Journal of Universal Computer Science, vol. 13, no. 3 (2007), 419-439
submitted: 30/11/06, accepted: 15/2/07, appeared: 28/3/07 © *J.UCS*

On Pipelining Sequences of Data-Dependent Loops

Rui M. M. Rodrigues
(INESC-ID/IST, Portugal
ruirodrigues@gmail.com)

João M. P. Cardoso
(INESC-ID/IST, Portugal
jmpc@acm.org)

Abstract: Sequences of data-dependent tasks, each one traversing large data sets, exist in many applications (such as video, image and signal processing applications). Those tasks usually perform computations (with loop intensive behavior) and produce new data to be consumed by subsequent tasks. This paper shows a scheme to pipeline sequences of data-dependent loops, in such a way that subsequent loops can start execution before the completion of the previous ones, which achieves performance improvements. It uses a hardware scheme with decoupled and concurrent data-path and control units that start execution at the same time. The communication of array elements between two loops in sequence is performed by special buffers with a data-driven, fine-grained scheme. Buffer elements are responsible to flag the availability of each array element requested by a subsequent loop (i.e., a ready protocol is used to trigger the execution of operations in the succeeding loop). Thus, the control execution of following loops is also orchestrated by data availability (in this case at the array element grain) and out-of-order produced-consumed pairs are permitted. The concept has been applied using *Nau*, a compiler infrastructure to map algorithms described in Java onto FPGAs. This paper presents very encouraging results showing important performance improvements and buffer size reductions for a number of benchmarks.

Keywords: Loop Pipelining, Compilation, Hardware Schemes, FPGAs
Categories: D.3.4, C.1.2, C.1.4, C.5.4, B.1.2, B.5.2, B.7.2

1 Introduction

Dedicated architectures, implemented in FPGAs, are being used to satisfy certain algorithm and application requirements (e.g., performance demands) [Gokhale, 05]. Those architectures are usually implemented by hardware design experts, due to the lack of efficient push-button compilation flows using software programming languages as the starting point. Although FPGAs are rich hardware resource platforms, their broad acceptance suffers from the lack of compilation flows and techniques able to efficiently map software languages onto FPGA flexible and programmable hardware resources. This is also a required support in order to deal with time-to-market pressures. The increasing number of available FPGA resources stimulates the research of new ideas and hardware schemes. Design decisions are now not fully driven by resource limitations, as has been in the past, but more focused on "how to take advantage of the very large number of available resources?".

Particularly, optimizing the implementations for speed may result in larger hardware but with unmatched performance.

Since the computationally intensive parts of most algorithms are related to loop behavior, it is comprehensible that most optimization techniques have been focused on the generation of dedicated architectures to efficiently implement inner loops and loop nested structures. Loop pipelining is one of such optimization techniques. It permits to generate architectures able to start subsequent iterations of a loop before the end of the previous iterations.

Sequences of loops occur in most applications. Some sequences have nested loops with large iteration spaces and take long runtimes. Mapping such loop sequences to hardware structures that maintain the imperative model (i.e., only after a loop or a set of nested loops finishes execution, subsequent loops start their execution) leads in most cases to a sub-optimal hardware implementation, as far as performance is concerned. When those sequences of loops are data-dependent, they cannot be implemented as fully parallel tasks. On those sequences, a stage (herein identifying a loop or a set of nested loops) usually produces data consumed by a subsequent stage. Since a producer stage is usually able to produce data required by the consumer stage before the producer finishes execution, they would gain if concurrent execution is accomplished in a pipelining mode. Approaches to pipeline sequences of data-dependent loops have been recently addressed in [Ziegler, 03] and [Rodrigues, 05a].

The idea presented in this paper decouples the control units of each stage and uses inter-stage buffers to signal the availability of array elements to the subsequent stage. Doing that, the scheme achieves pipelining of sequences of loops, even when array elements produced by a set of loops are not consumed in the same order by subsequent loops. The inter-stage buffers used in this paper, for fine-grain synchronization of the pipelining stages, is similar to the empty/full tagged memory scheme used in the context of shared memory multiprocessor architectures [Smith, 81]. Although with similarities, the technique presented in this paper distinguishes from the work in [Ziegler, 03] by using a fine-grained synchronization scheme between stages and a hash-function concept tailored to reduce the inter-stage buffers. The main contributions of this paper are:

- A technique to pipeline sequences of data-dependent loops using a fine-grain synchronization scheme is presented and a discussion of its applicability is also included;
- A scheme to reduce the size of the memory buffers for inter-stage pipelining is also proposed and evaluated. The scheme permits to substantially reduce the storage requirements of the original architectures for the examples being used;
- An experimental setup, suitable to evaluate the technique by cycle-based, behavioral, RTL (Register Transfer Level) simulation, is introduced. The setup is used to determine the size of the buffers needed to communicate data between pipelining stages. This is performed by monitoring the data communication conflicts that can occur if an inappropriate buffer size is used;
- Experimental results on applying the technique to a number of benchmarks are shown and comparisons are drawn;

This paper is structured as follows. Next section presents the technique to pipeline data-dependent sequences of loops (referred herein as stages). Section 3 explains the experimental setup used to test the technique with a number of benchmarks. Section 4 shows the obtained results and Section 5 presents the related work. Finally, last section concludes the paper and introduces future work.

2 Pipelining Sequences of Data-Dependent Loops

The main idea behind the pipelining of sequences of data-dependent loops (PSL) is to overlap the execution of iterations of distinct loops, located in sequence, (i.e., iterations of a subsequent loop start before a previous loop finishes execution).

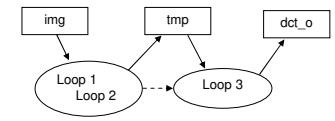
As an example of typical sequences of loops presented in real code, consider the Fast DCT (Discrete Cosine Transform) algorithm, available from Texas Instruments Inc., shown in Figure 1. The algorithm calculates the DCT of an input image using blocks of 8×8 pixels of the image. The algorithm includes two stages of data-dependent loops. The first stage consists of two nested loops (see Loops 1, 2 in Figure 1(a) and (b)) and the second stage consists of a single loop (Loop 3 in Figure 1(a) and (b)). A typical architecture for this algorithm, obtained by compilation, uses a global FSM (Finite State Machine) to control the data-path (including memory accesses), and to reflect the behavior of the loops, executing them in sequence (see Figure 1(c) and (d)).

As can be seen, the inter-stage data communication is performed using the array *tmp*. The data items of the array *tmp* are produced in the first stage and used by the second stage. Those data items are produced in Loop2 by the indexing order: 0, 8, 16, 24, 32, 40, 48, 56, 1, 9, 17, (…) and are then consumed in Loop3 by the indexing order: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (…). Since data items of array *tmp* produced in the initial iterations of Loops 1, 2 are requested in the initial iterations of Loop 3, the execution of Loop 3 can be ideally started before Loops 1, 2 finish execution. This type of implementation produces an overlapped execution of the stages and may achieve important performance improvements (see Figure 2). Instead of coupled and global, control and data-path units, an implementation of this concept uses a data-path and a control unit for each of the stages (see Figure 2(b)). Those data-path and control units can then be executed concurrently. Inter-stage buffers are used to communicate array elements between the stages and to flag subsequent stages of the availability of data items (fine-grain synchronization). This is the main idea of the PSL loop pipelining technique.

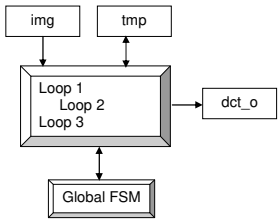
An inter-stage communication scheme must allow that array elements being produced in a stage became available to the subsequent stage. Note that in this example, the order of array elements being produced is not the same as the order of array elements being consumed and a scheme based on FIFOs (First-In-First-Out) cannot be used to communicate data between the two stages without additional overhead. Next sub-section explains the use of the inter-stage buffers.

```
#define M 64
#define num_fdcts 7500 //image of 800x600
#define SIZE num_fdcts*M

short img[SIZE];
short tmp[SIZE];
short dct_o[SIZE];
…
i_1 = 0;
for (i=0; i<num_fdcts; i++) {  // Loop 1
 for (j=0; j<N; j++) {         // Loop 2
  f0=img[ 0+i_1]; f1=img[ 8+i_1];
  f2=img[16+i_1]; f3=img[24+i_1];
  f4=img[32+i_1]; f5=img[40+i_1];
  f6=img[48+i_1]; f7=img[56+i_1];
  // computations
  …
  tmp[ 0+i_1]=F0;     tmp[ 8+i_1]=F1>>13;
  tmp[16+i_1]=F2>>13; tmp[24+i_1]=F3>>13;
  tmp[32+i_1]=F4;     tmp[40+i_1]=F5>>13;
  tmp[48+i_1]=F6>>13; tmp[56+i_1]=F7>>13;
  i_1++;
  }
 i_1 += 56;
}

i_1 = 0;
for (i=0; i<N*num_fdcts; i++) { // Loop 3
 f0=tmp[0+i_1]; f1=tmp[1+i_1];
 f2=tmp[2+i_1]; f3=tmp[3+i_1];
 f4=tmp[4+i_1]; f5=tmp[5+i_1];
 f6=tmp[6+i_1]; f7=tmp[7+i_1];
 // computations
 …
 dct_o[0+i_1]=F0r; dct_o[1+i_1]=F1r;
 dct_o[2+i_1]=F2r; dct_o[3+i_1]=F3r;
 dct_o[4+i_1]=F4r; dct_o[5+i_1]=F5r;
 dct_o[6+i_1]=F6r; dct_o[7+i_1]=F7r;
 i_1 += 8;
}
```

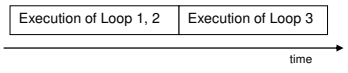
(a)



(b)



(c)



(d)

Figure 1: Fdct example: (a) Part of the fdct code; (b) Task graph where bubbles represent sets of loops in the code and rectangles represent the array variables; (c) Block diagram of a typical architecture, with global control and data-path units and the use of one distinct memory for each array variable, generated by a compiler; (d) Execution of the loops according to the previous block diagram.

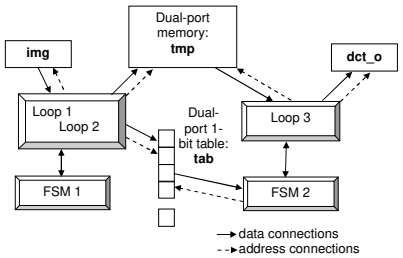
2.1 Schemes for buffering and communication between stages

When data is consumed in the same order of the data being produced, FIFOs having on each stage an array element can be used to communicate data between stages. This is the scheme used in [Ziegler, 03] in the context of data communication between designs on different FPGAs. However, that approach requires an analysis to determine the number of FIFO stages needed to maintain high throughput or a

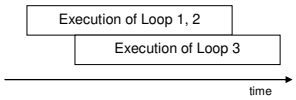
handshake protocol in both sides of the FIFOs to temporally stall each stage in the sequence [Ziegler, 06]. With that approach (seen as a coarse-grained synchronization scheme) and when considering out-of-the-order producer/consumer rates, the width (number of array elements in each FIFO stage) of the stages of the FIFOs needs to be determined and must have a sufficient size in order to ensure a correct functionality. Note that in this case, all the array elements on each FIFO stage must be consumed before a new FIFO stage is considered. Thus, the approach by [Ziegler, 06] may need FIFOs with a very long width. To circumvent this limitation, the idea in this paper addresses other inter-stage mechanisms as is explained next.

```
…
boolean tab[SIZE]={0, 0,…, 0};
…
for(i=0; i<num_fdcts; i++){//Loop 1
 for(j=0; j<N; j++){       //Loop 2
   // loads
   // computations
    ...
   // stores
   tmp[48+i_1] = F6 >> 13;
   tab[48+i_1] = true;
   tmp[56+i_1] = F7 >> 13;
   tab[56+i_1] = true;
   i_1++;
 }
 i_1 += 56;
}

i_1 = 0;
for (i=0; i<N*num_fdcts; i++){//Loop 3
 L1: f0 = tmp[0+i_1];
    if(!tab[0+i_1]) goto L1;
 L2: f1 = tmp[1+i_1];
    if(!tab[1+i_1]) goto L2;
 // remainig loads
 // computations
 …
 // stores
 i_1 += 8;
}
```

(a)



(b)



(c)

Figure 2: The fdct example using PSL and tables for inter-stage buffers: (a) Insertion of the behavior needed in the original code to explain how PSL works in this case; (b) Block diagram of the architecture using two decoupled and concurrent control and data-path units; (c) Overlap of the execution of the two loop sets using the PSL pipelining technique.

One possible implementation of PSL is illustrated in Figure 2 for the *fdct* example. It uses a dual-port memory to store the *tmp* array and a dual-port 1-bit table (*tab*) to store flag values representing the absent or availability of array elements (similar to ready signals). Herein, the *tab* table is referred as the "empty/full table". Each flag indicates if an array element has been already produced (by stage with

Loops 1,2) and thus can be consumed by the data-path of the second stage (Loop 3). The two concurrent FSMs are used to control each stage. When loading an element of the *tmp* array, the second FSM only has a transition to a new state if and only if the correspondent flag in table *tab* has value one, which indicates that the associated array element is ready/available. Otherwise, the FSM will stay in the current state waiting for data availability.

This scheme uses, for each array being produced-consumed, inter-stage buffers containing a buffer and an empty/full table, both with the same size of the correspondent array. This may require off-chip inter-stage buffers in order to have the memory space needed to store temporary data related to image processing stages, for instance. One better solution would be to enable on-chip inter-stage buffers by using a hash-based scheme instead of the "empty/full table" previously explained.

The size of the inter-stage buffer needs to be sufficient to accommodate the data elements needed at each execution cycle to preserve the functionality of different producer/consumer rates and ordering. An ideal inter-stage buffer (called herein as perfect hash) would need a fully associative memory. Such type of memory would lead to inter-stage buffers with the minimal number of memory positions. In this case, the producer stage would write the value and associate it with the related address. The consumer stage would check for the availability of the address being read, and in the case of a hit would read the array element associated with that address and remove both the array element and the address from the buffer. However, that solution is typically impractical in terms of hardware resources and/or in terms of the read/write execution cycles overhead.

A more feasible solution uses buffer units indexed by values, obtained by a function of the addresses generated by both stages (represented by $\mathcal{H}$ in Figure 3). Using a hash-based scheme for the *fdct* example, the *tmp*/*tab* pair is replaced by a buffer (*hash_t*) addressed via the hash function ($\mathcal{H}$). Figure 4 illustrates a block diagram of the inter-stage buffers using hash-functions employed in the context of this work. The hit-miss flag in Figure 4 is used by the subsequent stage in order to wait for data availability in the case of a miss or to proceed in the case of a hit. According to Figure 4: (a) a write operation stores the input data in the buffer position determined by the hash function $\mathcal{H}$ and flags in the same position the empty/full table with $\mathcal{R}$ (it corresponds to a simple store of '1' when an empty/full table with size L=1 is used); (b) a read operation loads the output data from the buffer position determined by the hash function $\mathcal{H}$ and checks if the item stored on that position of the empty/full table flags data availability ('1' in the cases of empty/full tables with sizes L=1 are used); (c) the read operation also updates the correspondent item of the empty/full table using a function $\mathcal{T}$ (stores a '0' when an empty/full table with size L=1 is used).

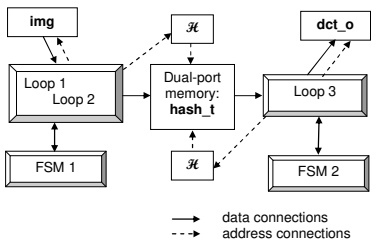
The function $\mathcal{H}$ is similar to the hash functions since it performs a transformation of a key k (here, it is an integer with the address) into a value, $\mathcal{H}(k)$, indexing a buffer element. The function $\mathcal{H}$, present on both stages to pipeline, need to be scheduled in the memory cycles of the read and write operations in order to maintain the initial execution cycles (i.e., without needing to add an additional overhead to the memory write/read operations). Thus, those functions need to be simple to implement and, if possible, should not lead to degradations of the maximum clock frequency previously achieved by the original architecture (without inter-stage pipelining). This

scheme also needs a table of flags with the same size of the correspondent inter-stage buffer.

From the hash functions proposed in the literature (see [Knuth 73]), the hash function $\mathcal{H}(k) = k$ MOD m, where m represents the size of the hash-table, is the simplest one. When m is a power of two integer, this hash function is simplified to the use of the $\lceil \log_2(m) \rceil$ least significant bits of the integer k. Such hash functions do not need additional hardware resources and do not impose an additional delay because their implementation only requires simple connections. Although the use of buffers with a power of two size leads to a number of memory positions never used, their implementation with memories (even if they are on-chip RAMs, e.g., block RAMs) always requires such power of two sizes.

```
…
hash_t[SIZE];
…
for(i=0; i<num_fdcts; i++){// Loop 1
 for(j=0; j<N; j++){       // Loop 2
    // loads
    // computations
         …
    // stores
    hash_t.put(H(48+i_1), F6 >> 13);
    hash_t.put(H(56+i_1), F7 >> 13);
    i_1++;
 }
 i_1 += 56;
}

i_1 = 0;
for (i=0; i<N*num_fdcts; i++){// Loop 3
 L1: f0 = hash_t.remove(H(0+1_1));
     if(f0==null) goto L1;
 L2: f1 = hash_t.remove(H(1+i_1));
    if(f1==null) goto L2;
 // remaining loads
 // computations
 …
 // stores
 i_1 += 8;
}
```

(a)



(b)

Figure 3: The fdct example using PSL and hash-based inter-stage buffers: (a) Addition of the behavior needed in the original code to explain how PSL works in this case; (b) Block diagram of the architecture using two decoupled and concurrent FSMs.

This type of implementation requires no collision among writes (i.e., a write from the producer must write to a vacant buffer position[1]). The calculation of the sizes of the inter-stage buffers that guarantee no collisions is not trivial. It depends on the

[1] Assuming a single write of the producer to each array element being read by the consumer stage.

producer/consumer ordering and on the write/read rates achieved by the producer and the consumer for a particular algorithm implementation. In the approach presented in this paper, those sizes are automatically determined by cycle-based RTL simulation of the generated architecture including inter-stage buffers able to monitoring the read/write operations and to perform the required calculations. In the end of the simulation, the output sizes are then used as parameter values to synthesize the final architecture. The potential conflicts by using simple hash-functions are avoided by selecting the right buffer size taking into account the producer/consumer rates and ordering.

Figure 5 shows the two algorithms used in the inter-stage buffer component to calculate the size needed for the buffer. The algorithm in Figure 5(a) calculates a lower bound that would be needed if a perfect hash function was used. The minimum size of the buffers to avoid conflicts using the hash function simplified to the use of a number of the least significant bits of the addresses is calculated using the algorithm in Figure 5(b). To reduce the search space, those calculations are started with the lower bound determined by the first algorithm (Figure 5(a)).

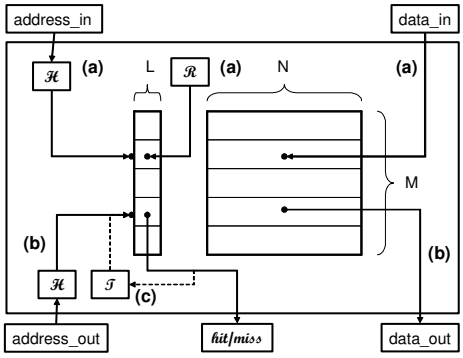


Figure 4: Inter-stage buffer structure with a hash-function.

2.2 The case when consumer stages read more than once the same array element

When a consumer stage reads more than once the same array element, previously produced only once or less than the number of times read by the consumer, data reuse can be applied to the consumer stage in order to remove multiple reads to the same array element. The other solution is to use inter-stage buffers with empty/full tables of width L greater than 1. In this case, L must be sufficient to flag with "full" until all the reads to the same array element have been done (equation (1) presents the empty/full conditions). Note that in the case of a single read to the same array element an L with value 1 is used, and for each read to the inter-stage buffer leading to an hit the correspondent flag is changed to '0'. In the case of multiple reads mentioned here, the empty/full table could store integer values assigned in the first time to the maximum value of reads to the same array element that are performed. In this case, each successful read could decrement the value in the table.

Although the later is a possible solution, it requires subtract operations that would undoubtedly impose delays on the read operations of the architecture. To alleviate this problem, the solution presented in this paper requires L having the value of the number of maximum multiple reads. Each time the producer stores an array element, the correspondent table element is assigned to an L bit word with all the L bits equal to '1'. For each successful read the table element is shifted by one bit to the right. To test the availability of data the consumer only has to check the LSB (least significant bit) of the word read from the empty/full table.

```
1.  int size = 0;
2.  while executing do
3.    writing: store.value(address_in, hash_t);
4.      size = maximum(size, hash_t.current_size);
5.    reading: remove.value(address_out, hash_t);
6.      // read_accesses[address_out]++;
7.  end while;

8.  return size;
```

(a)

```
1.  int size = size obtained by perfect hashing;
2.  int bits_needed = ⌈log2(size)⌉;

3.  while executing do
4.    int ad_in = ℋ(address_in, size);
5.    int ad_out = ℋ(address_out, size);

6.    writing:
7.      while(hash_t.contains(ad_in)) do
8.        bits_needed++;
9.        size = 2^bits_needed;
10.       ad_in = ℋ(address_in,size);
11.     end while;
12.     hash_t.put(ad_in, data_in);
13.     // hash_t.put(ad_in, data_in, read_accesses[address_in]);
14.   reading:
15.     hash_t.remove(ad_out)
16.     // if(read_accesses[address_out] >= 1)
17.     //    data_out = hash_t.get(ad_out);
18.     //    read_accesses[address_out]--;
19.     // else hash_t.remove(ad_out);
20. end while;

21. return size;

22. int ℋ(int k, int M) {
23.   return (k MOD M);
24. }
```

(b)

Figure 5: Algorithms to determine the size of the inter-stage buffers for the hash-based PSL scheme: (a) calculating the minimum size needed if a perfect-hash was used; (b) calculating the size needed to avoid collisions using a simple hash function.

$$\begin{cases} \text{Empty, if } EF_{table}(\text{index}) == 0 \\ \text{Full, if } EF_{table}(\text{index}) > 0 \end{cases} \quad (1)$$

The minimum required size of the inter-stage buffers are determined by RTL simulation with similar algorithms to the ones illustrated in Figure 5. In this case the commented line with number 6 in Figure 5(a) is included in order to determine the number of reads of the consumer to each array element. Those numbers are stored in a table that is then used by the algorithm in Figure 5(b), where line 12 is substituted by line 13 and line 15 is substituted by lines 16 to 19.

2.3 Comments on the PSL Technique

The use of the PSL technique leads theoretically to an upper bound speed-up determined by:

$$\text{speed - up} \leq \text{Latency(architecture with all stages)}/\max(\text{Latency}_i) \quad (2)$$

where Latency_i represents the execution time in cycles of stage i ($1 \leq i \leq N$) and N represents the number of stages of the algorithm. Note that this theoretical upper bound must be calculated using individual latencies of the stages when an independent architecture is used, because in this case there is no need to multiplex data and address sources among the producer and the consumer stages, which is close to the architecture using PSL. Using PSL, each stage is able to access concurrently the memories for inter-stage data communication. In the case of stages with equal execution time, the theoretically upper bound speed-up is about N (it may slightly surpass this value due to the fact that stand-alone stages may execute faster than when integrated in a global architecture).

Although the PSL technique can be important to extend even more the repository of optimization techniques, it cannot be directly applied in the following cases: when a producer stage writes more than once to the same array element not consumed yet; and when one or more of the array elements being read will not be written by the producer. Note, however, that those cases can be solved by changing the program accordingly.

Although a simple hash function is always envisaged, since it is difficult to justify a more complex one due to the influence on performance degradation it may have, other hash functions can be evaluated. Depending on the particular algorithm being compiled, there can also be simple hash functions that may lead to inter-stage buffer size minimizations by using bits of the address in different positions than a set of least significant bits as is the case with the hash function used in this work. The use of simple one stage bitwise operations (e.g., XOR, AND) can also be used in hash functions with low delay overheads.

The type of ready protocol used in this paper can be exploited at block level instead of array element level, i.e., the producer could flag the consumer of the availability of a block of data instead of an array element. With that, the consumer stage would not require to test of the availability of each array element. The full advantages of this technique are, however, not trivial as the fine-grain synchronization would be lost.

Stages with non-deterministic producer/consumer rates also impose problems to determine the size of the inter-stage buffers since this would need to perform RTL simulations with worst case scenarios which might be very difficult to achieve. However, this would require an analysis of the algorithms having this kind of behavior.

3 Experimental Setup

The test infrastructure used to evaluate the PSL technique consists of the ***Nau*** compiler, which is based on the Galadriel/Nenya framework [Cardoso, 03], and an RTL simulation environment [Rodrigues, 05b]. As can be seen in Figure 6, the compiler receives as input an algorithm described in a subset of Java bytecodes and generates a description of a specific architecture for the algorithm being compiled. ***Nau*** outputs an RTL structural description of the data-path of the generated architecture and an RTL behavioral description for each data-path's control unit, both using XML (eXtended-Markup Language) dialects. Those XML representations are then translated to the target language by XSLT engines. This is useful because it permits to define a set of XSL translation rules to the chosen language representation (e.g., Verilog, VHDL, etc.).

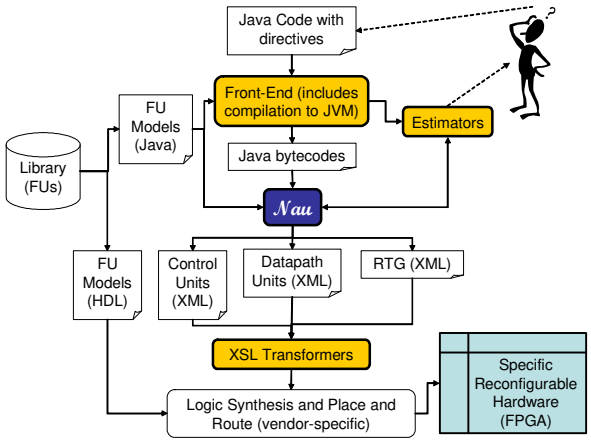


Figure 6: Compilation framework used to evaluate the PSL technique.

The functional test of the architectures generated by ***Nau*** is performed using the Hades [Hendrich, 00] simulation engine. Hades is an event-based simulator that uses behavioral descriptions using the Java language. A library of operators written in Java, reproducing the behavior of each operator existent in the compiler library, used to describe the data-path (e.g., adders, registers, RAMs), is used for the referred RTL simulation [Rodrigues, 05b]. Memory contents and I/O data are stored in files.

One of the advantages of this kind of simulation environment is the possibility to add monitors and behavior to the components of the architecture by including pure Java code. This has been helpful for determining the size of the inter-stage buffers and

to monitor possible communication conflicts as has been explained by the algorithms previously presented. Having access to Java APIs and to object-oriented features, contribute to the use of algorithms and data-structures, hard or impossible to code in hardware description languages, for instance.

Architectures without using the PSL technique are obtained by using directly the compilation flow illustrated in Figure 6. When applying the PSL technique the algorithm is split in its stages (representing the sequences of data-dependent loops) and each stage is then compiled by *Nau* to generate a specific architecture for each stage. The PSL technique is then applied to the set of architectures generated by *Nau*. At the moment, a script is used to change the memories used to communicate data between two stages to the type of inter-stage buffers used by PSL. The script also changes the FSMs of subsequent stages in order they include behavior to test the availability of data in the inter-stage buffers. Although PSL can be included in the compiler itself, evaluations of the technique have been easier to do without changing the compiler.

After the changes in the XML files of the individual data-path and control units, functional simulations are performed using Hades. This is the step where the inter-stage buffers are monitored and their sizes determined by RTL simulation.

Having the size of the inter-stage buffers, the HDL (hardware description language) code is generated (VHDL in this case). This HDL code is then input to commercial logic synthesis and place and route tools. After this final step the architecture using the PSL technique is ready to be tested in the target FPGA board.

4 Experimental Results

To evaluate the impact of the PSL loop pipelining technique on performance and on the size of the inter-stage buffers needed, a number of experiments has been performed. Table 1 shows the benchmarks used, and their characteristics. The benchmarks are all from the image processing domain and have different complexities. They also represent typical image processing stages. The benchmarks are: a fast, 2-D, DCT version (*fdct*); a forward, 2-D, Haar wavelet transform (*fwt2D*); a transformation of an RBG image to pixels in gray with 256 levels plus the histogram calculation on that gray pixels (*RGB2gray + histogram*); and a *smooth* image filter (using a 3×3 window) plus a *sobel* edge detector (*smooth + sobel*). To test the examples a simple board containing a Xilinx Spartan-3 (xc3s400-4) FPGA has been used. Due to the small capabilities of those devices, the examples shown have been prototyped using small data sets in order to have enough on-chip memories and RTL simulation is used to show latencies with realistic image sizes.

Table 2 shows the performance results achieved when comparing the benchmarks with sequential and PSL loop pipelined executions. The #cc column refers to the number of clock cycles needed to execute the sequential implementation of the algorithms, and #cc PSL the number of clock cycles used in the pipelined versions. The speed-ups achieved, which range from 1.29 to 2.02, are shown in the last column. One interesting observation is the fact that the experimental speed-ups almost achieve the theoretical speed-up limit given by equation (2). Also note that in all the examples, memories with 1 clock cycle to store and 2 clock cycles to load array

elements have been used. When using off-chip memories with a larger number of clock cycles to load/store data, the speed-ups achieved will increase since inter-stage buffers are promoted to on-chip and faster memories.

Although out-of-order producer/consumer rates are needed, the *fdct* is the example achieving the largest speed-up due to the almost balanced stages (the example consists of 2 stages).

The *fwt2D* example has 4 stages and the optimal PSL is obtained when applied between the 3rd and the 4th stages. This is because the execution times of stages 2 and 4 are much lower than the ones for stages 1 and 3. When applying PSL between all the stages of the example (i.e., s1, s2, s3 and s4) a speed-up of 1.29 is obtained (image sizes of 512×512 pixels) which is almost equal to the obtained speed-up when PSL is only applied to s2 and s3. Note that the overall achieved speed-up is also limited by the fact that the two stages that effectively take advantage of PSL are stages s2 and s3. The other two are always executing sequentially. The local speed-up obtained with stages s2 and s3 when applying the PSL is 1.8 (image sizes of 512×512 pixels).

Algorithm	**# Stages (sequences of loops or nested loops)**	**#loops in the code**	**Description**
fdct	2 {s1,s2}	3	Fast DCT (Discrete Cosine Transform)
fwt2D	4 {s1,s2,s3,s4}	8	Forward Haar Wavelet
RGB2gray + *histogram*	2 {s1,s2}	2	Transforms an RGB image to a gray image with 256 levels and determines the histogram of the gray image
Smooth + *sobel*, 3 versions: (a) (b) (c)	2 {s1,s2}	6	*Smooth* image operation based on 3×3 windows being the resultant image input to the *sobel* edge detector. Versions: (a): original code; (b): two innermost loops of the smooth algorithm fully unrolled (scalar replacement of the array with coefficients); (c): the same as (b) plus elimination of redundant array references in the original code of *sobel*.

Table 1: Benchmarks used in the experiments.

The *RGB2gray* followed by the *histogram* stage achieved practically the theoretical maximum speed-up (1.75).

The *smooth* followed by the *sobel* edge detector example represents a case where multiple reads to the same array element are performed by the second stage (*sobel*). Most image pixels output by the first stage are read 12 times by the second stage (in the example (a)). Pixels in the borders of the image are read fewer times. An empty/full table of width L equal to 12 is used. When using optimized versions of *sobel* and *smooth*, (b) and (c), a better balance between the two stages is accomplished and speed-ups of 1.80 and 1.92 are achieved, respectively. In those optimized versions, the maximum number of reads to the same array element is 8 instead of 12 and an empty/full table with 8-bit width is used.

Algorithm	Input data size	Stages	#cc w/o PSL	Speed-up Upper – Bound: equation (2)	#cc w/ PSL	Speed-up
fdct	320×240	(s1, s2) (s1) (s2)	628,805 312,003 307,203	2.02	293,015	2.01
	640×480	(s1,s2) (s1) (s2)	2,515,205 1,248,003 1,228,803	2.02	1,171,415	2.02
	800×600	(s1,s2) (s1) (s2)	3,930,005 1,950,003 1,920,003	2.02	1,830,215	2.02
Fwt2D	128×128	(s1,s2,s3,s4) (s1, s2) (s3, s4)	296,457 148,230 148,230	2.00	228,117	1.30
	256×256	(s1,s2,s3,s4) (s1,s2) (s3,s4)	1,182,729 591,366 591,366	2.00	914,965	1.29
	512×512	(s1,s2,s3,s4) (s1,s2) (s3,s4)	4,724,745 2,362,373 2,362,373	2.00	3,664,917	1.29
RGB2gray + *histogram*	640×480	(s1,s2) (s1) (s2)	4,300,816 1,843,209 2,457,609	1.75	2,457,622	1.75
	800×600	(s1,s2) (s1) (s2)	6,720,025 2,880,015 3,840,015	1.75	3,840,007	1.75
Smooth + *sobel* (a)	640×480	(s1,s2) (s1) (s2)	31,720,089 21,044,433 10,597,671	1.51	21,044,449	1.51
	800×600	(s1,s2) (s1) (s2)	49,634,009 32,929,473 16,606,951	1.51	32,929,489	1.51
Smooth + *sobel* (b)	800×600	(s1,s2) (s1) (s2)	30,068,645 13,364,109 16,606,951	1.81	16,640,509	1.81
Smooth + *sobel* (c)	800×600	(s1,s2) (s1) (s2)	25,773,809 13,364,109 11,862,791	1.92	13,364,117	1.92

Table 2: Execution cycles and speed-ups obtained with PSL.

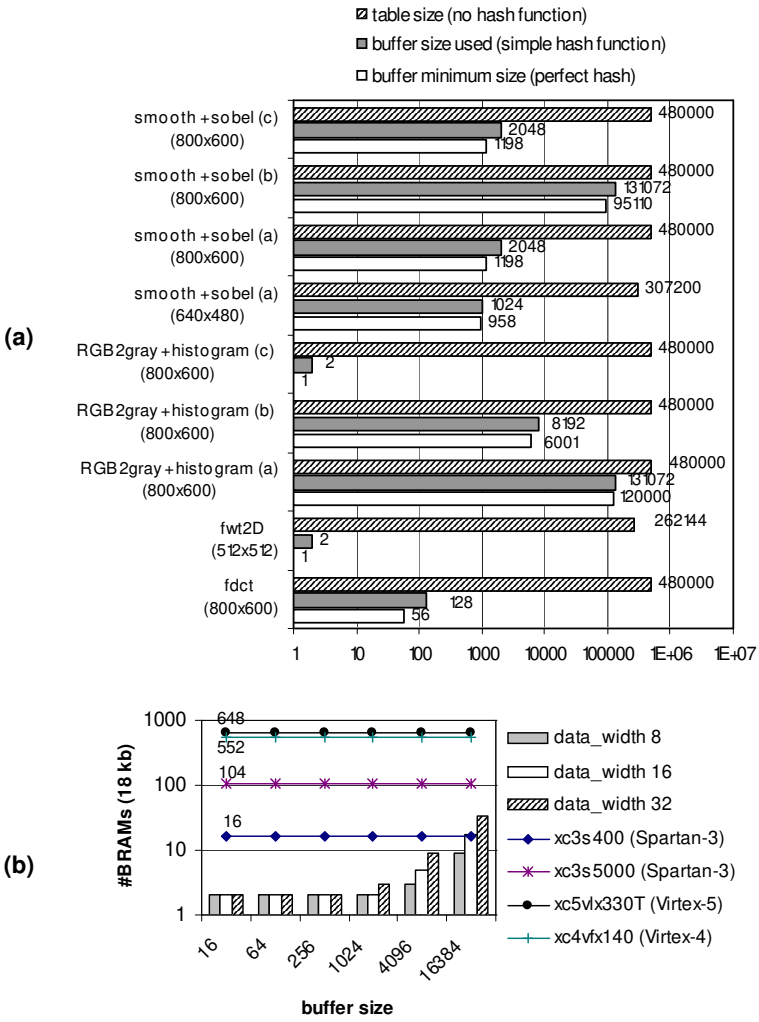


Figure 7: (a) Sizes of the inter-stage buffers for the implemented architectures using the PSL technique. (b) FPGA BRAM resources for different sizes of the inter-stage buffer (including an empty/flag table with 1-bit width).

Figure 7(a) shows the size of the inter-stage buffers for the architectures using the PSL techniques. The sizes for the examples with no hash functions are the same to the needed memory for the examples without using PSL. Note, however, that using PSL dual-port memories and an extra L-bit memory for the table with the flags identifying ready array elements (empty/full table) are needed. Those results show large reductions even when the simple hash-function needs over twice the size of the minimum buffer size if a perfect-hash (correspondent to a full-associative memory) was used (*fdct* case). Also note that the sizes of the buffers for the *fdct* and *fwt2D*

algorithms do not depend on the image input sizes. The hash-based inter-stage buffers used consider memories with sizes proportional to power of two values.

With respect to *RGB2gray* plus *histogram* example (a), the size of the inter-stage is always 25% of the size of the images. This is due to the fact that the values are produced at a higher rate than they are consumed. When a slowdown of the producer stage (*RGB2gray*) is performed by adding one (example (b)) and two (example (c)) states in the FSM responsible to the first stage, large reductions are achieved. A size of two is required for the inter-stage buffer when adding two clock cycles of slowdown for each loop iteration of the *RGB2gray* stage. Also important is the fact that the slowdown of the first stage has not restrained the PSL to achieve a latency similar to the latency presented in Table 2 (this one achieved without using the slowdown of the first stage).

The inter-stage buffer sizes for the *smooth* plus *sobel* example (a) have been achieved using the correct number of multiple reads to each array element. Note that the use of the maximum number of reads to the same array element, which is 12, would need a 5,568 minimum size perfect-hash buffer, corresponding to a buffer with the same size of the images used (800×600 pixels). This is due to the fact that a number of image borders are maintained in the inter-stage buffer, because their status never reaches the empty state. They are initialized with a word identifying 12 reads but there will never be 12 reads. When taking into account the correct number of reads per array element, an inter-stage buffer which size is presented in Figure 7 (*smooth*+*sobel*) leading to a 256× reduction is accomplished. The example *smooth* plus *sobel* optimized (c) requires a maximum of 8 reads to each array element (instead of the 12 reads for the versions (a) and (b)). This optimized version needs the same previously referred size for the inter-stage buffer.

Figure 7(b) shows the number of BRAMs (Block RAMs), each one with 18 Kb, needed for different sizes of the inter-stage buffer (implementing the empty/full tagged memory buffer with hash) in a number of Xilinx FPGAs, considering buffer data widths of 8, 16 and 32 bits. It is also shown the maximum number of BRAMs available in the considered Xilinx FPGAs. This gives an idea about the size of the inter-stage buffers possible to use with the actual FPGA devices. It can be seen that those devices have sufficient on-chip memory resources available to enable the inter-stage buffer scheme used in this paper.

Examples where the same ordering of producer/consumer is needed are the ones with potentially higher performance improvements, especially when the values are produced and consumed from the very beginning of the execution of the stages and producer/consumer operations are performed on continuous loop iterations. Although this may occur in some algorithms (some image kernels include this kind of behavior), most examples have different behavior. The technique presented in this paper is not constrained by the former kind of behavior and can also be applied to reduce both execution time and storage requirements.

Figure 8 shows the impact on resource usage when applying PSL. It can be seen that the number of 4-LUT, FF, and Slices used is almost the same. In some cases PSL achieved resource savings due to the elimination of the multiplexers to access the memories shared between two stages in the reference designs. Those memories are transformed to the dual-port inter-stage buffers when using PSL. The clock frequencies are also similar (Figure 8 shows the differences between the obtained

clock frequencies having as a reference each example without PSL). This was expected, because the PSL examples do not add hardware complexity. Note that designs with PSL also need on-chip memories (see in Figure 7(b) the BRAMs needed for different sizes of the inter-stage buffers).

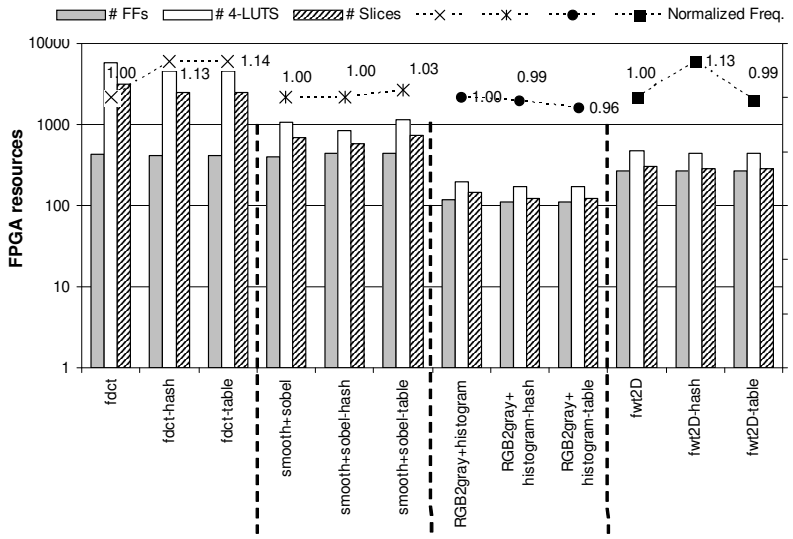


Figure 8: FPGA resources needed reported after P&R.

5 Related Work

Due to the impact in performance, loop pipelining has been focus of intense research efforts for many years [Allan, 95]. It has been considered for both microprocessors and application specific architectures. With respect to compilation for FPGAs, the work in [Weinhardt, 96] has been one of the first efforts considering pipelining of innermost, well-behaved, loops. Other examples of efforts on loop pipelining to FPGA-based systems are presented in [Maruyama, 00], [Diniz, 00], [Callahan, 00], [Haldar, 01], [Gokhale, 00], and [Snider, 02]. Beside the efforts on generating the pipelining structures for efficient execution of loops in FPGAs, authors have used some loop transformations (e.g., unrolling, interchange, tiling, etc.) to make, e.g., the most promising loops innermost loops, since only them are pipelined by most compilers [Weinhardt, 01]. Other loop pipelining schemes try to overlap subsequent iterations of an outer loop with an inner loop, as is shown by the example presented in [Bondalapati, 01].

In [Cardoso, 05], a dynamically loop pipelining scheme applied to a coarse-grained reconfigurable, data-driven, architecture is presented. It takes advantage of a ready/acknowledge protocol between operations to achieve loop pipelining of innermost loops. An example is also given showing loop pipelining of two nested loops. Although this scheme can be implemented in FPGAs, there is no strong

evidence of its efficiency using the fine-grained resources and synchronous model of the current commercial FPGAs.

Most relevant to our approach is the work presented in [Ziegler, 02], [Ziegler, 03], and [Ziegler, 06], which is a coarse-grained synchronization approach to overlap some of the execution steps of sequences of loops or functions (i.e., functions or loops waiting for data start computing as soon as the required data items are produced in a previous function or by a certain iteration of a previous loop). They communicate data to subsequent stages in the code using a FIFO mechanism. Each FIFO stage stores an array element or a set of array elements (e.g., a raw of pixels in an image). Array elements in each FIFO stage can be consumed by different order they have been produced. Examples with the same order of producer/consumer only need FIFO stages with one element. In the other case, each stage must store a sufficient number of array elements in order that all of them are consumed (by any order) before the next FIFO stage is considered. This is the major bottleneck of the technique, since a FIFO stage may need to store the entire image and in that case no loop pipelining is achieved, and is a consequence of using coarse-grain (the grain is related to the size of the FIFO stages) instead of fine-grain synchronization. Note also that the coarse-grain synchronization limits the applicability of the technique. In [Ziegler, 06] they present analysis schemes to determine the communication needed by this type of pipelining. To the best of our knowledge, the work of [Ziegler, 06] has been in the context of design space exploration and concerning speed-ups only one example is referred (with a speed-up of 1.76). Regarding their work, the work presented in this paper can be thought as a more generic approach, without having their major constraints.

Orthogonally to the work presented in this paper, empty/full tagged memories have been presented in [Smith, 81] and used more recently in machines such as the MIT Alewife [Agarwal, 95], both in the context of shared memory multi-processor architectures. Those memories are used for fine-grain synchronization of data produced/consumed by tasks executing in distinct microprocessors and can be seen as the empty/full tables used in the work presented in this paper. Note, also, that the optimized solution using the inter-stage buffers with hash-functions has not been, as far as we know, used before.

6 Conclusions

This paper presents a technique for pipelining sequences of data-dependent loops using a fine-grain synchronization scheme. With the proposed technique, before the end of a stage (typically a loop or a set of nested loops), a subsequent stage is able to start execution as long as the array elements required have been produced by the previous stage. The scheme is implemented using inter-stage buffers to communicate both the produced-consumed data between stages and the ready flags. It also uses concurrent units to control each stage with synchronization being achieved by the ready flags. Doing this, the approach presented in this paper is able to deal with irregular (out-of-order) produced-consumed array elements in a natural way.

Although different inter-stage buffer schemes are proposed, a simple hash function, without needing additional hardware resources and without degrading the maximum clock frequency achieved, is shown to be an efficient implementation. It

permitted to substantially reduce the size of the buffers for the examples used in the experiments.

The technique is able to both improve performance and to reduce the memory requirements related to the data communication between sequences of loops. The experiments conducted so far, give strong evidences that the technique is able to achieve performance improvements close to the theoretical limit.

It is also shown how the technique can be applied in the context of a compiler of imperative software programming languages (a subset of Java is used) to specific architectures suitable for implementation in FPGAs.

Ongoing work focuses on static analysis techniques to decide about the use of the loop pipelining scheme and to estimate the size of the inter-stage buffers needed to communicate array elements between the set of loops being pipelined. Future work will also include studies about the use of loop and data layout transformations that may lead to further reductions of the sizes of the inter-stage buffers. Another interesting path would be to evaluate the technique with stages having conditional structures that may impose non-deterministic behavior.

Acknowledgements

This work is partially supported by the Portuguese Foundation for Science and Technology (FCT) - FEDER and POSI programs - under the CHIADO project (POSI/CHS/48018/2002). The authors would like to thank the fruitful discussions with Pedro Diniz regarding aspects of the technique presented in this paper. We also thank the suggestions pointed out by the anonymous reviewers.

References

[Agarwal, 95] Agarwal A., Bianchini R., Chaiken D., Johnson K.L., Kranz D., Kubiatowicz J., Lim B.-H., Mackenzie K., and Yeung D., "The MIT Alewife Machine: Architecture and Performance," Proceedings 22nd Annual International Symposium on Computer Architecture (ISCA'95), 22-24 June,1995: pp. 2-13.

[Allan, 95] Allan V.H., Jones R.B., Lee R.M., Allan S.J., "Software Pipelining", ACM Computing Surveys, (Vol. 27, Issue 3), Sept. 1995: pp. 367-432.

[Bondalapati, 01] Bondalapati K., "Parallelizing of DSP Nested Loops on Reconfigurable Architectures using Data Context Switching", Proceedings of IEEE/ACM 38th Design Automation Conference (DAC'01), Las Vegas, Nevada, USA, June 18-22, 2001: pp. 273-276.

[Callahan, 00] Callahan T. J., Wawrzynek, J. "Adapting Software Pipelining for Reconfigurable Computing", Proc. of the Int'l Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'00), San Jose, CA, USA: ACM Press, Nov. 17-19, 2000: pp. 57-64.

[Cardoso, 03] Cardoso J. M. P., and Neto H.C., "Compilation for FPGA-Based Reconfigurable Hardware," IEEE Design & Test of Computers Magazine: March/April, 2003, vol. 20, no.2: pp. 65-75.

[Cardoso, 05] Cardoso J.M.P., "Dynamic Loop Pipelining in Data-Driven Architectures," in Proc. of the ACM International Conference on Computing Frontiers (CF'05), Ischia, Italy, 4-6 May 2005, ACM Press: pp. 106-115.

[Diniz, 00] Diniz P., Park J., "Automatic Synthesis of Data Storage and Control Structures for FPGA-based Computing Engines", Proc. of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00), IEEE CS Press, 2000: pp. 91-100.

[Gokhale, 05] Gokhale M. and Graham P. S., Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays, Springer, 2005, ISBN: 0387261052.

[Gokhale, 00] M. Gokhale M.., Stone J.M.., Gomersall E., "Co-synthesis to a hybrid RISC/FPGA architecture", Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology, (Vol. 24, No. 2), March 2000: pp. 165-180.

[Haldar, 01] M. Haldar M., Nayak A.., Choudhary A., Banerjee P., "A System for Synthesizing Optimized FPGA Hardware from Matlab", Proc. of IEEE/ACM Int'l Conference on Computer Aided Design (ICCAD'01), San Jose, CA, USA, Nov. 4-8, 2001: pp. 314-319.

[Hendrich, 00] N. Hendrich N., "A Java-based Framework for Simulation and Teaching", Proc. of the 3rd European Workshop on Microelectronics Education (EWME'00), Aix en Provence, France: Kluwer Academic Publishers, 18-19, May 2000: pp. 285-288.

[Knuth, 73] Knuth D., The Art of Computer Programming, Volume 3: Sorting and Searching, Addison-Wesley, 1973.

[Maruyama, 00] Maruyama T., Hoshino T., "A C to HDL compiler for pipeline processing on FPGAs", Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00), IEEE CS Press, 2000: pp. 101-110.

[Rodrigues, 05a] Rodrigues R., Cardoso J. M. P., "Pipelining Sequences of Loops: A First Example," in International Workshop on Applied Reconfigurable Computing (ARC2005), held in conjunction with IADIS International Conference Applied Computing 2005, Algarve 22-23, Portugal, pp. 147-151.

[Rodrigues, 05b] Rodrigues R., Cardoso J. M. P., "An Infrastructure to Functionally Test Designs Generated by Compilers Targeting FPGAs," Interactive Presentation at the Design, Automation and Test in Europe Conference (DATE'05), Munich, Germany, March 7-11, 2005, IEEE Computer Society Press, pp. 30-31.

[Smith, 81] Smith B., "Architecture and Applications of the HEP Multiprocessor Computer System," Society of Photo-optical Instrumentation Engineers, 298, 1981: pp. 241–248.

[Snider, 02] Snider G., "Performance-constrained pipelining of software loops onto reconfigurable hardware", Proc. of ACM 10th Int'l Symposium on Field-Programmable Gate Arrays (FPGA'02), New York, USA: ACM Press, 2002: pp. 177-186.

[Weinhardt, 01] Weinhardt M., Luk W., "Pipeline Vectorization", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, (Vol. 20, no. 2), Feb. 2001: pp. 234-233.

[Weinhardt, 96] M. Weinhardt M., "Portable Pipeline Synthesis for FCCMs", Proc. 6th Int'l Workshop on Field-Programmable Logic and Applications (FPL'96), Darmstadt, Germany: LNCS 1142, Springer-Verlag, Sept. 1996: pp. 1-13.

[Ziegler, 02] Ziegler H., So B., Hall M., P. Diniz, "A Parallelizing Compiler Approach for Coarse-Grain Pipelining on Multiple FPGA Architectures", Proc. of 10th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02), Napa, CA, USA, 2002: pp. 77-86.

[Ziegler, 03] Ziegler H., Hall M., Diniz P., "Compiler-Generated Communication for Pipelined FPGA Applications", Proceedings of the 40th Design Automation Conference (DAC'03): pp. 610-615.

[Ziegler, 06] Ziegler H., *Compiler-directed design space exploration for pipelined field-programmable gate array applications,* PhD thesis, University of Southern California, California, USA, May 2006.