Building an Application-specific Memory Hierarchy on FPGAs

Harald Devos*, Jan Van Campenhout, and Dirk Stroobandt

Parallel Information Systems, ELIS-Dept., Ghent University,
Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium
Harald.Devos@elis.ugent.be

Abstract. The high potential performance of FPGAs cannot be exploited if a design suffers a memory bottleneck. Therefore, a memory hierarchy is needed to reuse data in on-chip memories and minimize the number of accesses to off-chip memory. Buffer memories not only hide the external memory latency, but can also be used to remap data and augment the on-chip bandwidth through parallel access of multiple buffers. This paper presents a step-by-step methodology to construct such a memory hierarchy. Special care is taken of the reusability of design modules and the optimization of address expressions to improve the performance.

1 Introduction

FPGAs (Field Programmable Gate Arrays) offer a high computational power thanks to the massive parallelism available and the huge on-chip bandwidth. However, the total on-chip memory size is fairly small. Usually an off-chip memory is needed. The bandwidth to this external memory may become a bottleneck. Typically, this memory is made in a technology, e.g., SDRAM, with an indeterministic latency and with a low bandwidth if transfers are not done in burst mode. This indeterministic latency may also be caused by the fact that the main memory is shared with other cores on the FPGA.

To reduce the bandwidth requirements a memory hierarchy is needed. If frequently used data is stored in on-chip buffers, the number of off-chip memory accesses can be reduced and grouped into bursts. Using multiple parallel accessible memory banks (Fig. 1(a)), or expensive memory technologies (SRAM, Z-RAM) increases the available off-chip bandwidth. However, to use the external bandwidth more efficiently a memory hierarchy should still be used and the memory access pattern should be optimized to use this hierarchy (Fig. 1(b)).

In processor-based systems a memory hierarchy consists of caches or scratch pad memories. This is a fixed memory hierarchy and the application code has to be optimized to optimally use this given memory structure, e.g., by doing loop transformations to increase the data locality, i.e. to bring accesses to the same or neighboring data elements closer together in the time. On an FPGA

* Harald Devos is supported by I.W.T. grant 060068

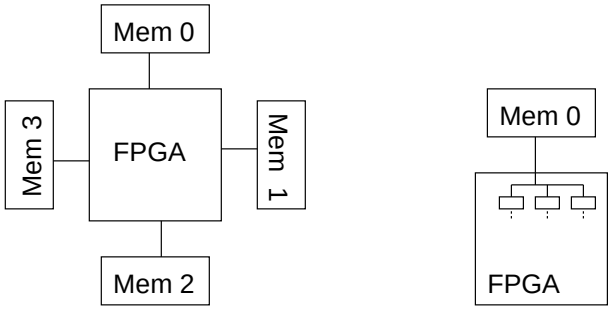


Fig. 1. Multiple memory banks (a). Single memory bank with on-chip memory buffers (b).

the designer has to construct the memory hierarchy using the available memory blocks. This offers the freedom to build an application specific memory hierarchy, i.e. to adjust the memory system to the application and not only the application to the memory hierarchy. In this paper, some aspects of this problem will be studied starting from a hardware design point.

This paper does not focus on ways to improve the locality of data accesses or map data to memories. For this we refer to related work (Sect. 2). Instead, we focus on hardware implementation aspects of building a memory hierarchy and the impact of choices made by the way data is mapped onto buffers.

We present the following contributions:

- To increase the reusability, a modular architecture is used. The computation core is shielded from the IO (Input-Output) infrastructure to make adjustments to different platforms more feasible. Hash functions will be used to localize (i.e. make easier to adapt) the description of the memory mapping. (Sects. 3 and 5)
- A step-by-step methodology is given to insert a memory hierarchy into a system. (Sect. 5)
- Thanks to the modular architecture the complexity of address expressions can be adapted. For cycle-by-cycle accesses to on-chip memories, simple, i.e. fast and cheap, expressions will be used. For block transfers to off-chip memory more complex expressions are allowed. (Sect. 4)

As a case study an implementation of an Inverse Discrete Wavelet Transform (IDWT) will be extended with a memory hierarchy and integrated in a video decoder on FPGA (Sect. 6).

2 Related Work

The focus of this paper is the system integration of a computation core on an FPGA. To benefit from the construction of a memory hierarchy, complementary techniques found in related work have to be applied.

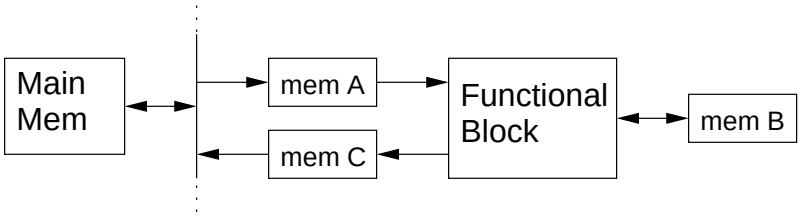


Fig. 2. General memory hierarchy. Several of the memories may coincide or consist of multiple memories.

A good data locality is needed to reuse data stored on-chip as much as possible. Loop transformations can be used to improve this locality [1, 2].

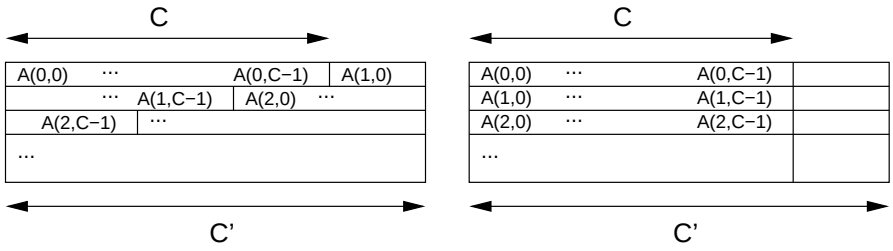
Since burst mode offers a higher memory bandwidth, off-chip memory accesses should be grouped into bursts or blocks. The term prefetching [3] is used to denote the fact that a data transfer from an external memory to a buffer is started before the execution reaches a point where that data is needed. This contrasts with a classical cache where data is only fetched after a cache miss.

A side effect of loop transformations is that address expressions may become complex. Therefore, address optimization techniques have been developed. Many exploit the repetitive evaluation of these expressions in a loop and use differences of the terms of an expression to calculate the next value. Sheldon et al. [4] present techniques to eliminate division and modulo operations, by inserting conditionals and using algebraic axioms and loop transformations. Most techniques optimize the evaluation of a given set of address expressions, possibly sharing logic among different address expressions [5]. Only a few remap data to simplify the address expressions [6, 7]. Most methods are useful for both software and hardware implementations. [8] focuses on address optimization for FPGAs.

An extensive overview of memory optimization techniques is found in [6].

3 Memory Hierarchy Template

Figure 2 shows the general memory hierarchy model that will be used here. A computation core is connected with an external memory (Main Mem) via one or multiple buffer memories. Intermediate results are stored in on-chip memories (B), e.g., registers, RAM blocks or FIFOs, or when not enough on-chip memory is available in the external memory, passing the I/O buffers (A and C). Also the input data and final output have to pass these buffers. The usage of dual-port memories allows to transfer data between the main memory and the buffers in parallel with the operation of the functional unit, and to use different clock domains for both tasks. Here only 3 buffer memories are shown. More separate buffers create more on-chip bandwidth. The off-chip memory is considered to be made in a technology, e.g., SDRAM, with an indeterministic latency and with a low bandwidth if transfers are not done in burst mode. This indeterministic



(a) Address $(A(i,j)) = i \times C + j$ (b) Address $(A(i,j)) = i \times C' + j$

Fig. 3. By inserting free space into the memory, address expressions can be simplified.

latency may also be caused by the fact that the main memory is shared with other cores on the FPGA. Synchronization points are needed to ensure that read (write) operations are delayed if data is not fetched (stored) from (to) the main memory yet.

I/O modules drive the transactions between main memory and buffers. They take care of address translations, similar to the *direct* and *strided mapping* supported by the Impulse memory controller [7].

To benefit from a memory hierarchy, the reuse of data in on-chip buffers should be maximized and transferring data to and from off-chip memory should be minimized and done in bursts as much as possible. Loop transformations can be used to improve the data locality. Multiple memory banks can be used if needed, but then with a memory hierarchy for each of them.

4 Address Complexity

Accesses to on-chip memory have to be fast since they limit the speed of the functional unit. Therefore, the address expressions for these accesses have to be kept simple. Address calculations for off-chip memory accesses can be spread over several clock cycles, since only one evaluation is needed for an entire burst transfer. By sacrificing on-chip memory, the addresses can be simplified.

Consider for example an image processing system that reads a 2-dimensional array A. The typical way to store this array leads to address expressions with a multiplication:

$$\text{Address}\,(A(i,j)) = BA + i \times C + j \ ,$$

with BA the base address and C the number of columns. If C is not a power of 2 or C is a parameter, which only receives a value during execution, this is an expensive operation. After copying data to on-chip memory the base address can be removed (Fig. 3(a)). A straight-forward simplification is to align all lines to a multiple of a power of 2. This is shown in Fig. 3(b), where C' is the smallest power of 2 for which $C' \geq C$, if C is known, or $C' \geq C_{\max}$ if C is a parameter with $C_{\max}$ the maximal possible value.

Only a few lines can be stored in the buffer. If this number of lines R' is also set to a power of 2 and the mapping of lines of the image to lines of the buffer

is done in a circular way the address becomes

$$\begin{aligned} \text{Address}\,(A(i,j)) &= (i \bmod R') \times C' + j \\ &= i(r-1 \text{ downto } 0)\ \&\ j\ , \quad \text{with } R' = 2^r\ . \end{aligned} \tag{1}$$

As a result, only the least significant bits of i have to be generated. Note that the addition is in fact only a concatenation, denoted with "&".

Since RAM blocks on FPGAs have sizes that are a power of two, in many cases no extra memory blocks are needed to apply the techniques described in this section. Even when the off-chip memory can be accessed in one clock cycle just like the on-chip RAM blocks, one can take benefit from on-chip buffers, since simpler address expressions can be used, which may lead to higher clocking frequencies. Another benefit is that when a buffer is composed of multiple RAM blocks mutliple elements, one for each RAM, can be accessed in parallel.

5 A Step-by-step Approach

Many high-level synthesis tools generate one memory for each array in the input code. Also when building a design manually, it is easier to start the design process with multiple memories and only construct a memory hierarchy with one main memory later on. Therefore, we present a step-by-step design flow to transform a system with multiple memory banks, similar to the systems in Fig. 1(a) and 4(a), to a system with one external memory and on-chip buffers, similar to the systems in Fig. 1(b) and 4(c). Here, an overview of the flow is given. A detailed elaboration is found in the case study in Sect. 6.

- On-chip memories are added to contain intermediate data sets that are small enough to fit in them. If results produced by one operation are consumed by another operation in the same order, FIFO buffers can be used
- Buffers are inserted between each external memory and the functional unit. The size is kept as large as the external memory itself so that no remapping of data and changes in address expressions are needed. A single copy transaction (for each memory/buffer pair) of all data to the buffer at the start of execution and a single transfer to the *external* memory at the end suffices for correct behavior. Separate I/O modules take care of these copy transactions.
- The two large copy operations are split into smaller prefetch and store operations, such that at each moment only a small amount of data in the buffers is alive (= transferred and still needed). Synchronization between the data transfers and the operation of the functional unit is needed to ensure correct behavior.
- The data in the buffers is remapped such that the buffers can be resized and fit in on-chip memories. A *hash function* (cf. caches) is used to translate the indices of the arrays into the new address expressions.
- The external memories are merged to form one main memory. Base addresses are added to the addresses used in the prefetch and store transfers. The I/O

modules are all connected to the same memory. Arbitration between the transfers is needed to avoid conflicts.

By doing the transformations in small steps, errors can be detected easier and faster, since simulation is possible at any time. To increase the reusability, a modular architecture is used. When transferring the design to another platform only the I/O modules have to be adapted. By using hash functions instead of simply adapting the address expressions, the data mappings can be changed in an easier way, e.g., when a device upgrade offers the option to use more on-chip memory. This does not result in an area overhead since bits not used by the hash function will be optimized away by the synthesis tools (cf. (1)).

6 Case Study: System Integration of an IDWT

The 2D Discrete Wavelet Transform (DWT) and its inverse (IDWT) are commonly used in image processing and compression applications, e.g, JPEG-2000. The design without memory hierarchy we will start from is generated with CLooGVHDL as described in [9]. The design flow is shown in Fig. 5. A software implementation in C is split into statement definitions and loop control structure. Loop transformations are applied on the latter to improve the spatial locality, resulting in a so-called line-based variant. With CLooGVHDL a loop control entity is generated. The statement definitions are translated to a VHDL syntax using VIM-scripts [10].[1] Array accesses are translated to memory accesses with a one-cycle access time. The result is a list of execution steps for each statement. On this, scheduling optimizations are done. The *Steps2process* tool generates a finite state machine to execute the statements based on the schedule specifications. The architecture of the generated design is shown in Fig. 4(a).

The fact that software code equivalent to the hardware is available can be exploited for the construction of a memory hierarchy as shown below.

6.1 Adding New Hardware Structures

The design tools mentioned above will be reused for the extension of the architecture. New hardware constructs are inserted one after another, iterating over the following steps:

First, C preprocessor macros that simulate the behavior of the new construct are written. For example, a *push* and a *pop* macro to write to and read from a FIFO buffer. In C this is simulated by accessing an array and incrementing a counter. A VHDL block that corresponds with the new construct is written or instantiated. Procedures or functions that correspond to the functionality of the C macros are written. For a FIFO this is the instantiation of FIFO entities and writing the VHDL procedures *push_fifo* and *pop_fifo* that access such a

[1] This is only a temporary solution. Plans are to integrate a C parser. This is only an implementation issue and does not influence the methodology.

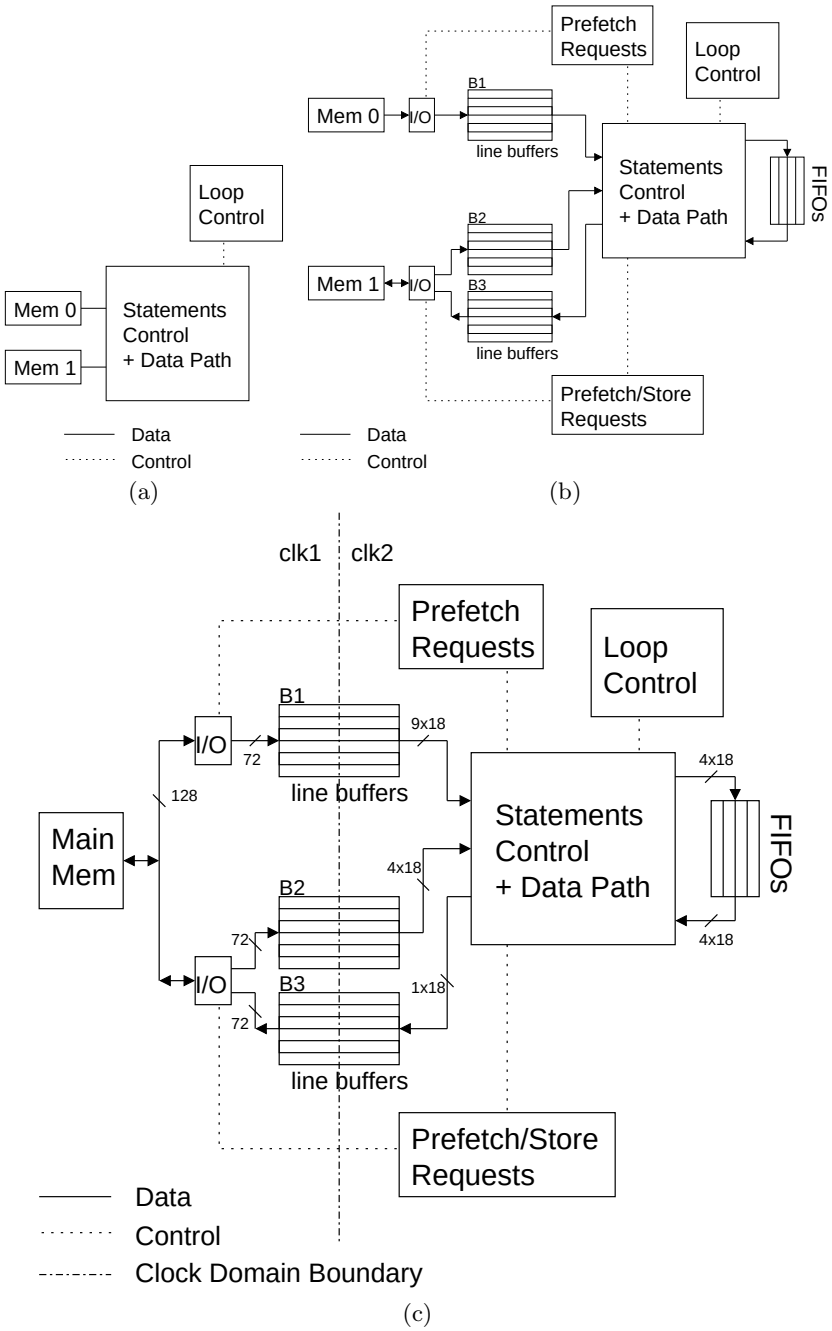


Fig. 4. Line-Based IDWT without memory hierarchy (a), after adding buffers (b), and after full system integration (c).

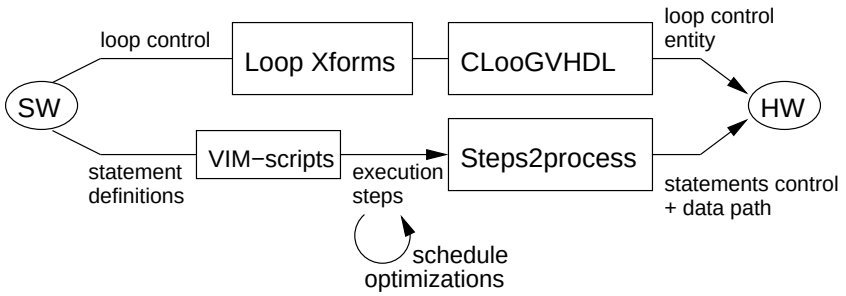


Fig. 5. Tool flow used to generate the IDWT implementation.

structure. Next, the VIM-scripts are extended to replace the C macros with the corresponding VHDL procedure or function calls.

After the equivalence of the C and VHDL constructs and the translation is tested, the new C macro is used in all C code, where desired. Finally, the generated VHDL is tested and the impact of the new hardware structure may be examined after synthesis. Where possible optimizations are done. For example, removing unused data structures or doing schedule optimizations. After the introduction of the FIFOs more data elements can be accessed in parallel, which allows to shorten the schedules. These optimizations can be included in the scripts.

This approach is similar to the way many high-level synthesis tools work. They extend C with macros that correspond to predefined hardware structures. Here, the difference is that new macros, not predefined by such tools, can be added and used with little effort.

6.2 Inserting Buffers

The extension of the design from Fig. 4(a) to (b) is done step-by-step. For each new type of construct a flow is used as described in Sect. 6.1.

Four FIFO buffers are inserted to transfer data from the vertical to the horizontal wavelet transformation. This halves the accesses to the *main* memories Mem 0 and Mem 1. Line buffers (B1, B2 and B3) in which several lines of an image can be stored are inserted between the main memories and computation elements. A block transfer system copies data from the main memories to and from the buffers. A queue of prefetch and store requests is kept in the Prefetch(/Store) Requests entities. A new fetch request is added each time space becomes available in the buffers and not just before the data is needed. Therefore, if the system is not bandwidth limited, only in the beginning time is wasted waiting for data. A block transfer is specified by the source and target address and the amount of data to be copied. Synchronization points are used to ensure that when a line in one of the buffers is accessed, all transfers between that line and the main memory are finished. Some line buffers are split into parallel accessible buffers of one line to increase the on-chip bandwidth. This results in a large reduction in the

Table 1. Synthesis results of the IDWT with and without memory hierarchy. CIF resolution = 288 × 352 pixels. Results obtained with Altera QuartusII v6.1 for the Altera Stratix EP1S25F1020C5 (S25C5) and EP1S60F1020C6 (S60C6, lower speed grade). The number of cycles and frame rate assume a one cycle ($1/f_{\max}$) access time to the main memories. With an SDRAM, this would be the performance when a cache running at $f_{\max}$ is used and no misses occur (i.e. the latency to the SDRAM is hidden, e.g., by prefetching).

Mem-hierarch.	LE	Buf Mem (bit)	DSP bl. (#Mul)	Cycles (72×88)	$f_{\max}$ (MHz) S25C5	$f_{\max}$ (MHz) S60C6	Frames/s (CIF) S25C5	Frames/s (CIF) S60C6
Fig. 4(a)	10836	0	18 (9)	161037	50.12	43.40	19.71	17.07
Fig. 4(b)	17350	297504	18 (9)	59240	47.22	45.60	50.50	48.77

number of clock cycles as shown in Table 1. For the Stratix S60 the clock speed is increased after adding the memory hierarchy, thanks to the simplification of the address expressions. The clock speed on the S25 is lowered due to congestion in the FPGA routing (area usage of almost 70%).

6.3 Further Integration

Further integration work is needed to put the design on an Altera PCI Development Board with a Stratix EP1S60F1020C6 FPGA and 256 MiB of DDR SDRAM memory (from Fig. 4(b) to (c)).

The content of the two main memories is mapped onto the single DDR SDRAM memory. The Avalon switch fabric [11] is used to connect the DDR core (memory controller) with the I/O blocks. The latter take care of the conversion of local addresses, used within the IDWT, to addresses in the *global* memory space. Since the 18 bit word width, used until now, does not correspond to the 128 bit data ports of the DDR controller, the word size at the left side of the line buffers is set to $4 \times 18 = 72$ bit and converted to and from $4 \times 32 = 128$ bit using sign extension and truncation.

The Avalon fabric only supports burst transfers that are a multiple of 16 B (128 bit) long and start at an address that is a multiple of 16 B. Therefore, the lines in all wavelet subbands are aligned to a multiple of 128 bit by letting each row start at a multiple of 512 pixels (1 pixel = 4B). This inserts more unused space than strictly needed, but memory space was not a problem in the DDR-memory and it simplifies address calculations, similar to the example in Sect. 4. A DMA controller (Direct Memory Access) is used to drive the burst transfers [11].

To allow the memory controller and the wavelet transform to run at their maximal frequency, different clock domains are introduced. The dual-port memories offer a safe clock domain crossing for the data. For the control signals extra registers are inserted (brute-force synchronization).

Finally, other blocks are connected to the switch fabric to build the RESUME scalable wavelet-based video decoder described in [12]. It can decode

26.15 frames/s (clocking the DDR at 65 MHz, limited by the FPGA synthesis proces). The IDWT on its own, clocked at 54 MHz (reached with other tool settings than for Table 1), can transform 53 frames/s.

7 Conclusions

A system using multiple memories with short access times can be transformed step-by-step into a system with a memory hierarchy connected to an external memory with less predictable access times. Using a modular design description increases the reusability. With a good choice of data mapping in the buffers, addresses can be simplified to optimize the performance, possibly at the cost of a higher memory usage. This has been demonstrated with the system integration of an IDWT.

References

1. Wolf, M.E., Lam, M.S.: A data locality optimizing algorithm. In: ACM SIGPLAN Programming Language Design and Implementation (PLDI). (1991) 30–44
2. McKinley, K.S., Carr, S., Tseng, C.W.: Improving data locality with loop transformations. ACM Trans. Program. Lang. Syst. **18**(4) (1996) 424–453
3. Vanderwiel, S.P., Lilja, D.J.: Data prefetch mechanisms. ACM Comput. Surv. **32**(2) (2000) 174–199
4. Sheldon, J., Lee, W., Greenwald, B., Amarasinghe, S.: Strength reduction of integer division and modulo operations. In: Languages and Compilers for Parallel Computing (LCPC). Volume 2624 of LNCS., Cumberland Falls (2001) 1–14
5. Miranda, M.A., Catthoor, F.V., Janssen, M., De Man, H.J.: High-level address optimization and synthesis techniques for data-transfer-intensive applications. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **6**(4) (1998) 677–686
6. Panda, P.R., Catthoor, F., Dutt, N.D., Danckaert, K., Brockmeyer, E., Kulkarni, C., Vandercappelle, A., Kjeldsberg, P.G.: Data and memory optimization techniques for embedded systems. ACM Transact. Des. Automat. Electron. Syst. **6**(2) (2001) 149–206
7. Zhang, L., Fang, Z., Parker, M., Mathew, Binu K.and Schaelicke, L., Carter, John B.and Hsieh, W.C., McKee, S.A.: The Impulse memory controller. IEEE Transactions on Computers **50**(11) (2001) 1117–1132
8. Zissulescu, C., Kienhuis, B., Deprettere, E.: Expression synthesis in process networks generated by LAURA. In: 16th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP). (2005) 15–21
9. Devos, H., Beyls, K., Christiaens, M., Van Campenhout, J., D'Hollander, E.H., Stroobandt, D.: Finding and applying loop transformations for generating optimized FPGA implementations. Trans. on HiPEAC I, LNCS **4050** (2007) 159–178
10. Oualline, S.: Vi IMproved (VIM). New Riders (2001)
11. Eeckhaut, H., Christiaens, M., Stroobandt, D.: Improving external memory access for Avalon systems on programmable chips. In: FPL'07, 17th International Conference on Field Programmable Logic and Applications. (2007)
12. Eeckhaut, H., Devos, H., Faes, P., Christiaens, M., Stroobandt, D.: FPGA design methodology for a wavelet-based scalable video decoder. In: Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS). Volume 4599 of LNCS. (2007) 169–178