

in image processing applications with loops over small constant-size templates [30]. Specific examples include the skeletonization program used in Section VII-B or filters with a constant number of taps.

If an innermost loop is completely unrolled, the next outer loop can be vectorized. However, this does not always lead to a feasible pipeline coprocessor for the outer loop since the transformed loop body might be larger than the available hardware resources or the coprocessor might be too slow due to too many vector inputs and outputs. Therefore, partitioning will decide if the original or the unrolled candidate is selected.

Partial unrolling is not useful for pipeline vectorization since it increases the number of vector inputs and outputs and the pipeline cycle time and the hardware size. It is only useful when combined with vectorizing the next outer loop. This is achieved by loop tiling, described next.

B. Loop Tiling

Loop tiling is a 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 (Section IV-D). We adapt this technique for pipeline vectorization.

Transformation steps 1 and 2 in Fig. 10 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

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

$\Rightarrow$ (1)

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

$\Rightarrow$ (2)

```
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      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); }
```

$\Rightarrow$ (3)

```
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      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); } }
```

$\Rightarrow$ (4)

```
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)
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); } }
```

$\Rightarrow$ (5)

```
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)
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); } }
```

Fig. 10. Hardware-specific loop tilting.

estimated by $tsize = (area_{HW} - area_{PRE} - area_{POST})/area_F$, 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 that 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` (also known as strip mining) 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`) are first "sunk" in the tile loop (by adding guards) since interchange is only possible for perfectly nested loops—loop nests without 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 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* [31]. This is the case if all dependences carried by these loops have nonnegative 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 the generated pipelines, no backward dataflow between "processing elements" is allowed but nonlocal forward flow is.

The output of step 2 cannot directly be vectorized. Thus, we devise additional hardware-specific transformations extending software loop tiling. The nonconstant 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. We rewrite this guard to

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

where $\%$ denotes the modulo operator. It is now explicit that this formula can only evaluate to false for the last tile $jt = (n-1)/tsize$. Step 3 shows this transformation.

Now the inner loop can be unrolled in step 4. Unfortunately, the guard has to be replicated, too, although it can be omitted for the case $j = 0$. This condition is always true because every tile performs at least one inner loop iteration.

Implementing the guards in hardware adds a comparator to each processing element. However, since the guards do not depend on the index variable `i`, flags for the guards can be assigned outside the vectorized loop. In codesign mode, the outer loops can generate the flags in software and pass them to the hardware, thus avoiding the hardware comparators. Step 5 in Fig. 10 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 dataflow graph adjusted to the given hardware resources. An example will be given in Section VII-A.

C. Loop Merging

Loop merging is another means of increasing parallelism in loop bodies. Its scope is limited to loops (or loop nests) traversing the same index space and all dependences of the original loops must be preserved in the merged loop. If direct merging violates dependences, shifted loop merging may be possible. It combines loop alignment with loop merging [27].

```
1 for (v=1; v < vlen-2; v++)
2   for (h=1; h < hlen-2; h++)
3     p2[v][h] = F(p1[v-1],p1[v],p1[v+1]);
4 for (v=1; v < vlen-2; v++)
5   for (h=1; h < hlen-2; h++)
6     p3[v][h] = G(p2[v-1],p2[v],p2[v+1]);

                    ⇒

1 for (v=1; v < vlen-2+d; v++)
2   for (h=1; h < hlen-2; h++) {
3     if (v < vlen-2)
4       p2[v][h] =
5         F(p1[v-1], p1[v], p1[v+1]);
6     if (v >= 1+d)
7       p3[v-d][h] =
8         G(p2[v-1-d],p2[v-d],p2[v+1-d]); }
```

Fig. 11. Shifted loop merging.

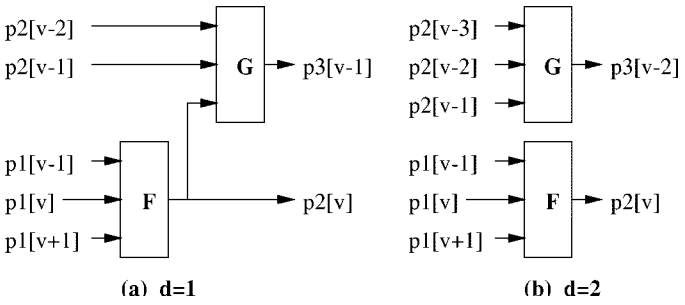

Fig. 12. Dataflow graphs for merged loop.

An example is given in Fig. 11, where F and G represent linear image processing operators depending on a 3×3 neighborhood. Since direct merging would violate dependences from line 3 to line 6 in the original program, the loops have to be aligned: iteration i of the first loop is fused with iteration $i - d$ of the second loop. The alignment parameter d is chosen as the minimal integer that preserves all dependences. The shifted merged loop contains an extra d iterations and guards are added to skip execution during either the first d or the last d iterations. For this example, $d = 1$ (for the outer loop) is sufficient.

Fig. 12(a) shows the corresponding dataflow graph for $d = 1$. The operator G is delayed by one iteration of the outer loop. We see that the merged pipeline requires five vector inputs and two outputs. This might slow down the pipeline considerably and not make merging 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 VII-B.

By choosing a larger alignment parameter d, this transformation becomes suitable for multichip systems. Fig. 12(b) shows the resulting dataflow graph 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 that 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 speed-up factor up to n can be achieved compared to sequential execution. The minimal or overdelayed d value is chosen depending on the given target architecture.

D. Other Loop Transformations

The following paragraphs discuss loop transformations, which are of limited importance for pipeline vectorization. We do not attempt to transform entire loop nests as in [31] 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.

Loop Distribution: 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. We do not consider loop distribution any further since it is only necessary for very limited FPGA resources.

Loop Interchange: Loop interchange swaps perfectly nested loops. As discussed for loop tiling, it is legal if the interchanged loops are fully permutable. This transformation 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.

Strip Mining: 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.

E. Partitioning Extensions

Automatic hardware–software partitioning is extended by a recursive algorithm that 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 Fig. 1 is activated.

V. Runtime 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 that are not constant, thereby reducing a design's delay and area [32]. Whenever a value changes, the circuit is reconfigured. Rather than changing the input of flexible operators, a design that exploits 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. Therefore, we only consider those variables for value propagation that do not change inside the loops to be vectorized. The hardware–software partitioning must evaluate the tradeoff 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.

A. Limited Value Propagation

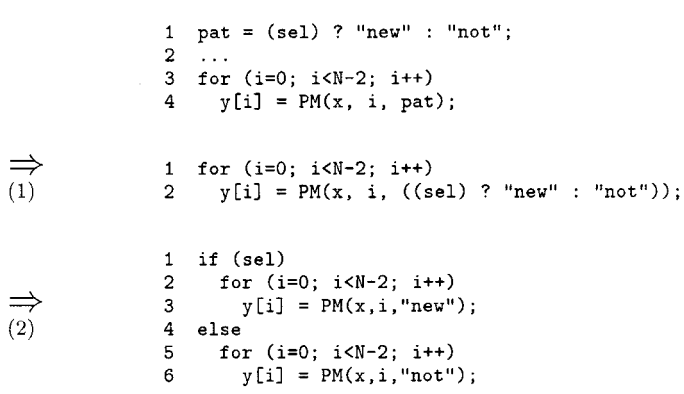

Fig. 13. Limited value propagation.

A. 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 Fig. 13. The program is a string pattern matcher where $PM(\texttt{x}, \texttt{i}, \texttt{pat})$ 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 (Fig. 13, step 1). 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 Fig. 9 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.

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. Fig. 10). 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 tradeoffs 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. Therefore, chip-level RTR will not be useful for examples like the pattern matcher in Fig. 13, 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 [33] to determine the fastest partial configuration to switch between two similar designs. Hence, small changes can be performed very quickly.

B. Arbitrary Value Propagation

The second case of RTR occurs if a variable can assume any value at runtime. 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 runtime, this case cannot be handled with FPGAs that can only be configured completely. It is only suitable for partially reconfigurable FPGAs that allow adaptation of an operator to any constant input values within a few cycles at runtime. Therefore, a circuit "skeleton" is synthesized, which reserves area for the largest possible constant input operator. At runtime, all these operators are adapted to the given values. Doing this also requires a special component library that 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 controller synthesis—but not dataflow graph generation—is repeated for these new implementations (see path II in Fig. 9). As for limited value propagation, the tile size for partially unrolled loops is increased. Thus, tiling should be repeated. Eventually, the best suited processors are implemented (see path III in Fig. 9) as outlined in Section IV.

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 VII-A.

C. Runtime Reconfiguration 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 tradeoff 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 speed-up estimation, cf. [3], [4]. 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 that just change the differences between two consecutive configurations. Therefore, even the configuration times of unrelated coprocessors are reduced, especially if they share the same control circuitry.

VI. Compiler Implementation

A. SUIF Pipeline Compiler

Our pipeline vectorization prototype implementation is based on the SUIF compiler framework [34], which provides C and Fortran front ends and powerful loop analysis and transformation libraries. We have implemented a prototype SUIF pipeline compiler (SPC), which targets FPGA-based reconfigurable systems. The supported input language is C.

The compiler's analysis phase produces explanations whether a loop is a hardware candidate and if it can be vectorized. This helps the user to change the program accordingly, for instance, by eliminating dependences, so that faster hardware can be generated for more loops. 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 decide which parts of an application benefit from FPGA hardware.

The user selects hardware candidates and loop transformations either interactively or using program annotations. Then an operator-level netlist is synthesized for the selected candidates and output to a file.

B. Target Systems

SPC is not developed for specific board architectures or FPGA families. Only the number and size of the target architecture's memory banks and FPGA-specific component libraries have to be provided so that SPC can generate an architecture and device-specific netlist. An FPGA family-specific extension generates a constraint file for place and route tools.

We currently use a PC-based RC1000-PP board [7] with a Xilinx XC4085XL FPGA and two 2-MB memory banks. The board allows for fast DMA transfers between host and local on-board RAM at 100 MB/s. FPGA configuration takes 780 ms on this board but we expect much faster configuration for the new Xilinx Virtex FPGA used in the next version of this board.

C. Tool Integration

Both SPC and the low-level vendor tools are controlled by a compilation script. After completion of SPC, the generated netlist is combined with hardware descriptions of RC 1000-PPs host and RAM interfaces to form a complete FPGA design. The RAM interface contains logic to generate control signals for the board's asynchronous RAM so that it can be accessed like synchronous RAM by the generated circuit. Next, the script calls the vendor tools and generates a bitstream. The place and route tool uses the constraint file generated by SPC, which also specifies the maximum delays permitted for multicycle operators in the pipelines. Thus, all generated designs run at the same clock speed. For the current chip generation, we use 25 MHz or $T_C = 40$ ns.

The results given in Section VII have been produced with the assistance of SPC. For all applications, the SPC runtime is just a few seconds. The entire compilation time is by far dominated by the FPGA vendor's place and route tool.

```
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; }
```

Fig. 14. String pattern matcher program.

TABLE I
ANALYSIS OF STRING PATTERN MATCHER

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

D. Limitations

SPC is currently limited to the core design flow in hardware mode and the two most important loop transformations: unrolling and tiling. Due to the limitations of current FPGA technology, no floating-point operations are allowed. Since irregular dependences require complicated control within a pipeline cycle, SPC does not support these yet. Finally, since most reconfigurable systems do not have direct access to host memory, we do not handle pointers. We are constantly extending the SPC prototype to remove these restrictions and to implement the entire pipeline vectorization framework.

VII. RESULTS

This section presents pipeline vectorization results. First, two detailed case studies are presented in Sections VII-A and VII-B. Finally, Section VII-C summarizes performance results of these and other benchmark programs.

A. String Pattern Matcher

This case study, a string pattern matcher, evaluates the benefits of loop tiling and runtime circuit specialization. Therefore, it is implemented on a PC-based Xilinx 6200 DS board [8] using a partially reconfigurable XC6216 FPGA. The program, shown in Fig. 14, is the same as that in Fig. 13 but with arbitrary pattern lengths and values.[3] Therefore, the inner loop cannot be unrolled. However, the outer loop (index `i`) can be vectorized after the tiling transformation has been applied. The resulting pipeline circuit is a linear data path of comparators and registers. Both compile-time reconfigurable (CTR) and RTR versions are possible. The CTR version contains generic comparators and the XC 6200s protected registers so that pattern bytes can be loaded directly from the host, whereas the specialized RTR version contains constant comparators. The XC 6216 is large enough to implement the controller and 54 CTR processing elements or 90 smaller specialized RTR processing elements.

Table I shows the raw performance of the implementations running at 25 MHz in 10^6 comparisons per second and 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.

However, the hardware performance data do not include the overheads for initializing 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. Fig. 15 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 RTR further *improves* the performance by approximately 50% for large patterns.

B. Morphological Skeletonization

In this section, we apply our method to a morphological skeletonization algorithm [30]. It is implemented on the RC 1000-PP board mentioned in Section VI-B. This example evaluates loop unrolling and shifted loop merging. Fig. 16 shows the algorithm's 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.

The erosion operator consists of two nested inner loops that 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 II gives pipeline frequencies F in megahertz (the reciprocal of pipeline cycle lengths), raw performance P in 10^6 operations per second, and execution times T in milliseconds for a 512×512 pixel image as well as the total time for the independent execution of all skeletonization operators.

The performance can be improved by merging all operators to produce one large pipeline. The last line in Table II 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 measured 65 ms for the completion of one skeletonization iteration for a 512×512 pixel image on the RC 1000-PP. Even including data transfer (5 ms amortized over 15–30 iterations), the hardware coprocessor was measured to be 16 times faster than software (1045 ms on the 300 MHz PC).

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

C. Benchmark Results

Table III summarizes performance results of several benchmark programs. All but `patmatCTR` and `patmatRTR` are implemented on the RC 1000-PP board. The columns show runtimes

[3]Note that much faster algorithms for string pattern matching exist. However, they cannot easily be implemented in hardware since they are less regular than the simple algorithm used here.

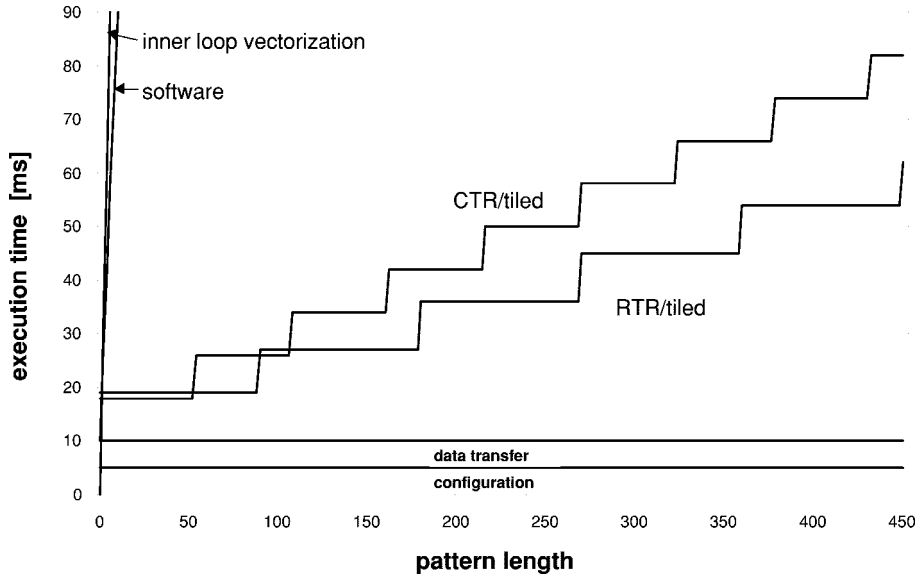

Fig. 15. Execution times for string pattern matcher for $N = 1\,000\,000$.

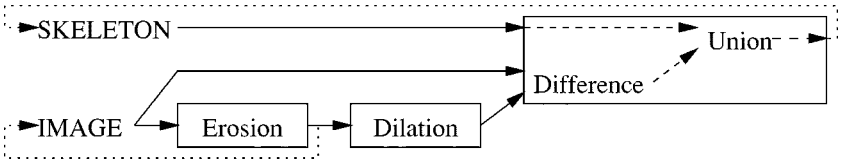

Fig. 16. Morphological skeletonization.

TABLE II
ANALYSIS OF SKELETONIZATION OPERATORS

	1 memory bank			2 memory banks		
	F	P	T	F	P	T
Erosion	4.2	84.0	63	5.0	100.0	52
Dilation	4.2	84.0	63	5.0	100.0	52
Difference/Union	6.2	12.5	42	12.5	25.0	21
Total			168			125
Merged	2.1	88.2	126	3.3	176.4	63

TABLE III
BENCHMARK RESULTS

Program	SW	HW	SpUp	Utilization	PF
`patmatCTR`	1613	74	21.8	100 %	54
`patmatRTR`	1613	54	29.9	100 %	90
`skel`	1045	65	16.1	39 %	44
`sharpen5x5`	153	53	2.9	20 %	29
`convolv3x3`	156	32	4.9	57 %	20
`smooth3x3`	48	32	1.5	13 %	9
`edge3x3`	122	32	3.8	14 %	17
`matmult32`	213	4	53.3	40 %	32

in milliseconds for software (SW) on a 300 MHz Pentium II PC and hardware (HW), hardware speedup (SpUp), device utilization (percent of logic cells used), and parallelization factor (PF). The latter is an indication of the effect of vectorization: it shows how many arithmetic or logic operations can, in principle, be performed in parallel in a pipeline. These operations would have to be performed sequentially without parallelization. The high values of PF indicate that the observed hardware acceleration is mainly achieved through vectorization. Note that the speedup cannot directly be derived from PF since the number of microprocessor clock cycles required per operation depends on the type of the operation and runtime effects such as cache misses. Also, the hardware cannot exploit the available parallelism completely if the pipeline is I/O bound. Note that the device utilization reported is very inaccurate since cells used only for routing are counted, too. Additionally, the current SPC prototype is not optimized for area. For instance, the pipelining algorithm used does not minimize the registers inserted.

The `patmatCTR` and `patmatRTR` programs are the tiled pattern matcher implementations on the 6200 DS board discussed in Section VII-A. The given runtimes result from a pattern length of 400 and an input length of 100 000. The speedups are lower than those reported in Table I since not all processing elements are used in all tiles. Note that the parallelization factor depends on the size of the used FPGA in these tiled implementations. In all other programs, it is limited by the programs themselves.

The `skel` program is the image skeletonization presented in the previous section and `sharpen5` $\times$ 5 is an image sharpen operator which also uses a 5 $\times$ 5 template. `convolv3` $\times$ 3 is a general convolver with a 3 $\times$ 3 template of loadable coefficients, whereas `smooth3` $\times$ 3 is a similar program with constant coefficients—an image smoothing operator. Because general multipliers are substituted by constant multipliers or eliminated at all, the area requirement is hugely decreased. However, because both designs have the same memory access patterns and can both be pipelined, similar runtimes are measured. The same is true for program `edge3` $\times$ 3, the edge detector program in Fig. 2. `edge3` $\times$ 3 shows that several operators (vertical and

horizontal edge detectors) combined in a bigger pipeline can execute as fast as a simple operator (`smooth3` $\times$ 3). The runtimes for all these programs refer to 512 $\times$ 512 pixel greyscale images.

Finally, `matmult32` is a binary matrix multiplication program. It multiplies input vectors with a loadable 32 $\times$ 32 binary matrix. We measured results for 100 000 input vectors.

The results show that pipeline vectorization results in FPGA circuits, which speed up programs from 50% (factor 1.5) up to 53 *times* over a fast microprocessor. The parallelization factor, an indicator independent of the implementation technology, shows that the vectorized implementation of all circuits is one to two orders of magnitude faster than a nonvectorized circuit generated from the same high-level program.

VIII. Conclusion and Future Work

This paper presents a framework for producing optimized pipelined circuits from high-level programs. It combines the vectorization of inner loops to extract parallelism in a sequential program with circuit pipelining to exploit this parallelism in hardware. The framework includes new optimizing transformations that customize hardware processors to meet specific resource constraints and exploit RTR. The case studies show that some transformations result in hardware acceleration that cannot be achieved easily by hand. Others improve the performance of processors significantly. All benchmarks show that pipeline vectorization generally synthesizes much more efficient circuits than simpler sequential high-level design techniques. The time efficiency is often comparable with manually designed VHDL designs, although the circuits might not be as area efficient. To produce more competitive designs, future compilers will include advanced design techniques like the use of on-chip RAM as delay lines [28]. Our framework can select, generate, and integrate coprocessors automatically while retaining the flexibility to allow users to influence the synthesis process.

Our approach appears to be simpler to use than other C-based hardware design systems. Software source code can often be left unchanged or only minor changes are necessary to enable vectorization and pipeline synthesis. In contrast, parallel design languages, even if C-based, require the user to identify available parallelism and synchronize parallel program components. For instance, to design an efficient convolver, parallel statements are required to pipeline the computations. The designer needs to have a clear idea of the resulting circuit and the resulting program often has little resemblance to the original software source code. Also, memory allocation and hardware–software interface generation are error-prone manual tasks.

In its current state, pipeline vectorization is not as universal as high-level synthesis systems: no design-space exploration, general scheduling techniques, or resource sharing are employed. Instead of using heuristic optimizations to tackle these problems, we use higher level dependence analysis information in software source code. Hence, we can use what often is the most effective schedule available for regular iterative computations: pipelining.

Our current compiler prototype targets FPGAs. We do not optimize space by sharing operators for the sake of both compilation and execution speed. It is more important to have very fast processors for the program "hot spots" rather than a slow universal design with many idle operators. Space is not our main concern since reconfigurable systems have other options for computations not fitting on a given hardware design: software or reconfiguration. Moreover, hardware sharing may increase the amount of routing to the shared resource, increasing both delay and size of the resulting circuit.

Our research shows that the systems available today are generally useful for reconfigurable computing applications. For most loosely coupled reconfigurable architectures, however, the slow communication over the system bus is still a major obstacle to achieving high speedups. Advanced tightly coupled systems [10], [23] could improve this situation. Another problem is the long runtime of the FPGA vendor tools that are not comparable to modern software compilers. These tools need to improve in order to make reconfigurable computing more attractive. For instance, the place and route tools could offer a prototyping mode for quick results and a slower optimizing mode just as software compilers do. Coarse-grain FPGAs specifically designed for reconfigurable applications might be another solution to this problem since simple fast mapping tools can be developed for them [11].

Future work will include combining the fine-grain vectorization presented in this paper with coarse-grain task-level parallelism. With this approach, the abovementioned communication latency could be hidden by overlapping communication and computation. Strategies to transform entire loop nests will also be studied and automatic partitioning will be included in our compiler prototype. 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.

References

[1] C. E. Kozyrakis and D. A. Patterson, "A new direction for computer architecture research," *IEEE Computer*, vol. 31, pp. 24–32, Nov. 1998.
[2] M. Weinhardt and W. Luk, "Pipeline vectorization for reconfigurable systems," in *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*. Napa, CA, Apr. 1999, pp. 52–62.
[3] M. Weinhardt, "Compilation and pipeline synthesis for reconfigurable architectures," in *Reconfigurable Architectures Workshop*, Geneva, Switzerland, Apr. 1997, pp. 105–112.
[4] ——, "Übersetzungsmethoden für strukturprogrammierbare rechner," Ph.D. dissertation (in German), Univ. Karlsruhe, Karlsruhe, Germany, July 1997.
[5] SystemC. The open systemC initiative. [Online]. Available: http://www.systemc.org
[6] J. E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. H. Touati, and P. Boucard, "Programmable active memories: Reconfigurable systems come of age," *IEEE Trans. VLSI Syst.*, vol. 4, pp. 56–69, Jan. 1996.
[7] Embedded Solutions Ltd. RC1000-PP product information sheet. [Online]. Available: http://www.embedded-solutions.ltd.uk/ProdApp/RC1000PP.htm
[8] S. Nisbet and S. A. Guccione, "The XC6200DS development system," in *Field Programmable Logic and Applications*, New York: Springer-Verlag, 1997, pp. 61–68.
[9] P. M. Athanas and H. F. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *IEEE Computer*, vol. 26, pp. 11–18, Mar. 1993.

[10] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *Proc. FPGAs Custom Computing Machines*. Napa, CA, Apr. 1997, pp. 12–21.
[11] T. J. Callahan and J. Wawrzynek, "Instruction level parallelism for reconfigurable computing," in *Field Programmable Logic and Applications*, New York: Springer-Verlag, 1998, pp. 248–257.
[12] S. A. Guccione and M. J. Gonzalez, "A data-parallel programming model for reconfigurable architectures," in *Proc. FPGAs Custom Computing Machines*. Napa, CA, Apr. 1993, pp. 79–87.
[13] S. Guccione, "Programming fine-grained reconfigurable architectures," Ph.D. dissertation, Univ. Texas, Austin, TX, 1995.
[14] D. Galloway, "The transmogrifier C hardware description language and compiler for FPGAs," in *Proc. FPGAs Custom Computing Machines*. Napa, CA, Apr. 1995, pp. 136–144.
[15] I. Page and W. Luk, "Compiling Occam into FPGAs," in *FPGAs*. Abingdon, U.K.: Abingdon, 1991, pp. 271–283.
[16] *Handel-C Reference Manual*, Embedded Solutions Limited, Abingdon, U.K., 1998.
[17] R. Camposano, "From behavior to structure: High-level synthesis," *IEEE Des. Test Comput.*, vol. 7, pp. 8–19, Oct. 1990.
[18] R. Camposano and W. Wolf, *High-Level VLSI Synthesis*. Norwell, MA: Kluwer, 1991, ch. 3.
[19] D. D. Gajski, N. D. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Norwell, MA: Kluwer , 1992.
[20] "Behavioral Compiler User Guide," Synopsys, Mountain View, CA, v1998.08, 1998.
[21] D. Soderman and Y. Panchul, "Implementing C algorithms in reconfigurable hardware using C2Verilog," in *Proc. FPGAs Custom Computing Machines*. Napa, CA, Apr. 1998, pp. 339–342.
[22] M. B. Gokhale and J. M. Stone, "NAPA C: compiling for a hybrid RISC/FPGA architecture," in *Proc. FPGAs Custom Computing Machines*. Napa, CA, Apr. 1998, pp. 126–135.
[23] C. R. Rupp, M. Landguth, T. Garverick, E. Gomersall, and H. Holt, "The NAPA adaptive processing architecture," in *Proc. FPGAs Custom Computing Machines*. Napa, CA, Apr. 1998, pp. 28–37.
[24] E. Fabiani, D. Lavenier, and L. Perraudeau, "Loop parallelization on a reconfigurable coprocessor," in *Proc. Workshop Design, Test, Applications*, Dubrovnik, Croatia, June 1998.
[25] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers—Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.
[26] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*. Reading, MA: Addison-Wesley, 1991.
[27] M. Wolfe, *High Performance Compilers for Parallel Computing*. Reading, MA: Addison-Wesley, 1996.
[28] M. Weinhardt and W. Luk, "Memory access optimization and RAM inference for pipeline vectorization," in *Field Programmable Logic and Applications*, New York: Springer-Verlag, 1999, pp. 61–70.
[29] C. E. Leiserson and J. B. Saxe, "Optimizing synchronous systems," *J. VLSI Comput. Syst.*, vol. 1, pp. 41–67, 1983.
[30] H. R. Myler and A. R. Weeks, *Computer Imaging Recipes in C*. Englewood Cliffs, NJ: Prentice-Hall, 1993.
[31] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, pp. 452–471, Oct. 1991.
[32] M. J. Wirthlin and B. L. Hutchings, "Improving functional density through runtime constant propagation," in *ACMIGDA Int. Symp. Field-Programmable Gate Arrays*. Monterey, CA, Feb. 1997, pp. 86–92.
[33] W. Luk, N. Shirazi, and P. Y. K. Cheung, "Compilation tools for run-time reconfigurable designs," in *Proc. FPGAs Custom Computing Machines*. Napa, CA, Apr. 1997, pp. 56–65.
[34] The Stanford SUIF Compiler Group, Stanford University. [Online]. Available: http:// suif.stanford.edu

Markus Weinhardt received the Dipl. and Dr.Ing. degrees in informatics from the University of Karlsruhe, Karlsruhe, Germany, in 1992 and 1997, respectively.

He is currently a Marie Curie Postdoctoral Fellow in the Department of Computing, Imperial College, University of London, U.K. His research interests include reconfigurable and parallel computing, focusing on high-level compilation techniques.

Wayne Luk (S'85–M'89) received the M.A., M.Sc., and D.Phil. degrees in engineering and computing science from the University of Oxford, Oxford, U.K.

He is a Member of the Academic Staff in the Department of Computing, Imperial College, University of London, U.K. His research interests include theory and practice of customizing hardware and software for specific application domains such as graphics and image processing, multimedia, and communications. His current work involves high-level compilation techniques and tools for parallel computers and embedded systems, particularly those containing reconfigurable devices such as field-programmable gate arrays.