Parametric Design for Reconfigurable Software-Defined Radio

Tobias Becker[1], Wayne Luk[1], and Peter Y.K. Cheung[2]

[1] Department of Computing, Imperial College London, UK
[2] Department of Electrical and Electronic Engineering, Imperial College London, UK

Abstract. Run-time reconfigurable FPGAs are powerful platforms for realising software-defined radio systems. This paper introduces a parametric approach to designing such systems based on application and device parameters. We analyse the potential for reconfiguration in several software-defined radio components and demonstrate how the degree of parallelism in a reconfigurable module influences reconfiguration time and performance. In a case study with a reconfigurable FIR filter, our method increases the performance by a factor of 2.4.

1 Introduction

Software-defined radio is an emerging technology that involves processing digital radio signals by means of software techniques. Today's mobile applications feature an abundance of radio standards with ever increasing bandwidth. This leads to a demand in both increased design productivity and flexibility after device deployment. Software techniques are seen as a solution to quickly design flexible mobile applications. In this paper we consider the implementation of the physical radio interface on reconfigurable hardware devices, such as FPGAs, which combine hardware-like performance and efficiency with software-like flexibility.

FPGAs are powerful computing devices that deliver performance through parallelism and can be more efficient than DSPs or general-purpose processors. Combining reprogrammability and performance, they are displacing ASICs in telecom equipment such as mobile base stations. SRAM-based FPGAs can also be reconfigured at run-time, a feature that has been subject to much research [1]. Run-time reconfiguration has also been proposed for software-defined radio applications [2] and successfully demonstrated [3].

This paper focuses on a structured design approach based on application and device parameters. Most importantly, this includes analysing how the degree of parallelism in a reconfigurable application affects reconfiguration time and global performance. We further consider application data buffering and configuration storage. Using an example of software-defined radio, we show how our method can be used to explore the design space and optimise the reconfigurable system.

The rest of the paper is organised as follows. Section 2 explains the background of software-defined radio and outlines four basic reconfiguration scenarios. In section 3, we analyse the potential for reconfiguration in several components

J. Becker et al. (Eds.): ARC 2009, LNCS 5453, pp. 15–26, 2009.

of a digital receiver. In section 4, we introduce our parametric approach and show that choosing between a serial or parallel implementation can be used to optimise performance and area. In section 5, we illustrate our approach on the example of a reconfigurable FIR filter. Finally, section 6 concludes the paper.

2 Background

Software-defined radio was first motivated by moving from analogue to digital technology in radio systems [4]. Because of the exponential increase in performance and productivity of digital technology compared to slow advances in analogue circuits, digital technology will continue to move closer to the antenna and replace much of the analogue front end.

Figure 1 shows a receiver where the signal is digitised directly after being mixed to an intermediate frequency (IF). A digital radio however is not synonymous with a software-defined radio. In current mobile phones, digital baseband processing is usually performed by inflexible but power-efficient ASICs. In order to support multiple standards, multiple pieces of IP have to be assembled into a complex system, that subsequently needs to be tested and manufactured. In contrast, a software-defined radio performs the same task by employing software-programmable and flexible DSPs or general-purpose processors in order to support multiple radio standards [5].

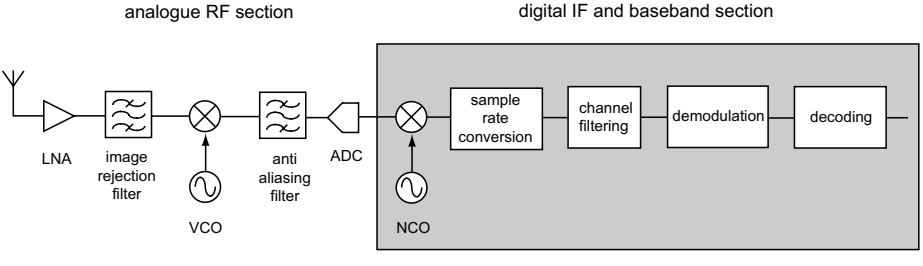

Fig. 1. Receiver architecture of a software-defined radio

However, classic software-programmable architectures such as DSPs or processors are often too inefficient to satisfy the power and performance constraints of communication devices. FPGAs are considered as a solution to this because they can deliver the flexibility of programmable architectures with power efficiency and performance much closer to ASICs. But despite successful demonstration of run-time reconfiguration for software-defined radios [3], we find a lack of systematic methodology in designing and optimising such systems. We therefore propose a more structured approach. Table 1 identifies different opportunities for reconfiguration in software-defined radio.

Although reconfigurability has promising aspects in all four scenarios, we mainly consider the latter two because they are the most challenging to realise.

Table 1. Scenarios showing opportunities for reconfiguration in software-defined radio

1. *Pre-deployment*. The programmability of the target architecture is used to make changes late in the design process before the device is deployed. Generic programmable devices also simplify the design by reducing the number of components needed.
2. *In-field upgrade*. The device is updated to support a new standard or feature that was not included at deployment. In this case, the device will be rebooted and downtime during reconfiguration does not matter.
3. *Reconfiguration per call or session*. The device is reconfigured at the beginning of a call or data transfer session e.g. to select the most efficient or cheapest service available at the moment. In this case, the reconfiguration downtime has to be short, preferably below the level of human perception but no data need to be buffered.
4. *Reconfiguration during a call or session*. The device is reconfigured during a call or data session e.g. to hand over from one service to another. Reconfiguration downtime time has to be very short and streaming data has to be buffered during reconfiguration.

3 Software-Defined Radio

Table 2 gives an overview of some of the most important communication standards that are currently being employed. These standards vary widely in their technical details and satisfy different needs. We now analyse how the components of a digital receiver as illustrated in figure 1 can benefit from reconfiguration.

As the first digital processing step, the real signal is down-converted into a complex baseband signal. This is done by multiplying the IF signal with sine and cosine signals from a numerically controlled oscillator (NCO). One common method of generating sine and cosine functions in digital hardware is the

Table 2. Comparison of different mobile communication standards

	GSM	EDGE	DECT	UMTS TDD	UMTS FDD	Bluetooth
Frequency band	900, 1800, 1900 MHz	900, 1800, 1900 MHz	1.9 GHz	1.9, 2 GHz	1.9, 2.1 GHZ	2.4 GHz
Channel bandwidth	200 kHz	200 kHz	1728 kHz	1.6 or 5 MHz	5 MHz	1 MHz
Channel access	FDMA/TDMA	FDMA/TDMA	FDMA/TDMA	TD-CDMA	DS-CDMA	TDMA
Duplex	FDD	FDD	TDD	TDD	FDD	TDD
Modulation	GMSK	GMSK, 8-PSK	GMSK	QPSK	QPSK	GFSK
Pulse shaping	Gauss (BT = 0.3)	Linear. Gaussian	Gauss (BT = 0.5)	Root-raised cos (β = 0.22)	Root-raised cos (β = 0.22)	Gauss (BT = 0.5)
Error correction	convolutional, CRC	convolutional CRC	CRC	convolutional, turbo, CRC	convolutional, turbo, CRC	-
Slot duration	0.577 ms	0.577 ms	0.417 ms	0.667 ms	0.667 ms	0.625 ms
Bit or chip rate	270.83 kbit/s	270.83 kbit/s	1152 kbit/s	1.28 or 3.84 Mchip/s	3.84 Mchip/s	1 Mbit/s
Net bit rate	13 kbit/s	up to 473.6 kbit/s	32 kbits/s	up to 2 Mbit/s	up to 2 Mbit/s	1Mbit/s

CORDIC algorithm. Since CORDIC is an iterative algorithm, a pipelined version would most likely be used to provide sufficient throughput. It is also possible to use an optimised, reconfigurable CORDIC processor to achieve higher performance and less area requirements [6]. The down-converter must be able to adapt quickly to different frequencies in order to support a change of channel between transmission slots (see table 2). This corresponds to scenario 4 in table 1.

Filters are applied at multiple stages of the receiver. In order to select the channel of interest, all other signals have to be removed. In practise, this is done by combining multiple stages of filtering and decimation. IIR filters are usually avoided due to instability issues and their non-linear phase. FIR filters are used instead, but in wideband architectures they can require a high order to filter narrowband-signal at high sample rate. Filters can be reconfigured between changing communication standards (scenario 3) or during a call or data session (scenario 4).

The last step in the receiver chain is decoding and error correction. Different methods of cyclic redundancy checks (CRC), convolutional codes and turbo codes are employed. GSM for example uses CRC for the most important bits and a convolutional decoder for less important bits. UMTS also uses turbo codes for error correction. Again, the reconfigurability can categorised as scenario 3 if the error correction is changed between standards. However, it is also possible to change an error correction to adapt to different channel conditions during a call or data session [7]. This case corresponds to scenario 4.

Finally there could be an additional encryption scheme such as AES at the end of the receiver chain. This is completely independent from any particular radio standard and could be reconfigured during or between sessions.

4 Design Parameters in Software-Defined Radio

Many run-time reconfigurable systems are designed in an ad-hoc manner. During the design process it is often unclear how the final implementation will perform. We propose a structured design approach that identifies design parameters and maps these onto the reconfigurable device. In the following, we consider three implementation attributes: performance, area and storage. First, we identify parameters of a reconfigurable module:

- Area requirement A
- Processing time t_p for one packet or datum
- Reconfiguration time t_r
- Configuration storage size Ψ
- The number of processing steps s in the algorithm
- The amount of parallelism p in the implementation

We can also identify parameters of the application that employs a reconfigurable module:

- Required data throughput ϕ_{app}
- The number of packets or items of data n that are processed between reconfigurations

The reconfigurable device is characterised by the following parameters:

- The available area A_{max}
- The data throughput of the configuration interface ϕ_{config}
- The configuration size per resource or unit of area Θ

4.1 Storage Requirements and Reconfiguration Time

All designs on volatile FPGAs require external storage for the initial configuration bitstream. Designs using partial run-time reconfiguration also need additional storage for the pre-compiled configuration bitstreams of the reconfigurable modules. The partial bitstream size and storage requirement Ψ (in bytes) of a reconfigurable module is directly related to its area A:

$$\Psi = A \cdot \Theta + h \approx A \cdot \Theta \tag{1}$$

A is the size of a reconfigurable module in FPGA tiles (e.g. CLBs) and Θ is a device specific parameter that specifies the number of bytes required to configure one tile. Configuration bitstreams often contain a header h. In most cases, this can be neglected because the header size is very small.

The time overhead of run-time reconfiguration can consist of multiple components, such as scheduling, context save and restore as well as the configuration process itself. In our case there is no scheduling overhead as modules are loaded directly as needed. There is also no context that needs to be saved or restored since signal processing components do not contain a meaningful state once a dataset has passed through. The reconfiguration time is proportional to the size of the partial bitstream and can be calculated as follows:

$$t_r = \frac{\Psi}{\phi_{config}} \approx \frac{A \cdot \Theta}{\phi_{config}} \tag{2}$$

ϕ_{config} is the configuration data rate and measured in *bytes/s*. This parameter not only depends on the native speed of the configuration interface but also on the configuration controller and the data rate of the memory where the configuration data are stored.

4.2 Buffering

As outlined in table 1, we can distinguish between run-time reconfigurable scenarios where data do not have to be buffered during reconfiguration, and scenarios where data buffering is needed during reconfiguration. For the latter case we can calculate the buffer size B depending on reconfiguration time t_r and the application data throughput ϕ_{app}:

$$B = \phi_{app} \cdot t_r = \frac{\phi_{app}}{\phi_{config}} \cdot \Psi \tag{3}$$

Table 3 outlines the buffer size for several receiver functions and a range of reconfiguration times. We can observe that the data rate is reduced through

Table 3. Buffersize for various functions and reconfiguration times

function	data throughput	buffer size for a given reconfiguration time		
		100 ms	10 ms	1 ms
down-conversion (16 bit)	800 Mbit/s	80 Mbit	8 Mbit	800 kbit
down-conversion (14 bit)	700 Mbit/s	70 Mbit	7 Mbit	700 kbit
demodulation UMTS	107.52 Mbit/s	10.75 Mbit	1.07 Mbit	107 kbit
demodulation GSM	7.58 Mbit/s	758 kbit	75.8 kbit	7.58 kbit
error correction UMTS	6 Mbit/s	600 kbit	60 kbit	6 kbit
error correction GSM	22.8 kbit/s	2.28 kbit	228 bit	22.8 bit
decryption UMTS	2 Mbits/s	200 kbit	20 kbit	2 kbit
decryption GSM	13 kbit/s	1.3 kbit	130 bit	13 bit

all stages of the receiver. Hence, a reconfiguration-during-call scenario becomes easier to implement towards the end of the receiver chain. Obviously, the buffer size also increases with the bandwidth of the communication standard and the duration of the reconfiguration time.

A buffer can be implemented with on-chip or off-chip resources. Most modern FPGAs provide fast, embedded RAM blocks that can be used to implement FIFO buffers. For example, Xilinx Virtex-5 FPGAs contain between 1 to 10 Mbit of RAM blocks [8]. Larger buffers have to be realised with off-chip memories.

4.3 Performance

The performance of a run-time reconfigurable system is dictated by the reconfiguration downtime. If reconfigurable hardware is used as an accelerator for software functions, overall performance is usually improved despite the configuration overhead [9]. In our case we use reconfiguration to support multiple hardware functions in order to improve flexibility and reduce area requirements. In this case, the reconfigurable version of a design will have a performance penalty over a design that does not use reconfiguration. The reconfiguration of hardware usually takes much longer than a context switch on a processor. This is due to the relatively large amount of configuration data that need to be loaded into the device. Early research on run-time reconfiguration showed that a reconfigurable design becomes more efficient the more data items n are processed between reconfigurations [10]. The efficiency I of a reconfigurable design compared to a static design can be expressed as:

$$I = \frac{t_{static}}{t_{reconf}} = \frac{n \cdot t_p}{n \cdot t_p + t_r} = \frac{n}{n + \frac{t_r}{t_p}} \tag{4}$$

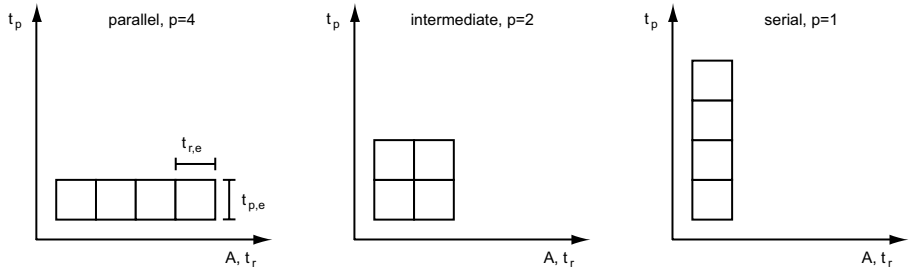

Fig. 2. Different spatial and temporal mappings of an algorithm with $s = 4$ steps

The reconfigurable system becomes more efficient by processing more data between configurations and by improving the ratio of configuration time to processing time. We propose a more detailed analysis where we consider the effect of parallelism on processing time and configuration time. Many applications can be scaled between a small and slow serial implementation, and a large and fast parallel or pipelined implementation. FIR filter, AES encryption or CORDIC are examples of such algorithms.

Figure 2 illustrates the different spatial and temporal mappings of an algorithm with regard to processing time, area and reconfiguration time. The processing time per datum t_p is inversely proportional to the degree of parallelism p. It can be calculated based on $t_{p,e}$, the basic processing time of one processing element, s, the number of steps or iterations in the algorithm, and p, the degree of parallelism:

$$t_p = \frac{t_{p,e} \cdot s}{p} \tag{5}$$

Parallelism speeds up the processing of data but slows down reconfiguration. This is because a parallel implementation is larger than a sequential one, and the reconfiguration time is directly proportional to the area as shown in equation 2. The reconfiguration time t_r is directly proportional to the degree of parallelism p, where $t_{r,e}$ is the basic reconfiguration time for one processing element:

$$t_r = t_{r,e} \cdot p \tag{6}$$

We can now calculate the total processing time for a workload of n data items:

$$t_{total} = n \cdot t_p + t_r = \frac{t_{p,e} \cdot s \cdot n}{p} + t_{r,e} \cdot p \tag{7}$$

Figure 3 illustrates how parallelism can affect the optimality of the processing time. We consider an algorithm with $s = 256$ steps which is inspired by the observation that filters can have orders of 200 or higher. The plots are normalised to processing time per datum and we assume that the reconfiguration time $t_{r,e}$ of one processing element is 5000 times the processing time $t_{p,e}$ of one processing element. This value can vary depending on the application and target device but we estimate that at least the order of magnitude is realistic for current

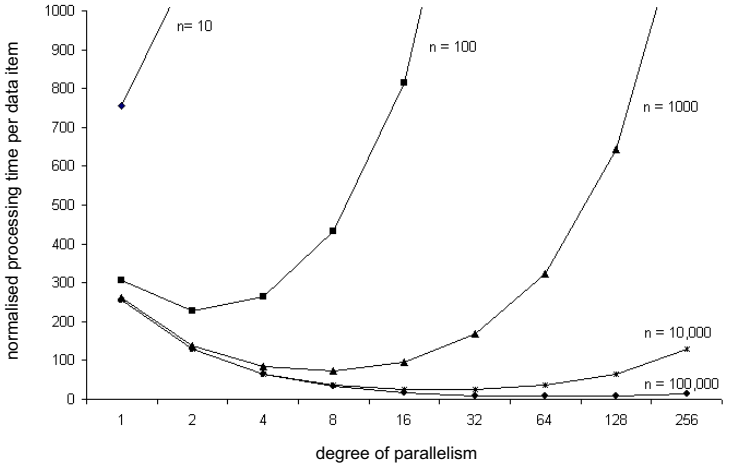

Fig. 3. Normalised processing times for a range of workload sizes n and different levels of parallelism p. The number of steps s is set to 256 and we assume $t_{r,e} = 5000t_{p,e}$.

devices. We can observe that fully sequential implementations are beneficial for small workloads. In this case, the short configuration time outweighs the longer processing time. However, the overall time is still high due to the large influence of the configuration time. Large workloads benefit from a fully parallel implementations since the processing time is more dominant than reconfiguration time. In case of medium workloads, the degree of parallelism can be tuned to optimise the processing time. An analysis of how different configuration speeds can influence the optimal implementation is shown in section 5.

In order to find the optimal degree of parallelism, we calculate the partial derivative of the function given in equation 7 with respect to p:

$$\frac{\partial t_{total}}{\partial p} = -\frac{t_{p,e} \cdot s \cdot n}{p^2} + t_{r,e} \tag{8}$$

To find the minimum, we set equation 8 to 0 and solve for p:

$$p_{opt} = \sqrt{\frac{s \cdot n \cdot t_{p,e}}{t_{r,e}}} \tag{9}$$

The result p_{opt} is usually a real number which is not a feasible value to specify parallelism. In order to determine a practical value for p, p_{opt} can be interpreted according to table 4.

After determining the optimal degree of parallelism that reduces the overall processing time per workload and hence maximises performance, it is still necessary to check if the implementation meets the throughput requirements of the application Φ_{app}:

$$\frac{n}{t(p)_{total}} = \Phi_{hw} \geq \Phi_{app} \tag{10}$$

Table 4. Interpretatipn of p_{opt} to determine a practical value for p

$0 < p_{opt} \leq 1$	fully serial implementation, $p = 1$
$1 < p_{opt} < s$	choose p such that $s/p \in \mathbb{Z}$ and $\lvert p_{opt} - p \rvert$ minimal
$s \leq p_{opt}$	fully parallel implementation, $p = s$

The resulting area requirement A also has to be feasible within the total available area A_{max}. In summary, to implement an optimised design according to our model, the following steps have to be carried out:

1. Derive Φ_{app}, s and n from application.
2. Obtain Φ_{config} for target technology.
3. Develop one design and determine t_p and A.
4. Calculate t_r, $t_{p,e}$ and $t_{r,e}$ using equations 2, 5 and 6.
5. Find p_{opt} from equation 9 and find a feasible value according to table 4.
6. Calculate t_{total} using equation 7 and verify throughput using equation 10.
7. Implement design with p from step 5 and verify if its actual throughput satisfies the requirement.
8. Calculate buffer size B using equation 3 and check $A \leq A_{max}$.

The above methodology can be adopted for a wide variety of applications and target technologies; it will find the highest performing version of the design. In order to find the smallest design that satisfies a given throughput requirement, one can try smaller values for p while checking equation 10.

5 Case Study: Channelisation Filter for GSM

We now demonstrate our method on the example of a channelisation FIR filter for GSM. As *step 1*, we determine the application parameters ϕ_{app}, s and n. The sample rate f_s in our example is $2.167MHz$ which corresponds to 8 times the baseband bit rate. This is a realistic scenario after a first round of filtering and decimation. With 16-bit samples, the application throughput Φ_{app} is $34.672MBit/s$. We filter a 200 kHz wide GSM channel and suppresses neighbouring channels with an attenuation of at least $-80dB$. The filter coefficients are calculated with the MATLAB Simulink Digital Filter Design blockset. This results in a filter with 112 coefficients. The number of processing steps s is therefore 112. We consider a scenario where the filter needs to be reconfigured between GSM frames. One GSM frame has a duration of $4.615ms$ and produces $10,000$ samples at the given sample rate. Hence, the workload size $n = 10,000$.

Step 2. We use a Xilinx Virtex-4 LX25 FPGA as target device for the implementation of our filter. Virtex-4 FPGAs provide an internal configuration access port (ICAP) that is 32 bit wide and runs at $100MHz$. This corresponds to a theoretical maximal configuration throughput $\Phi_{config} = 400MB/s$. However, when using the HWICAP core [11] in combination with a MicroBlaze processor

we only measure a throughput of $5MB/s$. But Claus et.al. recently presented an improved version with a throughput of approximately $300MB/s$ [12]. We therefore compare two scenarios of slow reconfiguration based on our measurement with $\Phi_{config} = 5MB/s$ and fast reconfiguration based on $\Phi_{config} = 300MB/s$.

Step 3. We first implement a fully parallel version of our filter as illustrated in the first row of table 5. The filter is created with Xilinx CORE Generator and implemented with ISE 9.2. The filter requires 6921 slices and is implemented in an area A of 1792 CLBs. In Virtex-4 FPGAs, the configuration size per CLB is $225.5bytes/CLB$. Hence, its corresponding partial bitstream has a size of $\Psi = 404.1kB$ (note: we use 1kB = 1000 bytes).

Step 4. The processing time per datum t_p is $4ns$. Since this is a fully parallel version, the processing time per element $t_{p,e}$ is also $4ns$. With HWICAP, the filter can be reconfigured in $t_r = 81ms$ and $t_{r,e}$ is $0.72ms$. When using the fast ICAP version we estimate that $t_r = 1.35ms$ and $t_{r,e} = 0.012ms$.

Step 5. For slow configuration, the optimal value for $p_{opt} = 2.49$ and feasible values are 2 and 4. With fast reconfiguration, $p_{opt} = 19.3$ and feasible values are 14 and 28.

Step 6. For slow reconfiguration we expect processing times of $t_{total} = 3.68ms$ for $p = 2$ and $t_{total} = 4.01ms$ for $p = 4$ and for fast reconfiguration we expect t_{total} to be $0.49ms$ for $p = 14$ and $0.50ms$ for $p = 28$. All of these satisfy our requirement of processing one frame in $4.615ms$.

Step 7. Our previous results are projections based on a fully parallel version. We now verify these results by comparing them to attributes of real implementations. Table 5 shows six different versions of the filter with p ranging from 112 to 4. Smaller, more serial versions could not be created with CORE Generator. We observe that $t_{p,e}$ and $t_{r,e}$ are not constant as previously assumed. Especially configuration size and reconfiguration time do not scale linearly with the degree of parallelism. This is because serialising a design introduces an overhead for the implementation. For $p = 4$ and with slow reconfiguration, the filter has an actual t_{total} of $6.8ms$ which violates our requirement. We therefore have to consider that projections with low degrees of parallelism and operating close to the limit of the application requirement can fail. With fast reconfiguration, our projection is correct in finding the highest performing versions. The actual t_{total} is $0.57ms$ in both cases which is 2.4 times faster than the fully parallel version.

Step 8. The buffer size for $p = 14$ is $9.2kbit$ and the storage size is $79.4kB$. Since all versions with fast reconfiguration meet the application requirement of $t_{total} < 4.615ms$, we could also choose the smallest version with $p = 4$. The buffer size for this version is $3.3kbit$ and the storage size is $28.9kB$. All versions of the filter fit into the device which provides $A_{max} = 2688$ CLBs.

Figure 4 illustrates the performance of our filters for different levels of parallelism based on the two different configuration speeds. We observe that slow configuration speeds lead to more serial implementations. Slow processing carries almost no weight compared to penalty of reconfiguration time. For faster configuration speeds, the optimum is a solution with an intermediate degree of

Table 5. Parameters of the FIR filter with different degrees of parallelism p when implemented on a Xilinx Virtex-4 LX25 FPGA

						slow reconfiguration: HWICAP [11]			fast reconfiguration: ICAP by Claus et.al. [12]		
p	slices	size $[kB]$	f_{max} [MHz]	$t_{p,e}$ [ns]	t_p [ns]	t_r [ms]	$t_{r,e}$ [ms]	t_{total} [ms]	t_r $[ms]$	$t_{r,e}$ $[ms]$	t_{total} [ms]
112	6921	404.1	250	4.0	4.00	80.8	0.72	80.9	1.347	0.012	1.39
56	3719	216.5	251	3.99	7.97	43.3	0.77	43.4	0.722	0.013	0.80
28	2111	122.7	256	3.91	15.63	24.5	0.88	24.7	0.409	0.015	0.57
14	1326	79.4	260	3.85	30.77	15.9	1.13	16.2	0.265	0.019	0.57
8	873	50.5	264	3.79	53.03	10.1	1.26	10.6	0.168	0.021	0.70
4	462	28.9	263	3.80	106.56	5.8	1.44	6.8	0.096	0.024	1.16

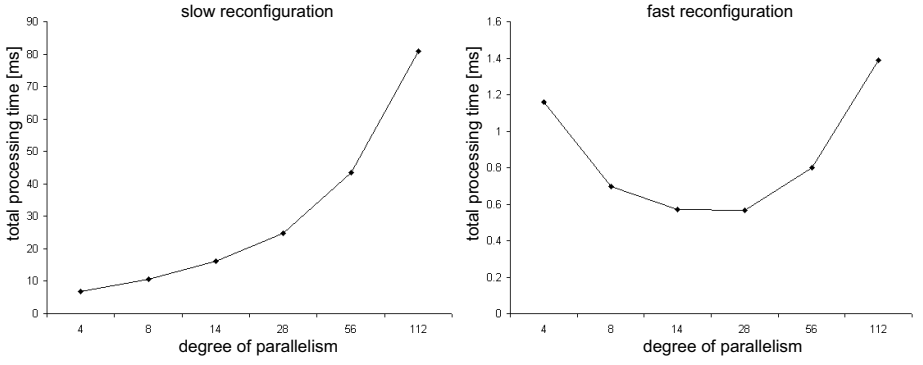

Fig. 4. Total processing time of our reconfigurable FIR filter when processing 10,000 data items between reconfigurations

parallelism. Increasing configuration speeds shift the optimal solution to more parallel implementations.

6 Conclusion and Future Work

This paper introduces a quantitative approach for developing run-time reconfigurable designs for software-defined radio applications. This approach is based on a simple model relating application-oriented parameters such as desired throughput, and implementation attributes such as area, processing time and reconfiguration time. It enables systematic development and optimisation of designs, which is illustrated in a case study involving a channelisation filter for GSM. Current and future work includes extending our approach to support a large variety of software-defined radio applications, and developing tools that automate our approach.

Acknowledgement. The support of Xilinx and UK EPSRC is gratefully acknowledged.

References

1. Compton, K., Hauck, S.: Reconfigurable computing: a survey of systems and software. ACM Computing Surveys 34(2), 171–210 (2002)
2. Cummings, M., Haruyama, S.: FPGA in the software radio. Communications Magazine 37(2), 108–112 (1999)
3. Uhm, M.: Software-defined radio: The new architectural paradigm. Xilinx DSP Magazin, 40–42 (October 2005)
4. Mitola, J.: The software radio architecture. Communications Magazine 33(5), 26–38 (1995)
5. Tuttlebee, W.: Software Defined Radio: Enabling Technologies. Wiley, Chichester (2002)
6. Keller, E.: Dynamic circuit specialization of a CORDIC processor. In: Reconfigurable Technology: FPGAs for Computing and Applications II. SPIE. The International Society for Optical Engineering, vol. 4212, pp. 134–141 (2000)
7. Tessier, R., Swaminathan, S., Ramaswamy, R., Goeckel, D., Burleson, W.: A reconfigurable, power-efficient adaptive Viterbi decoder. IEEE Transactions on VLSI Systems 13(4), 484–488 (2005)
8. Xilinx Inc. Virtex-5 Family Platfrom Overview LX and LXT Platforms v2.2 (January 2007)
9. Seng, S., Luk, W., Cheung, P.Y.K.: Run-time adaptive flexible instruction processors. In: Glesner, M., Zipf, P., Renovell, M. (eds.) FPL 2002. LNCS, vol. 2438, pp. 545–555. Springer, Heidelberg (2002)
10. Wirthlin, M., Hutchings, B.: Improving functional density using run-time circuit reconfiguration. IEEE Transactions on VLSI Systems 6(2), 247–256 (1998)
11. Xilinx Inc. Xilinx Logic Core: OPB HWICAP v1.3 (March 2004)
12. Claus, C., Zhang, B., Stechele, W., Braun, L., Hübner, M., Becker, J.: A multiplatform controller allowing for maximum dynamic partial reconfiguration throughput. In: Field Programmable Logic and Applications, pp. 535–538. IEEE, Los Alamitos (2008)