Exploiting dynamic reconfiguration of platform FPGAs: Implementation issues

Miguel L. Silva[1] and João Canas Ferreira[1,2]

[1]FEUP/DEEC
Rua Dr. Roberto Frias, s/n
4200-465 PORTO, Portugal
mlms@fe.up.pt

[2]INESC Porto
Rua Dr. Roberto Frias, s/n
4200-465 PORTO, Portugal
jcf@fe.up.pt

Abstract

The effective use of dynamic reconfiguration requires the designer to address many implementation issues. The market introduction of feature-full platform FPGAs equipped with embedded CPU blocks expands the number of situations where dynamic reconfiguration may be applied to improve overall performance and logic utilization. The paper compares the design of two similar systems supporting dynamic reconfiguration and the issues that were addressed in their implementation. The first system supports 32-bit data transfers between CPU and the dynamically reconfigurable circuits. The other implementation supports 64-bit transfers, but its effective use is more complicated and several restrictions must be taken into account. The work includes a performance comparison of the two designs on several simple tasks, including pattern matching, image processing and hashing.

1 Introduction

The present work is concerned with the implementation issues that arise for designs that try to exploit some platform FPGA's capability for configuration changes at run-time. The intent is typically to time-share the available hardware to support multiple (and mutually exclusive) tasks; alternatively, the designer may be seeking better performance by adapting the hardware implementation to the actual data being processed at a given instant.

Work partially funded by the Department of Electrical and Computer Engineering of the Faculty of Engineering of the University of Porto, under contract DEEC-ID/05/2003, and by FCT scholarship SFRH/BD/17029/2004.

In the context of the present work, we are interested in designs that contain a closely-coupled CPU, typically as a dedicated block or as a core implemented on part of the reconfigurable fabric. There are many hardware and software issues that must be considered in this context. The present work deals mainly with the need to establish an appropriate hardware environment in order to be able to carry out dynamic reconfigurations in an orderly fashion, and how the associated design choices affect the global performance. Since the actual details of the underlying reconfigurable platform may be important for the concrete analysis of the issues, the paper presents its discussion in the context of systems based on Virtex-II Pro devices, an advanced FPGA family from Xilinx.

The paper is organized around two actual system designs with similar overall organization. The first one is implemented on a XC2VP7-FG456-6 device and features a 32-bit data bus; the second one is implemented on a XC2VP30-FF896-7 and uses a 64-bit bus. The second system design was intended to implement the same basic approach as the first one, but to achieve better performance for tasks that depend on run-time reconfiguration.

Other systems based on dynamically reconfigurable platform FPGAs have been described in the literature (see, for instance, [13, 2, 1, 11]). Aspects of the 32-bit design used for this work have been presented in [5]. The 64-bit system is described here for the first time.

The rest of the paper is organized as follows. Section 2 describes the overall context of the work and presents the global design choices. The 32-bit system is summarized in section 3, which also characterizes the system's performance and presents experimental results for some simple application fragments. Section 4 then describes the 64-bit system, with emphasis on

1-4244-0054-6/06/$20.00

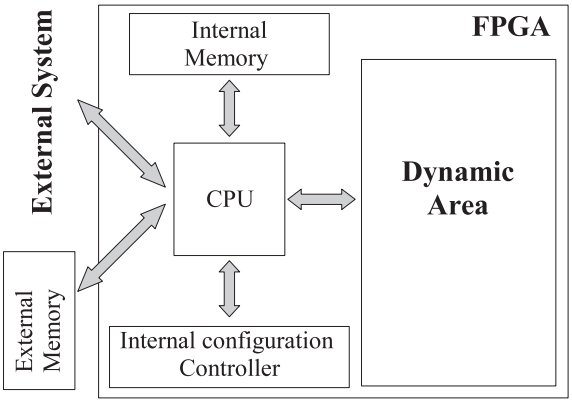

Figure 1. General system architecture

the different design choices and how they impacted the system's performance. The section also presents experimental results, including results for the same application fragments used with the 32-bit version. Finally, section 5 presents some final remarks and concludes the paper.

2 Dynamic reconfiguration of platform FPGAs

2.1 Generic system organization

A generic overview of a system organization used for this work is shown in figure 1. In addition to the CPU and the area for run-time reconfiguration (the dynamic area), the following modules are included:

- **Memory interface unit.** Depending on the needs, it may interface to internal and/or external memory. Both types of memory may store reconfiguration data (for the dynamic modules) and application-specific data.
- **Configuration control unit.** This module performs the actual reconfiguration of the dynamic area. It can be seen as an interface unit to the FPGA's configuration memory.
- **External communication unit.** This module, if available, is responsible for communications with an external system (e.g., a standalone computer) for data transfer, system control and debugging operations.
- **Dynamic area communication unit.** This unit is responsible for communications with the modules in the dynamic area. It includes the circuitry to use one of the data busses and, possibly, a DMA controller.

2.2 Partial configurations

The process of reconfiguring the dynamic area must handle the constraints imposed by the FPGA's architecture. Virtex-II Pro devices (like other Xilinx devices) are reconfigured by frames. A frame is a set of configuration bits that control a column of configurable resources. Each such column covers the entire height of the device. However, in practice, it is difficult to have a dynamic area that covers the entire height of the device, because that would isolate one side of the device from the other, i.e., circuits on the left of the dynamic area would not be able to connect to circuits on the left, and vice-versa. This may not be acceptable in practice. For instance, the layout of the board may constrain the layout of the resources inside the FPGA in such a way as to make a full-height dynamic region unavailable (typically because some external components are connected to pins on the upper or lower sides of the reconfigurable fabric).

Due to the need to take such layout constraints in consideration, a dynamic area will typically not occupy the full height of the device. Therefore, the partial configurations used to reconfigure the dynamic area must be produced in such a way as to not disturb the circuits below or above.

Another issue with the reconfiguration process arises because partial configurations are "differential" configurations, that is, they assume an initial state of the configuration resources, and only specify reconfiguration data for those resources whose state is to be different from the initial one. Since the dynamic area is used for multiple configurations in an order that is unknown at the time the partial configurations are produced, the problem of ensuring the correct state prior to reconfiguration arises.

Several ways to address the problem have been reported [11, 6, 10, 12]. One way is to have a tool like BitLinker [12], that is capable of ensuring that the configuration bitstream is complete (i.e., not "differential"). This has the side effect of increasing the configuration time.

The use of a configuration assembly tool may help solve other problems. For instance, BitLinker ensures that partial configurations do not disturb the circuits residing below or above the dynamic area.

It is possible to go further and provide means for assembling the correct partial configuration from the configurations of individual components [12, 6, 10]. In this way, components can be reused without going through the complete high-level design flow. This is particularly helpful when multiple similar configurations must be produced. All the configurations used in the experi-

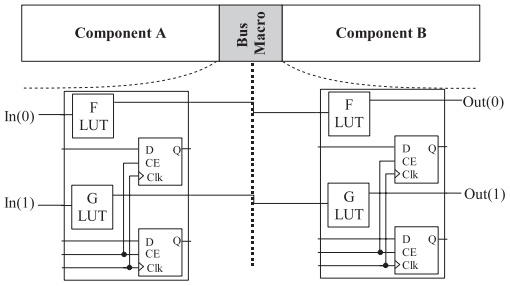

Figure 2. LUT-based bus macros

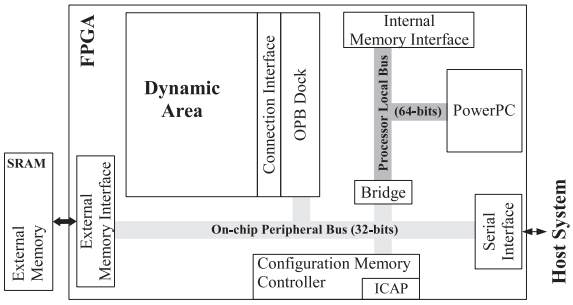

Figure 3. The 32-bit system architecture

ments reported in the next sections have been produced with the help of BitLinker.

The assembly of component configurations rises the problem of establishing communications between them. This requires a means of ensuring that the component's input and output ports are at fixed locations, so that the assembled configuration can be produced by appropriate concatenation of the individual components. To ensure that a component has input and output ports compatible with the assembly procedure, a so-called "bus macro" is used in the component's design. Figure 2 illustrates the situation. In this particular case, the communication is guaranteed by ensuring that the output signals of component A (called `In(0)` and `In(1)`) flow out of the component through specific LUTs and that the input signals of component B (called `Out(0)` and `Out(1)`) flow into the component from the corresponding LUTs.

Note that the components are designed separately (using the regular design flow): figure 2 shows the circuit that corresponds to the assembly of the configurations of the individual components, not the situation at design time. During the design process for A, no information about component B is used, except for the fact that the relative positions of the I/O connections are fixed by the "bus macro". The same applies to the design of component B and, indeed, to the design of any component that uses the same "bus macro".

"Bus macros" based on tristate connections have also been proposed, see [14]. The circuits mentioned in the next sections use LUT-based bus macros when necessary, since they consume less area.

3 The 32-bit system design

This section describes an implementation of the generic organization from section 2 around a 32-bit bus connection between CPU and dynamic area. The implementation described in section 3.1 corresponds to an updated implementation of the hardware system described in [5]. A performance assessment of the system is presented in section 3.2.

3.1 Implementation overview

For this system implementation we used a board with a Xilinx XC2VP7 (speed grade -6) FPGA and 32 MB of external static memory. This FPGA has 4928 slices and 44 RAM blocks (with 18 kb each).

An overview of the system setup is shown in figure 3. It corresponds roughly to the actual floorplan of the system. All modules (except the CPU) are implemented on the reconfigurable fabric. The on-chip intermodule bus system is an implementation of the CoreConnect Bus Architecture [7] as provided by the Xilinx Embedded Development Kit (EDK). The two busses used are the 64-bit Processor Local Bus (PLB) and the lower-performance, but less resource-consuming, 32-bit On-Chip Peripheral Bus (OPB).

The PLB connects to a memory controller for on-chip memory and to the PLB-OPB bridge. The OPB connects to the external memory controller and to the serial port. Using the OPB instead of the PLB to access external memory requires a much smaller controller. Although not shown on the figure, the OPB also connects to a General-Purpose I/O (GPIO) controller (for LEDs and push buttons). A Reset Block is also included; it can be used to externally reset the CPU and peripherals without affecting the fabric configuration. A dedicated block called JTAGPPC is also included; this special block connects the FPGA's JTAG port to the PowerPC core, and is used for data transfers and debugging.

The configuration memory controller (OPB HWICAP) is also connected to the OPB. Its purpose is to allow configurations to be changed internally through the Internal Configuration Access Port (ICAP), a dedicated block available in several Xilinx device families.

The Xilinx EDK was used to develop the system, so many on the necessary modules were already available.

The one remaining module that is directly relevant

Peripheral	Slices	LUTs	Flip Flops	BRAMs
PLB Arbiter	223 (5%)	334	54	
PLB Bram Controller	177 (4%)	136	249	16
PLB to OPB Bridge	524 (11%)	487	519	
OPB Arbiter	43 (1%)	53	6	
OPB SDRAM Controller	378 (8%)	510	295	
OPB Dock	77 (2%)	50	128	
OPB HWICAP	153 (3%)	208	155	1
OPB UART	64 (1%)	94	58	
OPB GPIO	77 (2%)	48	99	
Reset Block	48 (1%)	28	51	
Total	1764 (36%)	1948	1614	17

Table 1. Resource usage (32-bit system)

for this work is the OPB Dock. The OPB Dock is a a wrapper module, that connects the dynamic region to the rest of the system. It connects to the OPB bus in order to provide a 32-bit data channel to the dynamic region. The wrapper is assigned a fixed range of the OPB address space, and acts like an OPB slave peripheral, performing address decoding and I/O operations. The wrapper stores incoming data, so that it is kept available for processing by the components in the dynamic region between write operations.

The data communications between the wrapper and the dynamic region are made through a connection interface with two unidirectional channels, one for write and the other for read operations. Since the OPB is a 32-bit bus, each channel is 32 bits wide. The connection interface generates an additional signal, that indicates the occurrence of a write operation on the OPB. This signal can be used as a clock enable signal for any flip-flop in the dynamic region. The connection interface is implemented using the previously mentioned LUT-based bus macros.

The resource usage of the system implementation is shown in table 1. The CPU clock frequency is 200 MHz. Both the PLB and the OPB operate at 50 MHz. We were not able to obtain better operating frequencies while still satisfying the layout constraints required to obtain a dynamic area of useful size.

The dynamic region available in this implementation contains 6 RAM blocks and $28 \times 11 = 308$ Configurable Logic Blocks (CLBs). A Virtex-II Pro CLB includes 4 slices, each with two 4-input lookup tables and two flip-flops, so the dynamic area contains 25% of the total number of slices (and flip-flops).

3.2 Performance characterization

To assess data transfer performance, we measured the time necessary to transfer sequences of 32-bit values to/from external memory. Table 2 shows the average time per transfer for three situations: sequences

Number of operations	Average time per operation (µs)		
	Write	Read	Write/Read
1	4,80	4,90	5,66
10	1,95	1,84	2,56
100	1,67	1,56	2,21
1000	1,65	1,52	2,18
10000	1,65	1,52	2,18
100000	1,65	1,52	2,18

Table 2. Measured times for data transfers between dynamic region and external memory (32 bit)

Image size	Execution time (s)		Speedup
	Software	Hardware	
256 x 256	11,42	0,44	26,0
256 x 512	23,12	0,84	27,5
512 x 512	46,27	1,71	27,1
1024 x 1024	186,23	6,78	27,5

Table 3. Results for pattern matching in binary images (32 bit)

of write operations, sequences of read operations and sequences of interleaved write/read operations. The results include the overhead of the controlling software. Note that transfers between external memory and dynamic area use the data bus twice, since data is fetched from the origin to the CPU and then from the CPU to the destination.

The times reported in table 2 allow the developer to determine a lower bound for the time required to use the dynamic area. This lower bound can be used to make a first assessment of the improvements that can be obtained by moving a function from software to hardware.

Our first application example concerns a simple pattern matching task for bilevel images, where it is necessary to determine how many pixels of an 8×8 image pattern are equal to the corresponding pixels of a window that slides over a larger image. The hardware implementation is based around a pipeline of eight stages, each one calculating the number of matching pixels in a row of the pattern. The results of the eight stages are summed, producing the number of matching pixels for one position of the sliding window.

Table 3 shows the results obtained for a software-only implementation running on the embedded CPU versus the hardware/software version, where the dedicated matching pipeline is implemented in the dynamic area. As can be seen from the table, speedup factors of more than 26 were obtained. These results can be

Key length (bytes)	Execution time (ms)		Speedup
	Software	Hardware	
36	0,034	0,033	1,03
360	0,247	0,173	1,43
3600	2,384	1,578	1,51
36000	23,758	15,629	1,52
360000	237,489	156,144	1,52

Table 4. Results for hash function (32 bit)

Task	Execution time per output pixel (µs)		Speedup
	Software	Hardware	
Brightness adjustment	2,18	0,55	4,0
Additive blending	2,96	1,27	2,3
Fade effect	3,94	1,38	2,9

Table 5. Speedups for simple image processing tasks (32 bit)

explained by noting that: i) the task consists of many simple independent steps that can be executed in parallel; ii) a pipelined hardware implementation was used; iii) the task involves bit manipulations that are cumbersome to express in the C programming language, but simple to implement in hardware.

As another example consider the task of accelerating a public domain implementation of a hashing function that returns a 32-bit value for a variable-length key [8]. In this case, the whole hashing function was implemented in hardware. As the results of table 4 show, the speedup in this case is much more modest, since the original code had been optimized for 32-bit CPUs (like the PowerPC used in this case) and the data transfer times are significant when compared to the original software processing times.

Image processing tasks often involve the concurrent processing of small data items. The instruction set architectures of most desktop CPUs have been extended to include special instructions to handle packed sets of such data items [3, 9]. Since the PowerPC 405 core does not support such an extension, it makes sense to use the dynamic region to accelerate image processing tasks, that would otherwise be tackled by the CPU alone.

Table 5 presents the results obtained for some simple grayscale image processing applications (8-bit pixels):

- **Brightness adjustment:** The hardware adds an 8-bit unsigned pixel value to a signed constant value (saturating add). Four pixels are processed per data transfer.
- **Additive blending:** This task consists of adding (with saturation) the pixel values from two images to produce a third. The hardware receives four pixel values per transfer (two from each image) and produces two output pixels. In order to save on read operations, the resulting pixels are packed in groups of four, before being read back by the CPU.
- **Fade effect:** This task consists of combining the pixels of two images according to $(A-B)\times f+B$, where A is a pixel value from the first image, B is a pixel value from the second, and f is a constant that specifies the relative contribution of the first image to the result [9]. The fade-in-fade-out effect is obtained by processing the source images successively for different values of f. The data transfer pattern is identical to the one used in the additive blending task.

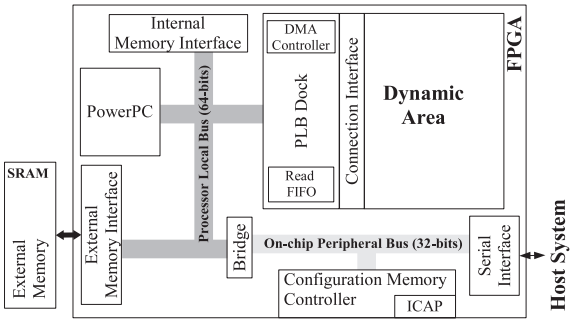

Figure 4. The 64-bit system architecture

Note that the two last tasks require that data from two sources be combined by the CPU, before being sent to dynamic area. This overhead is included in the measured times for the hardware implementation, and helps to explain the smaller speedups obtained in these two cases. The additive blending operation is simpler than the fade effect operation, and hence benefits less from being implemented in hardware.

4 The 64-bit system design

This section describes an implementation of the generic design from section 2, where the dynamic area is connected to the 64-bit processor local bus. The main differences in relation to the 32-bit designed are summarized in section 4.1 and the performance measurements are described in section 4.2.

4.1 Implementation aspects

The design from section 3 had a 32-bit bus, in order to use less resources and have a dynamic area with a usable size. For the alternative system implementation presented here, we used a board with a Xilinx XC2VP30 device and an external DDRAM memory of

512 MB. This device includes two CPU cores, but only one is used. The FPGA has 13696 slices (about 2.7 times more slices than the previously used device) and 136 internal RAM blocks. The speed grade is also better (-7).

An overview of the system setup is shown in figure 4. As for the previous system, this figure corresponds roughly to the actual floorplan of the system. Again, all modules except the CPU are implemented on the reconfigurable fabric.

When compared to the previous system, the present implementation has two main differences: i) the external memory controller is located on the 64-bit PLB; ii) the dynamic region wrapper is also connected to the processor local bus and has some added functionality (it is now called the PLB Dock). Minor differences include the addition of an interrupt controller attached to the OPB and the absence of the GPIO controller.

The PLB Dock now provides a 64-bit data channel to the dynamic region. The wrapper is assigned a fixed range of the PLB address space, and acts like an PLB master/slave peripheral. Besides performing address decoding for I/O operations and storing incoming data (like the OPB Dock implementation), this wrapper has three additional capabilities:

1. **DMA controller:** Direct transfers between memory and PLB dock are now possible without CPU intervention.
2. **Output FIFO:** The results produced by the dynamic area can be stored in a FIFO for subsequent DMA transfer to memory.
3. **Interrupt generator:** The PLB dock can send interrupts to the CPU.

The data communication between wrapper and dynamic region is unchanged, with the obvious difference that the channels are now 64-bit wide.

The main reason for moving the dynamic region from the OPB to the PLB was to obtain a better data transfer performance. Since the CPU does not support 64-bit wide data transfers at the instruction level (load and store instructions handle items of size up to 32 bits), program transfers to/from the dynamic area cannot directly benefit from the increased data width. Only transfers that go through the caches use 64-bit transfers. Therefore, communication between CPU and dynamic region through load and store instructions is still made by 32-bit-wide transfers. Without further measures the change would only benefit software implementations.

In order to use the full bus width, the PLB dock includes a scatter-gather DMA controller that supports 64-bit transfers. The controller is automatically generated by the Xilinx development tools. To profit from the DMA, data transfers to the dynamic area have to be done as a block; therefore, a memory buffer, the output FIFO, must be provided to store the results produced by the dynamic area, before they are sent to main memory.

Peripheral	Slices	LUTs	Flip Flops	BRAMs
PLB Arbiter	317 (2%)	493	66	
PLB Bram Controller	187 (1%)	163	259	32
PLB to OPB Bridge	552 (4%)	543	529	
OPB Arbiter	40 (0%)	41	6	
PLB DDRAM Controller	991 (7%)	1196	935	
PLB Dock	1761 (13%)	2275	1470	8
OPB HWICAP	139 (1%)	181	151	1
OPB UART	63 (0%)	97	59	
OPB Interrupt Controller	82 (1%)	55	76	
Reset Block	49 (0%)	28	51	
Total	4181 (31%)	5072	3602	41

Table 6. Resource usage (64-bit system)

Since the CPU is free during DMA transfers, it can be used for other purposes. To avoid the need for polling the PLB dock to determine the status of the transfers, an interrupt generator was added to the dock, thus requiring the inclusion of an interrupt controller in the design.

As the previous discussion made clear, the permanent circuits implemented on the reconfigurable fabric are larger and more complex for the second design. Resource usage of the system setup is summarized in table 6. Since the FPGA device used is faster and the layout constraints are less severe, the CPU clock frequency in this case is 300 MHz (vs. 200 MHz in the previous design) and both the PLB and the OPB operate at 100 MHz (vs. 50 MHz previously). It is clear that, without the introduction of DMA transfers, the design modifications would be more favorable for software-only implementations.

The dynamic region available in the new version contains 22 BRAMs and $32 \times 24 = 768$ CLBs, i.e., 3072 slices (22.4% of the total). The use of the remaining free slices is made more difficult by the presence of the second CPU core and alternative approaches (like having two separate dynamic areas) may be necessary to put them to use.

4.2 Performance assessment

To assess data transfer performance, we again measured the time necessary to transfer sequences of data to/from external memory. In this case, two situations must be considered, according to whether the data transfers are controlled by the CPU (as is the case with the 32-bit system) or by the DMA controller.

Table 7 shows the average time taken by program-

Number of operations	Average time per operation (µs)		
	Write	Read	Write/Read
1	1,01	0,96	1,20
10	0,34	0,30	0,64
100	0,26	0,24	0,58
1000	0,25	0,24	0,58
10000	0,25	0,24	0,58
100000	0,25	0,24	0,58

Table 7. Measured times for 32-bit data transfers between dynamic region and external memory (CPU controlled)

Number of operations	Average time per operation (µs)		
	Write	Read	Write/Read
1	9,55	9,81	12,61
10	0,95	0,98	1,29
100	0,14	0,13	0,20
1000	0,06	0,04	0,10
10000	0,05	0,04	0,09
100000	0,05	0,03	0,09

Table 8. Measured times for 64-bit data transfers between dynamic region and external memory (DMA-controlled)

Image size	Execution time (s)		Speedup
	Software	Hardware	
256 x 256	2,18	0,11	19,8
256 x 512	4,36	0,23	19,0
512 x 512	8,84	0,46	19,2
1024 x 1024	35,59	1,84	19,3

Table 9. Results for pattern matching in binary images (64 bit).

Key length (bytes)	Execution time (ms)		Speedup
	Software	Hardware	
36	0,008	0,008	1,00
360	0,061	0,039	1,56
3600	0,588	0,349	1,68
36000	5,855	3,455	1,69
360000	58,526	34,509	1,70

Table 10. Results for a hash function implementation (64 bit)

controlled transfers: sequences of write operations, sequences of read operations and sequences of interleaved write/read operations. In this case, each transfer involves a 32-bit value (as discussed previously). This operation is the same as the one performed in the 32-bit system and direct comparison of the values is legitimate. A decrease in transfer time between 4 and 6 times, depending on the transfer type, can be observed. Part of this decrease is due to improved bus speed (a factor of 2) and CPU frequency (a factor of 1.5). The additional improvement presumably comes from the fact that no PLB-to-OPB bridge is used.

Table 8 shows the average time per transfer when using DMA. In this method, each transfer involves a 64-bit value, using the data path to the fullest. The interleaved write/read operations are block-interleaved: the output data is stored in the output FIFO, while the write operation is being performed; when the FIFO becomes full, the write operation stops and the data contained in the FIFO is transferred the external memory by a DMA operation. These operations are repeated, until all the data is transferred. The current output FIFO stores up to 2047 64-bit values.

The times reported in tables 7 and 8 allow the developer to determine a lower bound for the time required to use the dynamic area with different transfer methods and different data widths. This lower bound can be used to make a first assessment of the improvements, that can be obtained by moving a function from software to hardware, and to evaluate the gains from using each of the two data transfers methods.

We took our first two implementation examples from the 32-bit system, and transferred them on the new system without any modifications: the data transfers don't take advantage of the 64-bit bus width and are controlled by the CPU. Tables 9 and 10 present the results. Both tasks benefit greatly from the new system and both software and hardware implementations perform considerably better.

In general, the results follow the trends observed for the transfer times, as expected. In the pattern matching task, a decrease in the hardware vs. software speedup is obtained, because the software implementation benefited more from the quicker access to memory. The hardware implementations still maintain a considerable performance advantage. The hash value calculation task, on the other hand, shows only a slightly better speedup for the hardware implementation.

We also tested the system with the more demanding hash function SHA1 [4]. This hashing algorithm is geared towards 32-bit implementations. Our implementation does not fit into the dynamic area of the 32-bit system, so no comparison can be done. The results of table 11 show a considerable performance gain for the hardware implementation (using 32-bit CPU-controlled data transfers). The software implementation (taken from the RFC document) has a large over-

Data length (bytes)	Execution time (ms)		Speedup
	Software	Hardware	
64	24,35	0,02	1217,5
640	30,54	0,24	127,3
6400	47,91	2,35	20,4
64000	261,95	23,51	11,1
640000	2402,56	235,09	10,2

Table 11. Results for SHA-1 implementation

Task	Execution time per output pixel (µs)			Speedup
	Software	Hardware		
		Data preparation	Total	
Brightness adjustment	0,48	n.a.	0,01	48,0
Additive blending	0,64	0,19	0,20	3,2
Fade effect	0,90	0,19	0,20	4,5

Table 12. Results for simple image processing tasks (64 bit)

head for smaller data sets. The overhead's relative importance decreases for larger data sets.

A 64-bit DMA-controlled implementation of the image processing tasks presented in section 3.2 was also made. The results are shown in table 12. For the first task, there is a clear increase of the speedup obtained by the hardware (on top of the increased performance of the software version). The reason for this is that the 64-bit data transfers could be employed without additional work, since only one image is involved. The other tasks show a significantly smaller speedup increase, because the data of the two source images had to be combined by the CPU, before being sent to the dynamic area. This time overhead appears in the table under the heading of "data preparation", and is directly attributable to the constraints of the DMA transfer mode.

5 Conclusion

The paper discusses several issues, that arise when trying to exploit effectively the run-time reconfiguration capabilities of platform FPGAs. Two implementations of the same general approach are presented, and compared with the help of several small studies. The issues associated with data transfers between embedded CPU and dynamically reconfigured circuits are shown to contribute significantly to overall performance.

For the systems considered in this work, the use of 64-bit data transfers is hampered by the fact that the CPU does not support programmatic 64-bit data transfers; only transfers that go through the caches profit from the higher bus width. In order to use the 64-bit bus width effectively to communicate with the dynamic area, it is necessary to employ DMA transfers. These, however, pose significant restrictions on data organization and access patterns, making the adaptation of the software-based algorithms more difficult and time consuming. When the difficulties can be overcome, significantly better performance can be achieved. It is also important to note that the changes from the first to the second system affect hardware and software in a different manner, so the relative merits of using a pure software approach vs. a combined hardware/software solution also change.

References

[1] B. Blodget, C. Bobda, M. Hübner, and A. Niyonkuru. Partial and dynamically reconfiguration of Xilinx Virtex-II FPGAs. In *Proceedings FPL'04*, pages 801–810, 2004.

[2] E. Carvalho, N. Calazans, E. Brião, and F. Moraes. PaDReH: a framework for the design and implementation of dynamically and partially reconfigurable systems. In *Proceedings SBCCI'04*, pages 10–15, 2004.

[3] K. Diefendorff, P. Dubey, R. Hochsprung, and H. Scale. AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85–95, 2000.

[4] D. Eastlake and P. Jones. *RFC 3174 — US Secure Hash Algorithm 1 (SHA1)*. RFC Editor, Sept. 2001.

[5] J. C. Ferreira and M. M. Silva. Run-time reconfiguration support for FPGAs with embedded CPUs: The hardware layer. In *Proceedings RAW'05*, Denver, Colorado, Apr. 2005.

[6] E. L. Horta, J. W. Lockwood, and S. T. Kofuji. Using PARBIT to implement partial run-time reconfigurable systems. In *Proceedings FPL'02*, pages 182–191, London, UK, 2002. Springer-Verlag.

[7] IBM. The CoreConnect bus architecture, Sept. 1999.

[8] B. Jenkins. Hash functions. *Dr. Dobb's Journal*, 22(9):107–109, Sept. 1997.

[9] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for multimedia PCs. *Commun. ACM*, 40(1):24–38, 1997.

[10] A. K. Raghavan and P. Sutton. JPG–a partial bitstream generation tool to support partial reconfiguration in Virtex FPGAs. In *Proceedings IPDPS'02*, page 192, Washington, DC, USA, 2002. IEEE Computer Society.

[11] P. Sedcole, B. Blodget, J. Anderson, P. Lysaght, and T. Becker. Modular partial reconfiguration in Virtex FPGAS. In *Proceedings FPL'05*, pages 211–216, 2005.

[12] M. L. Silva and J. C. Ferreira. Generation of hardware modules for run-time reconfigurable hybrid CPU/FPGA systems. In *Proceedings DCIS'05*, Lisboa, Portugal, Nov. 2005.

[13] M. Ullmann, M. Hübner, B. Grimm, and J. Becker. An FPGA run-time system for dynamical on-demand reconfiguration. In *IPDPS'04*, page 135a. IEEE Computer Society, 2004.

[14] Xilinx. Two flows for partial reconfiguration: Module base or small bit manipulations. Application note 290, Sept. 2004.