

Dynamic Loop Pipelining in Data-Driven Architectures

João M. P. Cardoso^{1,2}

¹ Faculty of Sciences and Technology, University of Algarve
Campus de Gambelas, 8000 – 117 Faro, Portugal

² INESC-ID, Lisbon, Portugal
jmpc@acm.org

ABSTRACT

Data-driven array architectures seem to be important alternatives for coarse-grained reconfigurable computing platforms. Their use has provided performance improvements over microprocessors and shorter programming cycles than FPGA-based platforms. As with other architectures, in data-driven architectures loop pipelining plays an important role to improve performance. Usually this kind of pipelining can be achieved using the dataflow software pipelining technique or other software pipelining approaches. Although performance improvements are achieved, those techniques heavily depend on the insertion of pipelining stages and thus require complex balancing efforts. Furthermore, those techniques statically define the pipelining and do not take fully advantage of the dynamic scheduling attainable by the data-driven concept. This paper presents a novel scheme to pipeline loops in data-driven architectures, orchestrated by a handshaking protocol. Using the new approach, self loop pipelining is naturally achieved. The scheme is based on duplicating cyclic hardware structures, in order they are autonomously executed, with synchronization being achieved by the data flow. It can be applied to nested loops, requires less aggressive pipeline balancing efforts than usual software pipelining techniques, and innermost loops with conditional structures can be pipelined without conservative pipelining implementations.

We show results of using the proposed technique when mapping algorithms in imperative programming languages to the PACT eXtreme Processing Platform (XPP). The results confirm improvements over the use of conventional loop pipelining techniques. Better performance and fewer resources are achieved in a number of cases.

Categories and Subject Descriptors

D.1.2 [Programming Techniques]: Automatic Programming—Program Synthesis; D.3.4 [Processors]: Optimization; C.1.3

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CF'05, May 4–6, 2005, Ischia, Italy.

Copyright 2005 ACM 1-59593-019-1/05/0005...\$5.00.

[Processors Architecture] — Other Architecture Styles — Data-flow architectures;

General Terms

Algorithms, Performance, Design.

Keywords

Software Pipelining, Data-Driven Architectures, Dataflow, Reconfigurable Computing, Compilation.

1. INTRODUCTION

Coarse-grained reconfigurable arrays are becoming suitable choices for extending traditional computing engines (*e.g.*, Von Neumann-style microprocessors) [16]. They efficiently realize spatial computing [11][4], which might be important to meet the energy consumption and computing demands of the future computing systems (*e.g.*, embedded systems).

A number of coarse-grained architectures with dataflow semantics (*e.g.*, KressArray [17], XPP [2], and WaveScalar [28]) have been focus of recent academic and commercial efforts, with encouraging results. They resemble many of the concepts of processor arrays, introduced in the 80's [31][12], namely *wavefront* [23] and data-driven arrays [22]. Those architectures devised a scalable and effective fashion to directly support the dataflow computational model and spatial computing. In data-driven architectures, the availability of operands triggers the execution of the operation to be performed on them [31]. Therefore, data-streams can be processed through the processing elements (PEs) of the array without requiring centralized memory elements such as RAMs. Their suitability for reconfigurable computing platforms also comes from the fact that data-driven arrays naturally support computing in space.

Also for the future ASIC scenario, some researchers advocate the use of hardware structures behaving in a static dataflow fashion [5][4]. One of the reasons is the avoidance of centralized control units, which is an envisaged goal since the evidence that interconnection delays are becoming predominant. Even asynchronous dataflow fine-grained arrays, based on FPGAs (Field Programmable Gate Arrays), may become a valid alternative to synchronous FPGAs [29].

Since dataflow schemes are becoming increasingly important, efficient schemes to map computational structures to data-driven architectures are focus of recent research work. The increasing

number of available hardware resources requires a different view when mapping algorithms to reconfigurable computing architectures. Rather than the traditional resource constrained problem, this is now more a question of how to take advantage of the large number of available hardware resources [4]. When mapping loops, one of the most efficient optimization is loop pipelining. Loop pipelining usually leads to significant performance improvements. Since in most reconfigurable architectures the memory elements to implement the pipeline stages are already on chip, it is worth to be applied. As far as dataflow computing is concerned, efficient loop execution has been achieved through *dataflow software pipelining* [14][15], which strongly depends on efficient balancing techniques to achieve maximum throughput. In data-driven architectures one can take advantage of their intrinsic features to schedule operations based on the control and data flow. This paper presents *self loop pipelining (SLP)*, a technique to map computational structures in order that loop pipelining is dynamically achieved. *SLP* requires less balancing efforts (*i.e.*, reduced number of registers or reduced size FIFOs) than previous *dataflow software pipelining* techniques. The technique has been briefly introduced in [7], but a heuristic to apply *SLP* and results with complex examples have not been presented. This paper discusses the technique and shows substantial results when targeting the XPP architecture [2]. The major contributions of this paper are:

- (1) A loop pipelining scheme, *SLP*, that fully takes advantage of the dynamic scheduling naturally achieved by the handshake support is presented;
- (2) The use of *SLP* to pipeline nested loops is also shown;
- (3) Limitations and *SLP* suitability are presented and discussed;

- (4) Compiler techniques to include *SLP* in the optimizations repertory are presented;
- (5) Examples of applying *SLP* are illustrated and experimental evidence of the importance of the technique is shown for a number of benchmarks.

This paper is organized as follows. Next section briefly introduces data-driven issues. Section 3 explains *self loop pipelining* and shows how it can be applied to loops. Section 4 discusses compilation strategies to include *SLP*. Section 5 shows experimental results using the technique with a set of benchmarks. Section 6 describes the related work. Finally, section 7 gives some concluding remarks and delineates ongoing and future work.

2. DATA-DRIVEN ARRAY ARCHITECTURES

In the data-driven computational model, with static dataflow semantics [31], self-timed is achieved by a handshaking protocol and there is no need to statically schedule operations. Operations are executed as soon as data are available on their inputs and their output data have been consumed. The interconnection of functional units (FUs) naturally creates a pipelined dataflow structure and data streams may continuously flow through the structure without additional control or centralized schedulers.

A data-driven array mainly consists of a matrix of $N \times M$ PEs (processing elements) and interconnection resources (see in Figure 1 a simplistic scheme of the XPP architecture [2], as an example). Dataflow operations, which are implemented by PEs, include usual arithmetic and logic operations, and especial operations to deal with conditional branches (*e.g.*, SWITCH and

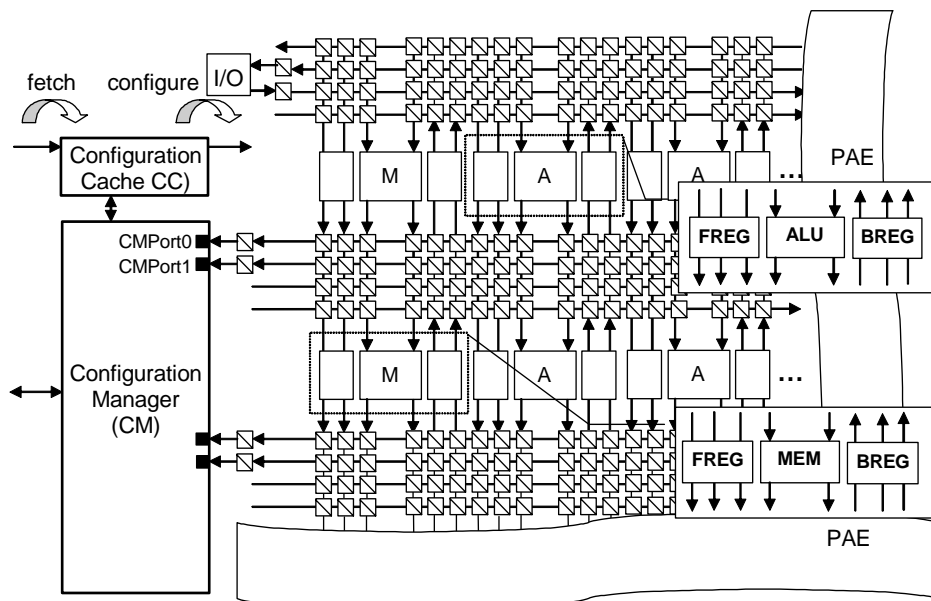


Figure 1. Simple diagram of the PACT XPP architecture. PAEs may include an ALU or a memory. The PAEs with memories are located in the left and rightmost columns of the array. Each PAE also includes two more elements: FREG and BREG. These elements can perform special operations, can be used as pipeline stages, or for vertical routing.

MERGE operations [31]). The execution of an operation (also known as the firing of a node) involves the removal of the data items in the input ports, and the production of data items in output ports. Some architectures use an enabling rule to fire the execution of the operation, *i.e.*, the execution besides the data-driven concept has forms of control. Such rules can be dependent on runtime decisions and permit to implement control flow (*e.g.*, loops, if-then-else structures, etc.). As aforementioned, a centralized control unit is not needed, and it is the data flow that dynamically imposes the execution of a particular operation (notice, however, that it is possible to statically define an order among operations using control tokens). Both data and control tokens may flow concurrently through the array structures, and fine-grain parallelism and multiple flows of control are naturally exposed.

SWITCH operations are used to route data items to one of the two outputs based on a control event (usually named control token). Standard MERGE operations do not have an enabling rule and just output the first data item present in one of the two inputs. There are, however, different implementations of MERGE. One MERGE uses a control signal to select between the two input data tokens and discards the data token (*i.e.*, the token is consumed but not copied to the output) not selected. According to the enable rule, there are also different MERGE implementations. One only triggers the execution when the control token and the two data tokens are ready, the other one triggers the execution as soon as the control token and the selected data token are ready (this type of evaluation is called lenient in [5]). Other special operators are specifically used to discard tokens, *e.g.*, the T- and F-Gates used in some dataflow machines, which only copy input data to output when the control token has value “true” or when has value “false”, respectively [12].

Enhanced data-driven arrays support the semantics of imperative programming languages to manipulate array variables (*e.g.*, load/store operations). When memories are located in special PEs, array structures are used to access them, and MERGE operations without discard are needed to multiplex data tokens. For instance, in the XPP the implementation of load/store operations is realized with array structures connected to the target memory (internal or external). Other architectures directly support load/store as PE operations (*e.g.*, WaveScalar [28]).

The array interconnections are responsible to flow data and control tokens. Their bit-width is a property dependent on the granularity of the PE. Some arrays include explicit lines for control events (*e.g.*, XPP). Other architectures merge control events in data lines. The interconnection topologies between PEs vary widely with the architecture. Some of the arrays use special horizontal and vertical connection resources (*e.g.*, XPP). Others explicitly use PEs for routing and provide interconnections between PEs in a mesh (*e.g.*, KressArray [17]) or in a hexagonal topology (*e.g.*, [22]).

Each configuration defines the operations in the PEs and the interconnections among them. Additional units are needed to control reconfiguration. For efficient support of the reconfiguration flow (*i.e.*, sequences of configurations), architectures may include an on-chip configuration manager (CM) and a configuration cache (CC) as is the case in the XPP (see Figure 1). Such amenities enable efficient and effective implementations of large programs by using temporal partitioning

(*i.e.*, programs are split in sequences of sections being each section implemented by the array resources), especially when the number of resources to map a given algorithm exceeds the available array resources [8].

This kind of architectures can be programmed with a structural language, a functional or dataflow language, or an imperative language.

3. SELF LOOP PIPELINING (SLP)

Software pipelining (see, *e.g.*, [1] for a survey) is a scheduling technique to pipeline loops (*i.e.*, overlap computations of subsequent loop iterations) and usually leads to significant performance improvements. Usual software pipelining techniques statically define epilogue, kernel, and prologue sections.

Pipelining loops in data-driven architectures does not need a scheduler of operations. It can be simply achieved by creating a structure connecting the operations of the loop body and the hardware structures responsible for the loop iterations (see Figure 2a and 2b). With this scheme, similar to *dataflow software pipelining* [14], efficient loop execution can be achieved (see Figure 2b). In Figure 2b, the CNT module represents a counter which starts at a given number and keeps incrementing it by a pre-defined value until a certain limit is not exceeded. Using a data-driven model with handshaking, the counter only furnishes a new count value if the previous one has already been consumed. To enable optimum software pipelining, full balancing of paths is required (see Figure 2b), *i.e.*, the counter indexing consecutive elements of the arrays A, B and C, requires that the two paths arriving to the destination memory where array C is located have the same number of pipeline stages. The two paths are related to the operations computing the data items to be stored in the array C, and to the address generation structure. Hence, to accomplish loop pipelining and maximum throughput, balancing is performed through the addition of pipeline stages (sequences of registers or FIFOs behaving in a data-driven manner) in certain paths of the dataflow structure¹.

This is partially needed because loop iterations are controlled by a centralized unit (as an image of the source imperative programming model). Since operations are triggered by the presence of data during runtime, the computational structures, needing explicit control from the hardware structures ensuring the loop iterations, do not require a centralized unit. Specifically, multiple hardware structures generating loop iterations can be used (see Figure 2c). This is the main idea of *self loop pipelining*. The original centralized counter, responsible for the control iterations of the FOR loop, is duplicated and two decentralized counters are responsible to control the loop behavior². The counters are decoupled and synchronize indirectly due to the data flow. As is depicted, there are now two independent paths furnishing the index value (i) to access array elements (see Figure 2c).

¹ Traditional loop pipelining implementations strongly depend on *pipeline balancing* techniques. This is also true in the context of *dataflow software pipelining*.

² Note that another valid *SLP* implementation would use three counters (one for each memory).

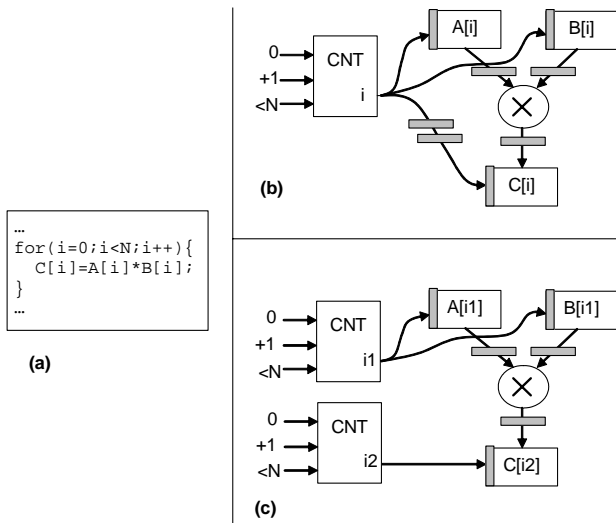


Figure 2. Loop pipelining on data-driven machines: (a) simple example - each array is mapped to a distinct memory; (b) traditional loop pipelining; (c) the proposed self loop pipelining technique. Rectangles in gray represent pipeline stages using the handshake protocol (they can be implemented as registers, FIFOs or queues).

Figure 3 shows a more complex example. In this case, the eight references to array variable x are implemented with a counter producing 8 indexes (ix) for each i produced by the counter

related to the FOR loop. In Figure 3, COPY represents a generator of N -copies of the value of the input data token to the output. The operators SE-PAR and PAR-SE are the operators presented in [7]. In this case they represent a self controlled 1:8 DEMUX (after demuxing 8 data items in the input, it re-starts with the next 8 items that may arrive) and a similar 8:1 MUX. As can be seen, the decoupled loop control structures are responsible to stream the data according to the consumption rate and maybe ahead of each other in the loop iteration space.

Albeit the possible presence of conditional paths taking different latencies on different loop iterations, most approaches on pipelining loops enforce, using balancing, a fixed and statically known loop body latency. This is the case of the *dataflow software pipelining* technique initially presented. Considering different latencies on the loop body requires much more complex implementations and might lead to difficulties to control the pipeline. This is especially true when operations are statically scheduled. One of the approaches considering, at some extension, different latencies is the one presented in [24]. However, that scheme requires complex centralized control units. In our approach there is no problem with different latencies on the loop body, since the pipelining rather than statically is dynamically achieved by the data-driven mechanism. The throughput of the loop is achieved by hardware structures, decoupled, decentralized and replicated (as has been illustrated in Figure 2 and Figure 3) and thus the technique does not require balancing of all the paths implementing conditional constructs (e.g., if-then or if-then-else). Obviously, some paths connecting functional units still need balancing to achieve maximum throughput.

As has been shown, *SLP* is relatively simple to apply to innermost

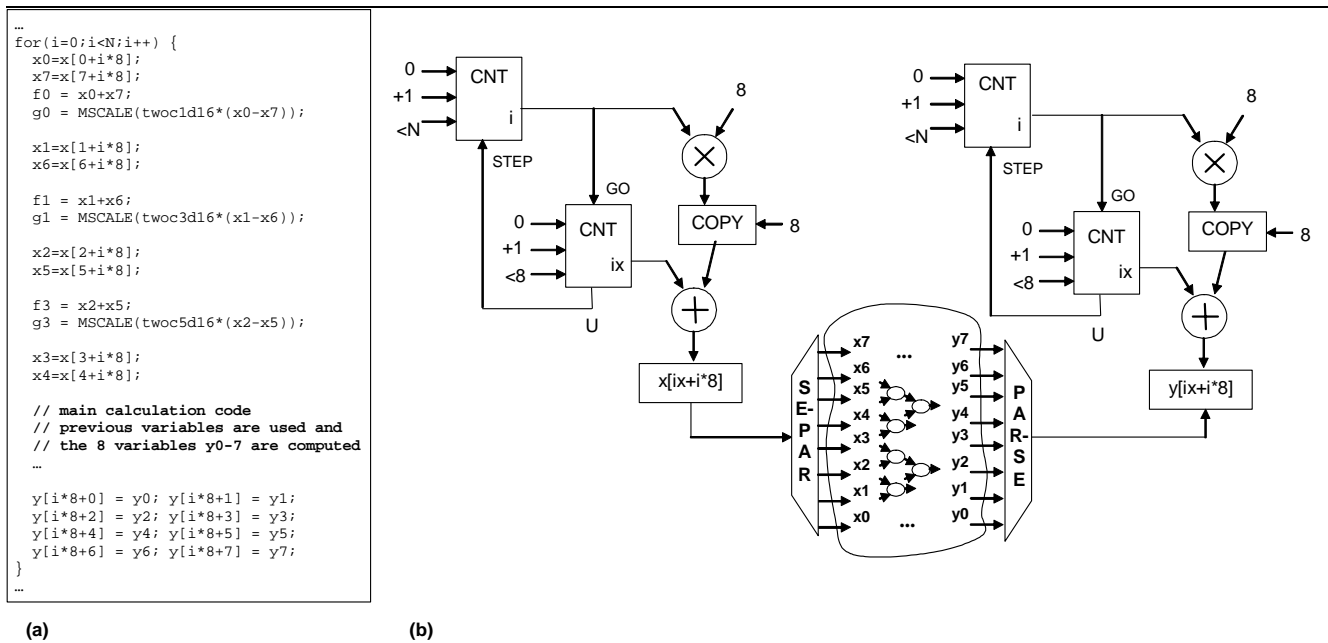


Figure 3. Example of applying self loop pipelining: (a) sample code of the *LeeDCT* example [18]; (b) possible SLP implementation. When a counter CNT reaches the limit generates an event on output U. An event arriving at input GO in counters starts a new counting after the previous one has finished.

loops, but is it applicable to nested loops? An example with two nested loops is shown in Figure 4. Figure 5 presents the dataflow graph (DFG) of a possible data-driven implementation of the example (notice that pipeline levels are omitted). Values in circles and rectangles represent a value generated once and as much as needed by an input, respectively. Figure 6 depicts the use of *SLP* to the innermost loop (Loop 2 in Figure 4). The duplicated structures are related to the computation of the indexes for loading values from the *sd* array (see line 4 in Figure 4) and to assignments to the scalar variable *sum* (line 4 in Figure 4). Figure 6 shows the DFG of the new structure. To apply the technique to the outer loop (Loop 1 in Figure 4), other duplications are employed in a similar manner. A structure can be used to assign zero to the scalar variable *sum* (line 2 in Figure 4), in each iteration of Loop 1. Two other structures can be used to control each of the structures referred above. Finally, a structure can be used to furnish the index *i* to the array *ac* (line 6 in Figure 4). Figure 7 depicts the DFG after applying the technique to the outermost loop, as well. The latter uses 6 hardware structures to control the loop iterations instead of the 2 original ones (see Figure 5). In this case, the duplication of various modules to control the iterations of each loop permits to start a new iteration of the outer loop before the end of its previous iteration, as is shown in the temporal diagrams of Figure 8.

```

...
1. for (i = 0; i < M; i++){ // Loop 1
2.   sum = 0;
3.   for(k = 0; k < N; k++){ // Loop 2
4.     sum += sd[k+i*N];
5.   }
6.   ac[i] = (sum >> SHIFT);
7. }
...

```

Figure 4. Median example: source code.

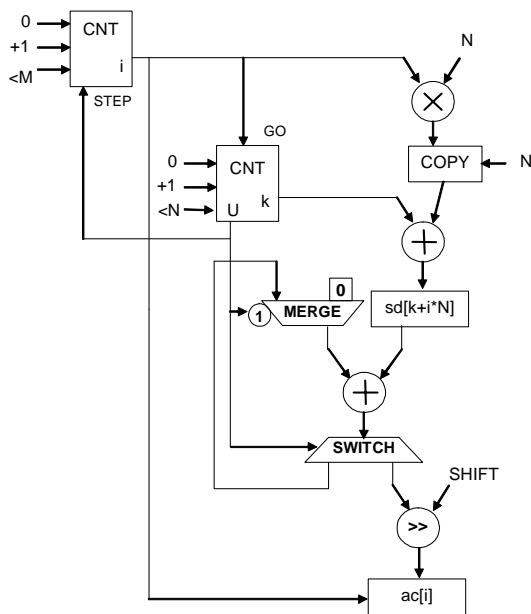


Figure 5. Median example: DFG representation of the nested loops in the example without using the *SLP* technique.

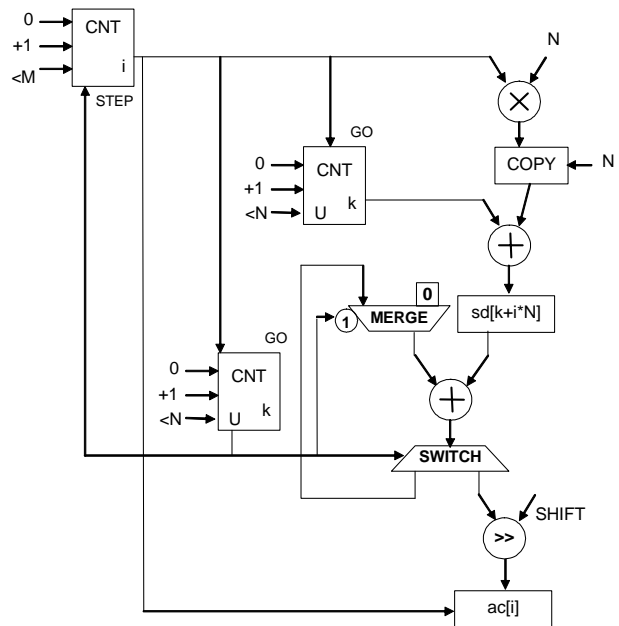


Figure 6. Median example: DFG after applying *SLP* to the innermost loop.

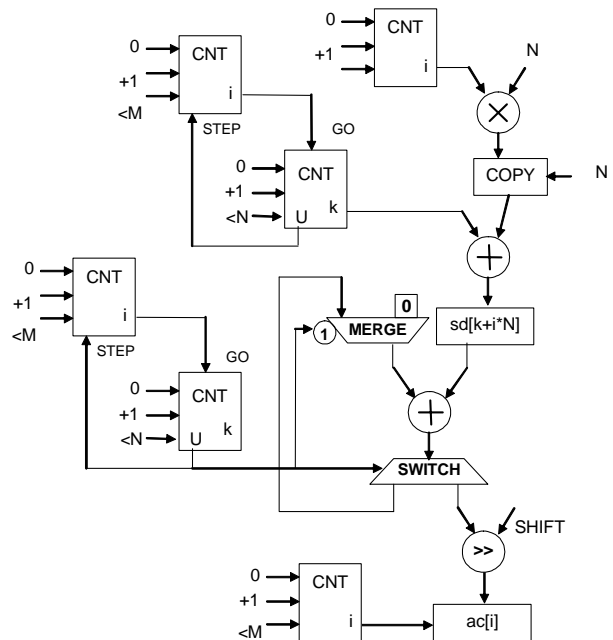


Figure 7. Median example: possible DFG representation after applying *SLP* to both inner and outermost loops.

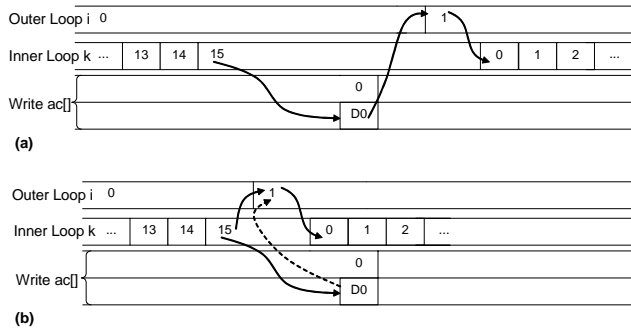


Figure 8. Temporal behavior for the *median* example (see Figure 4): (a) *SLP* applied to the innermost loop (see Figure 6); (b) *SLP* applied to the two nested loops (see Figure 7). The dashed arrow indicates that the outer loop starts a new iteration before the end of the previous one.

4. COMPILING TO DATA-DRIVEN ARCHITECTURES

Besides work on using dataflow languages [21] to program dataflow machines, some successful efforts translate imperative programming languages to dataflow models (e.g., [20][3][8]). Imperative programming languages can be transformed to the Program Dependence Web (PDW) [20], a representation that extends the Static Single Assignment (SSA) form [10] and the Program Dependence Graph (PDG) [13]. The PDW contains all the needed information for control-, data-, and demand-driven interpretation, and thus it can be used to generate the DFG akin to the required dataflow structure.

Mapping computational structures to data-driven machines is almost direct when straight-line code is input and each operation in the code can be directly implemented by a PE of the architecture. The handshaking mechanism permits to abstract the mapping from the timing details associated when the computational structures are implemented using a data-path and a centralized control unit (timing-driven model).

Selection points are explicitly represented in the SSA-form by Φ -functions. Those points can be directly implemented with MERGE operations with discard. The PDW uses the Gated Single Assignment (GSA) to generate the control conditions. Instead of using only the SSA Φ -functions, the PDW uses three types of functions (μ , γ , and η). μ -functions are used to represent selection points between loop carried values and loop initializations (MERGE operation). γ -functions are used to control forward data flow (MERGE operation). Finally, η -functions are used to control passage of values out of loop bodies (i.e., they are used to forward final data values after loop completion). Those η -functions can be translated to SWITCH nodes.

To enable the firing of some operations, control tokens are used, either directly (i.e., as a form of predicate execution controlled by guards) or as control mechanisms to cease the data flow. Architectures with PEs with firing rules enabled by special control inputs can implement almost directly predicated execution (e.g., XPP). When these types of firing rules are not directly supported, special operators can be added to enable/disable the data flow to destinations (to forward a copy or to discard the input

data item). Nevertheless, when speculative execution is used firing rules to enable/disable certain operations are not needed as long as data items generated in paths not taken are discarded. As opposite to non dataflow, where operations using a certain assignment are scheduled to time steps where data are already available, here we have to ensure that only data that must be used arrive to destination.

To achieve an optimized implementation several compiler optimizations are still required (see, for instance, [5]), such as array dependence analysis and elimination of redundant memory accesses (e.g., inter-iteration register promotion [9]). In data-driven arrays, *pipeline balancing* is usually performed during the place and route phases (as is the case in the XPP-VC compiler [8]).

As aforementioned, *self loop pipelining* is achieved by replicating N times the cyclic hardware structure responsible to control a certain loop. From each loop header the variable that controls the loop iterations is identified and the related DFG nodes are marked as being part of the loop control structure. The identification of those DFG structures starts by the μ node associated to the loop control variable and collects all the nodes constituting a path from the output of the μ node to one of the inputs. Then, a simple template matching scheme is used to expose counters. Although this scheme works well for well-behaved loops (WHILE, DO-WHILE, or FOR loops with loop control based on a comparison, between two scalar variables or a scalar variable and a literal, and increment or decrement operations, executed on every iteration), and most loops in DSP (digital signal processing) applications are of this kind, it does not resolve other types of loop control. Study to surpass this restriction is the subject of ongoing work.

After identification of each loop control structure, a DFG of the loop body is used to apply *SLP*. A simple heuristic depicted in Figure 9 is currently being used. To decide about the cloning of loop control structures, this heuristic uses an unconstrained ALAP (as late as possible) scheduling scheme. Duplication of structures is based on the costs to pipeline a path from the source (hardware structure responsible for the loop iterations) to a sink and the cost of the loop control structures to be cloned (see line 8 in Figure 9)³. This scheme is used to make a trade-off between forwarding data computed by loop control structures and re-computing them by duplicating those structures.

5. EXPERIMENTAL RESULTS

We have performed several experiments using the XPP [2] as the target architecture. The architecture uses a global synchronization clock. It performs each PE operation and communicates each data item between elements (i.e., PEs or interconnection registers) in a single clock cycle. We have used 32-bits as the XPP bit-width. The XPP can be programmed with a structural language named NML [2]. A tool to place and route designs in NML and to generate the binary code for each configuration and the code to program the configuration manager is provided. This tool tries to fully balance paths previously specified with NML directives. A higher abstraction level is provided by a compiler that translates programs in a C-subset to one or more NML designs [8]. The

³ When targeting the XPP, the control structures of well-behaved FOR type loops are implemented with counters, being each counter implemented by a single PAE of the architecture.

compiler is based on the *pipeline vectorization* technique [32] to pipeline well-behaved innermost loops.

```

Input: DFG with loop regions identified
Output: transformed DFG

1. foreach innermost loop  $L_k$  with  $Ctrl_k$  as
   loop control structure do
2.   Sinks=Find Sinks( $L_k$ ,  $Ctrl_k$ ); // list of DFG nodes
   of  $L_k$  directly connected to  $Ctrl_k$ 
3.   Determine ALAP Latencies (loop body  $L_k$ );
4.   Ordering Sinks according to ascendant
   ALAP Latencies(Sinks);
5.   for  $i$  in 1 to NumSinks-1 do
6.     Sink $_i$  = Sinks( $i$ );
7.     Sink $_{i+1}$  = Sinks( $i+1$ );
8.     if (PipelineCost(ALAP(Sink $_{i+1}$ )-ALAP(Sink $_i$ ))
        $\geq$  Cost( $Ctrl_k$ )) then
9.        $Ctrl_r$ =CloneAndConnect( $Ctrl_k$ , Sink $_{i+1}$ );
10.      for  $j$  in  $i+2$  to NumSinks-1 do
11.        Connect( $Ctrl_r$ , Sink $_j$ );
12.      end for;
13.    end if;
14.  end for;
15. end for;

```

Figure 9. Heuristic to apply SLP to innermost loops.

A number of benchmarks (see Table 1 for main characteristics) is used to test *self loop pipelining* and to compare it with a traditional loop pipelining scheme assisted with *pipeline balancing*, particularly the *pipeline vectorization* technique included in the XPP-VC compiler [8]. The benchmarks include DSP kernels and other more complex DSP tasks. They include benchmarks from Texas Instruments [30] (identified in Table 1 by TI), from the MediaBench repository (identified in Table 1 by MB) [19], and the *LeeDCT* benchmark (see [18]).

For experiments with *adpcm* and *LeeDCT* we use an XPP array with 16×16 ALU-PAE cells and two columns with 16 MEM-PAE cells each. For all the other experiments, an array with 8×8 ALU-PAEs and 2×8 MEM-PAEs is used.

With respect to the XPP-VC compiler, all the selected benchmarks have innermost loops that are pipelined, and we have done the experiments using all the efforts to execute fully-balanced implementations. Note that sometimes this is impossible to achieve due to the unavailability of the required amount of resources to insert the needed number of register stages.

Table 2 shows results using the *pipeline vectorization* included in XPP-VC and using the *SLP* scheme. The numbers representing clock cycles (#ccs) in Table 2 are related to the execution of each configuration in the array and do not include the reconfiguration time needed. The reconfiguration time depends on the structures of a certain configuration and is not of special interest for the comparison. The number of resources (#elements) represents the sum of the used elements of the array (FREG, BREG, ALU, or MEM). The numbers in the last two columns illustrate the percentage of resources and percentage of execution cycles between *pipeline vectorization* and *SLP*. Negative and positive

percentages mean lower or higher number of resources or clock cycles used by *SLP*, respectively.

The results show that our approach leads to performance improvements and to reductions in the number of the used resources over *pipeline vectorization*. For all but one case, the use of *SLP* achieves better performance (from 1.2% to 68.4% fewer execution cycles). With respect to used resources, most of the examples use fewer PAEs when *SLP* is used. For the Median example, we exploit the use of the technique on nested loops. The use of 4 (last but one result in last row of Table 2) or 7 hardware cyclic structures (last result in last row of Table 2) lead, respectively, to reductions on execution cycles of 25.5% and 46.2% than using *pipeline vectorization*. Obviously, since we have used examples with the innermost loops fully pipelined by the XPP-VC compiler, better improvements can be achieved with examples having innermost loops not pipelined by the XPP-VC.

The *adpcm* is one example with conditional structures in the loop body which are pipelined by *SLP* without enforcing the longest path latency of the loop body for each loop iteration. For the *weighted vector sum* benchmark, *SLP* has permitted reductions on the number of clock cycles of 54.4% (without using pipeline balancing) and 68.4% (using pipeline balancing).

As is recognized, traditional loop pipelining techniques depends heavily on *pipeline balancing* to achieve good performance. As an example, for the *median* example the results presented in Table 2, using *pipeline vectorization* and *pipeline balancing*, represent 86.4% fewer clock cycles using 18.2% more resources than the same example without *pipeline balancing*. With respect to *SLP*, no *pipeline balancing* has been needed.

Although a first reaction about innermost loops would spot that the duplication of the hardware structures, to control loop iterations, does not lead to performance improvements and/or resource savings, this is not the case since almost all of our experimental results indicate the opposite. Note, however, that the savings in the number of resources heavily depends on the type of operations directly supported by the target architecture. The improvements achieved with *SLP* have origins in the more relaxed *pipeline balancing* requirements and in the unneeded matching of branches on conditional constructs. We call the reader's attention to the fact that in the XPP it is not possible to implement pipeline stages greater than one with a single PAE unit. Thus, sequences of pipeline stages are implemented using various PAE units. This is one of the reasons for resource savings when using *SLP*.

There is strong evidence that using *SLP*, *pipeline balancing* may not be needed or may be required less aggressively, which also results in fast compilation. Furthermore, since we are duplicating and decoupling hardware structures, the number of memory stages needed for balancing is lower.

6. RELATED WORK

Software pipelining has been focus of intense research efforts [1]. It has been considered for both microprocessor and application specific architectures. Due to the need of a statically-defined scheduler, algorithms to schedule loop operations are used. One of the schemes is Modulo Scheduling, which can be efficiently performed by Rau's Iterative Modulo Scheduling (IMS) algorithm [26]. When targeting specific architectures, authors have considered two approaches: the existence of specific epilogue,

Table 1. Main characteristics of the benchmarks

Benchmark	Source	Data size and other parameters	Description
add_array	-	Arrays with 1024 elements	See Figure 2. Add two arrays.
Sad	-	Arrays with 256 elements	Sum of absolute differences (used in MPEG and JPEG)
adpcm decoder	MB	1,024 data values	Decoder audio algorithm.
LeeDCT	PVRG	8 × 8 elements	Compute the Discrete Cosine Transform (DCT). It uses two sequential loops.
Max	TI	256 elements	Calculate the maximum value in a vector.
auto correlation	TI	N=256, M=16	Perform M autocorrelations each of length N
weighted vector sum	TI	256 elements	Perform an N-element vector sum of two vectors with one vector weighted by a constant.
block move	TI	256 elements	Move the elements of one array to the other.
Gouraud	TI	128 elements	Gouraud shading of a scanline of pixels.
Median	TI	N=16, M=256	See Figure 4. Compute the median for each window of samples.

Table 2. Results with loop pipelining.

Benchmark	with <i>Pipeline Vectorization</i> (#elements/#ccs)	with <i>Self Loop Pipelining</i> (#elements/#ccs)	% resources	% clock cycles (ccs)
add_array	23/1069	21/1045	-8.7	-2.2
Sad	38/787	35/531	-7.9	-32.5
adpcm decoder	103/15,408	91/11,304	-11.7	-26.6
LeeDCT (1st loop)	237/41	144/33	-39.2	-19.5
LeeDCT (2 nd loop)	261/63	228/55	-12.6	-12.7
Max	36/1,041	23/1,029	-36.1	-1.2
auto correlation	47/17,033	47/16,658	0	-2.2
weighted vector sum	33/1,166	26/532	-21.2	-54.4
		31/368	-6.1	-68.4
block move	17/1,094	14/625	-17.6	-42.9
Gouraud	70/668	65/531	-7.1	-20.5
Median	33/1,107	31/1,062	-6.1	-4.1
		40/825	+21.2	-25.5
		47/596	+42.4	-46.2

kernel, and prologue hardware structures; or the use of predicated schemes to avoid the explicit epilogue and prologue structures.

Software pipelining has been applied to reconfigurable hardware platforms by several researchers, especially when mapping innermost loops to FPGAs (see, for instance, [32], [27]). Most approaches restrict loop pipelining to well-behaved loops. In some cases the freedom to use specific hardware structures provided loop pipelining techniques in the presence of conditional memory writes [24], for instance. Hardware structures to implement the restoring step that might be needed when a certain branch is not taken can be used. Research on new reconfigurable hardware architectures has also considered software pipelining. One of the approaches uses Rau's algorithm to pipeline innermost loops in the Garp architecture [6].

Pipeline vectorization [32] has been adapted for pipelining innermost loops when compiling C programs to the XPP [8]. The technique is applied to innermost loops without true and with regular loop-carried dependences (*i.e.*, those with constant

dependence distances). Pipelining of loops with conditional structures including array references is not considered.

The previous approaches use centralized control units obtained by static scheduling of the loop operations and therefore, when mapping to data-driven array architectures with handshake, may achieve the best performance.

More related to our work is *dataflow software pipelining* [14][15], a software pipelining scheme specially developed to pipeline innermost loops in dataflow machines. Compared to traditional software pipelining techniques, such approach naturally exploits the dynamic scheduling obtained by the data flow. A similar scheme to *dataflow software pipelining* has been included by Budiu et al. in a C compiler to asynchronous circuits [4][5].

All the previous approaches rely heavily on *pipeline balancing* to achieve maximum throughput by using FIFOs (*e.g.*, [5]) or connections of register stages (*e.g.*, [8]). They clamp the latency of the loop body to the longest path latency even if conditional branches are present. The technique proposed in this paper solves

these problems in a number of cases, and thus it can be an efficient optimization option when compiling to data-driven arrays (e.g., [8]) or application specific circuits using handshake (e.g., [4]). The use of our approach in the presence of loop-carried array dependences requires further studies. Although at a first glance this constraint seems to be quite restrictive, it has not disabled the mapping of representative DSP kernels as shown by the experimental results. In [5], token generators are used to control the amount of slip between two operations that may be ahead of the each other on loop iterations. It is a scheme to enforce that memory dependences are satisfied when loop iterations are dynamically scheduled, by explicitly representing the dependence distance. This scheme can also be efficiently used in conjunction with *SLP* to deal with loop-carried array dependences and is the subject of our ongoing work.

The use of explicit epilogue and prologue structures by usual software pipelining approaches might lead to resource problems, especially in array architectures without PEs with load/store operations. In that case, more references to an array imply additional PEs for the hardware structures responsible to interfacing with the memory where the array is stored. Moreover, explicit epilogue and prologue schemes also require, besides the PEs to implement the kernel, PEs to implement the computational structures of the epilogue and prologue. The *SLP* technique does not require epilogue and prologue structures, which may also be an important property when targeting data-driven arrays.

In the context of coarse-grained reconfigurable architectures, a software pipelining approach without explicit epilogue and prologue has also been recently used [25]. Although the approach integrates placement, scheduling, and routing, and is able to generate more than one configuration when hardware virtualization is needed, it was not proposed for data-driven array architectures and thus, compared to the approach presented in this paper, does not take advantage of dynamic scheduling.

7. CONCLUSIONS

This paper introduces a novel form of loop pipelining, named *self loop pipelining (SLP)*, suitable to pipeline a large set of loops when targeting data-driven reconfigurable arrays. It involves replication of the hardware structures responsible for the control of loop iterations. Loops are naturally executed in a pipelining fashion, with synchronization being achieved by the data flow. By dynamically scheduling operations, *SLP* can outperform statically scheduled software pipelining techniques. Therefore, the technique, an enhancement scheme for *dataflow software pipelining*, can be thought as an optimization to extend the repertory of loop optimizations that may be included in an advanced compiler targeting data-driven arrays.

The technique can be applied to DO-WHILE, WHILE, and FOR loops, including nested loops. Innermost loops with conditional constructs can also be pipelined without conservative pipelining implementations (which usually enforces the critical path length of the loop body). The technique requires less sophisticated balancing efforts than previous software pipelining techniques.

The technique has been applied when mapping a number of benchmarks to the XPP. The results, by achieving performance improvements and in some cases even fewer required resources, strongly prove its importance.

Ongoing work aims techniques to overcome loop-carried array dependences and further experiments with other data-driven architectures.

8. ACKNOWLEDGMENTS

This work has been partially supported by the Portuguese Foundation for Science and Technology (FCT) - FEDER and POSI programs - under the CHIADO project (POSI/CHS/48018/2002). The author gratefully acknowledges the licenses and the development tools for the XPP donated by PACT XPP Technologies, Inc. The author would like to thank Pedro Diniz for his suggestions and comments. Also important have been email exchanges with Mihai Budiu, regarding some issues related to the content of this paper.

9. REFERENCES

- [1] V. Allan, R. Jones, R. Lee, and S. Allan. Software Pipelining. in *ACM Computing Surveys*, Vol. 27, Issue 3, pages 367-432, September 1995.
- [2] V. Baumgarte, et al. PACT-XPP – A Self-reconfigurable Data Processing Architecture. In *Journal of Supercomputing*, Kluwer Academic Publishers, vol. 26, issue. 2, pages 167-184, September 2003.
- [3] M. Beck, R. Johnson, and K. Pingali. From Control Flow to Dataflow. in *Journal of Parallel and Distributed Computing*, Volume 12, Issue 2, pages 118–129, June 1991.
- [4] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein. Spatial Computation. in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, Boston, MA, USA, ACM Press, pages 14-26, October 2004.
- [5] M. Budiu. *Spatial Computation*. Ph.D. Thesis, CMU CS Technical Report CMU-CS-03-217, December 2003.
- [6] T. Callahan and J. Wawrzynek. Adapting Software Pipelining for Reconfigurable Computing. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'00)*, San Jose, CA, USA, ACM Press, pages 57-64, November 17-19, 2000.
- [7] J. Cardoso. Self Loop Pipelining and Reconfigurable Dataflow Arrays. in *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS IV)*, Andy Pimentel and Stamatis Vassiliadis (eds.), Springer-Verlag, LNCS 3133, pages 234-243, 2004.
- [8] J. Cardoso and M. Weinhardt. XPP-VC: A C Compiler with Temporal Partitioning for the PACT-XPP Architecture. in *Proceedings of the 12th International Conference on Field Programmable Logic and Applications (FPL'02)*, Springer-Verlag, LNCS 2438, pages 864-874, 2002.
- [9] S. Carr, D. Callahan, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI'90)*, ACM Press, pages 53-65, June 1990.
- [10] R. Cytron, et al. Efficiently Computing static single assignment form and the control dependence graph, In *ACM*

- Transactions on Programming Languages and Systems*, vol. 13, no. 4, pages 451-490, October 1991.
- [11] A. DeHon. Very Large Scale Spatial Computing. In *Proceedings of the Third International Conference on Unconventional Models of Computation (CUMC'02)*, C. Calude, M. Dinneen, F. Peper (Eds.), Springer-Verlag, LNCS 2509, pages 27-37, October 15-19, 2002.
- [12] J. Dennis and D. Misunas. A computer architecture for highly parallel signal processing. in *Proceedings of the ACM National Conference*, ACM, New York, pages 402-409, November 1974.
- [13] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. in *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pages 319-349, July 1987.
- [14] G. Gao. *A Code Mapping Scheme for dataflow Software Pipelining*. Kluwer Academic Publishers, 1991.
- [15] G. Gao and Z. Paraskevas. Compiling for Dataflow Software Pipelining. in *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nicolau and D. Padua (eds.), Cambridge, Massachusetts, MIT Press, pages 275-306, 1991.
- [16] R. Hartenstein. A Decade of Reconfigurable Computing: a Visionary Retrospective. In *Proceedings of the International Conference on Design, Automation and Test in Europe (DATE'01)*, Munich, Germany, pages 642-649, March 12-15, 2001.
- [17] R. Hartenstein, R. Kress, and H. Reinig. A Dynamically Reconfigurable Wavefront Array Architecture. in *Proceedings of the International Conference on Application Specific Array Processors (ASAP'94)*, pages 404-414, August 22-24, 1994.
- [18] A. Hung. *Source C code for Lee DCT*. Stanford University Portable Video Research Group (PVRG), 1993, <http://www.ifi.uio.no/in383/src/jpeg/PVRG-JPEG-1.2.1/leedct.c>
- [19] MediaBench Home, <http://cares.icsl.ucla.edu/MediaBench/>
- [20] K. J. Ottenstein, R. A. Ballance, and A. B. Maccabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI'90)*, ACM Press, pages 257-271, 1990.
- [21] W. Johnston, J. Hanna, and R. Millar. Advances in dataflow programming languages. in *ACM Computing Surveys*, vol. 36, issue 1, ACM Press, pages 1-34, 2004.
- [22] I. Koren, et al. A Data-Driven VLSI Array for Arbitrary Algorithms. in *IEEE Computer*, pages 30-43, October 1988.
- [23] S. Kung, et al. Wavefront Array Processors - Concept to Implementation. in *IEEE Computer*, vol. 20, no. 7, pages 18-33, July 1987.
- [24] T. Maruyama and T. Hoshino. A C to HDL compiler for pipeline processing on FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*, pages 101-110, IEEE CS Press, 2000.
- [25] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Exploiting Loop-Level Parallelism on Coarse-Grained Reconfigurable Architectures Using Modulo Scheduling. in *Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, Munich, Germany, pages 10296-10301, March 3-7, 2003.
- [26] B. Rau. Iterative Module Scheduling: An Algorithm for Software Pipelining Loops. In *Proceedings of the ACM 27th Annual International Symposium on Microarchitecture (MICRO-27)*, ACM Press, New York, pages 63-74, 1994.
- [27] G. Snider. Performance-constrained pipelining of software loops onto reconfigurable hardware. In *Proceedings of the ACM 10th International Symposium on Field-Programmable Gate Arrays (FPGA'02)*, ACM Press, New York, pages 177-186, 2002.
- [28] S. Swanson, et al. WaveScalar. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO-36)*, San Diego, CA, USA, IEEE Computer Society Press, pages 291, December 3-5, 2003.
- [29] J. Teifel and R. Manohar. An Asynchronous Dataflow FPGA Architecture. in *IEEE Transactions on Computers*, Volume 53, Issue 11, pages 1376 - 1392, November, 2004.
- [30] Texas Instruments, Inc. TMS320C6000™ Highest Performance DSP Platform. 1995-2003, <http://www.ti.com/sc/docs/products/dsp/c6000/benchmarks/62x.htm#search>
- [31] A. Veen. Dataflow machine architecture. in *ACM Computing Surveys*, Vol. 18, Issue 4, pages 365-396, December, 1986.
- [32] M. Weinhardt and W. Luk. Pipeline Vectorization. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 2, pages 234-233, February, 2001.