SACJ 31(2) December 2019
Research Article

Improving functional density of time-critical applications using hardware-based dynamic reconfiguration and bitstream specialisation

Rikus le Roux, George van Schoor, Pieter van Vuuren

School of Electrical, Electronic and Computer Engineering, North-West University, Potchefstroom, South-Africa

ABSTRACT
The dynamic reconfiguration of an FPGA has many advantages, but the overhead from the process reduces the functional density of applications. Functional density is an indication of the composite benefits a reconfigured application obtains above its generic counterpart and measures the computational throughput per unit hardware resources. Typically, only quasi-static applications obtain a functional density advantage by dynamically reconfiguring its parameters. Contributing to the functional density reduction of applications with tight time constraints is the overhead to generate a new configuration, and the time it takes to load it onto the device. Normally these applications have to reuse their hardware numerous times between configurations before obtaining a functional density advantage. The most promising reconfiguration method to improve functional density with minimal hardware reuse was one that extracts certain characteristics from the bitstream and then implements a bitstream specialiser that generates new hardware at bit-level while the device is being reconfigured. While it was shown that this method allows reconfiguration of an application in real-time, its effect on functional density was not determined. This paper will show that a significant increase in functional density can be achieved for applications where reconfiguration is required before the next execution cycle of the application.

Keywords: Reconfiguration, functional density, direct bitstream manipulation, bitstream specialisation

Categories: • **Hardware** ∼ **Reconfigurable logic and FPGAs** • *Computer systems organization* ∼ *Other architectures*

Email:
Rikus le Roux rikuslr@gmail.com (CORRESPONDING),
George van Schoor george.vanschoor@nwu.ac.za,
Pieter van Vuuren pieter.vanvuuren@nwu.ac.za

Article history:
Received: 14 Jun 2019
Accepted: 18 Oct 2019
Available online: 20 Dec 2019

1 INTRODUCTION

Traditionally, the reconfiguration of an FPGA is not only resource intensive, but the overhead from this process significantly reduces the functional density. This is especially prominent when the application has an extremely short execution time and its hardware cannot be reused between reconfigurations. The two contributors to the reconfiguration overhead are the time it takes to generate new hardware

Le Roux, R.R., Van Schoor, G. and Van Vuuren, P.A. (2019). Improving functional density of time-critical applications using hardware-based dynamic reconfiguration and bitstream specialisation. *South African Computer Journal 31*(2), 162–177. https://doi.org/10.18489/sacj.v31i2.786

SACJ is a publication of the South African Institute of Computer Scientists and Information Technologists. ISSN 1015-7999 (print) ISSN 2313-7835 (online).

and the time it takes to transfer these new configurations from an external memory location to the device's configuration memory.

A promising method to reduce both parameters is to use hardware-controlled reconfiguration. This method involves generating the new hardware beforehand, storing the configuration data in the FPGA's block random access memory (BRAM), and using a hardware-based finite state machine to facilitate the reconfiguration process (le Roux, van Schoor, & van Vuuren, 2015). The problems with this approach are that the configurations need to be known beforehand, and since the FPGA's BRAM is extremely limited, only a small subset of configurations can be stored.

A balance between functional density and modularity can be achieved by bitstream specialisation, a term used by Bruneel (2011) who proposed different techniques to adapt the configuration of an FPGA using an additional layer of abstraction (Bruneel & Stroobandt, 2010; Heyse & Stroobandt, 2015). This layer is required since the proprietary nature of an FPGA's bitstream prevents direct bitstream manipulation. This unfortunately reduces functional density since it increases reconfiguration time or throughput. Le Roux, van Schoor and van Vuuren (2019) proposed a specialisation method for Xilinx® FPGAs that allows the bitstream to be specialized in real-time while it is being transferred to the FPGA's configuration logic. It is based on the BRAM-based reconfiguration architecture the researchers proposed in le Roux, van Schoor, and van Vuuren (2014, 2015), but with an added hardware specialiser implemented in the FPGA fabric. The authors have shown a significant decrease in reconfiguration time, which should improve functional density, but this was not investigated.

This paper aims to determine the functional density advantage of le Roux, van Schoor, and van Vuuren (2019)'s proposed reconfiguration method. It uses a distributed multiply-accumulate (MAC) as a baseline application, which is then reconfigured using five of the most common techniques for modifying the circuit. The functional density of each method is then calculated and compared. A Xilinx® Virtex®-5 XCVFX70T FPGA was used for implementation, since this is the same device le Roux et al. (2019) used to showcase their bitstream specialiser. In theory any Xilinx® FPGA can be used by applying the method proposed by le Roux et al. (2019), but the FPGA architectures from other vendors were not analysed.

The paper begins by discussing functional density in Section 2. Thereafter an overview of the bitstream specialisation process is given in Section 3. It then continues to illustrate the functionality and advantages of the specialiser by reconfiguring a distributed multiply accumulate (MAC) in Section 4. This reconfiguration method is then compared to the most common reconfiguration methods, and their implementation discussed in Section 5. The results of this reconfiguration process are then given in Section 6 and discussed in Section 7. The paper is concluded in Section 8.

2 OVERVIEW OF FUNCTIONAL DENSITY

Functional density (D) (Wirthlin & Hutchings, 1998) was first proposed to measure the composite benefits of dynamic reconfiguration above its static generic counterpart, and measures the computational throughput (in operations per second) per unit hardware resources. For the reconfigurable case, this is defined by Equation 1:

https://doi.org/10.18489/sacj.v31i2.786

$$D_r = \frac{1}{A_r(T_{r,exec} + T_{reconf})} \tag{1}$$

where D_r denotes the reconfiguration functional density, A_r the size of the area to be reconfigured, $T_{r,exec}$ the execution time of the reconfigurable implementation, and T_{reconf} the reconfiguration time. If the hardware component can be used multiple times before reconfiguration is required, T_{reconf} can be amortized over several executions, n, thus increasing the functional density, as seen in Equation 2:

$$D_r = \frac{1}{A_r(T_{r,exec} + \frac{T_{reconf}}{n})}. \tag{2}$$

There are thus four ways functional density can be improved:

1. Optimizing the execution time of the application
2. Reducing the area that needs to be reconfigured
3. Minimizing the reconfiguration time
4. Reusing the hardware multiple times between reconfigurations

The highest functional density is obtained if the circuit is static (i.e. not reconfigured) and optimized for a specific application. Traditionally, if modularity is required, multiple circuits are implemented in parallel and the output path determined according to certain parameters. This approach increases the area utilization (A_r) which reduces the functional density. Once the application is reconfigured, the area utilisation is reduced to the area utilised by the application and the logic required to facilitate the reconfiguration process. Unfortunately the reconfiguration time (T_{reconf}) further reduces the functional density.

3 OVERVIEW OF THE BITSTREAM SPECIALISATION PROCESS

The specialiser proposed by le Roux et al. (2019) utilises configuration strings derived from extracted bitstream characteristics. The general idea is that any lookup table (LUT) can be expressed as a truth table, with each line representing an individual Boolean function. The authors have shown that the 64-bit parameter used to initialise the LUT can be used in conjunction with the address lines to determine the output of the truth table. Once the required LUT primitive is identified, the associated Boolean expression can be derived by using the initialization parameter as an additional address line for its truth table. Using the BRAM-based hardware controlled reconfiguration architecture, this Boolean expression is evaluated using the configuration strings. The results are then injected into the bitstream at the required frame address while the base configuration is loaded from the memory and transferred to the device's configuration memory via the internal configuration access port (ICAP).

Figure 1 shows the top level block diagram of the specialiser. The specialisation process is triggered by a rising edge on *ENABLE*, whereafter a configuration string is produced on *ConfigString*,

https://doi.org/10.18489/sacj.v31i2.786

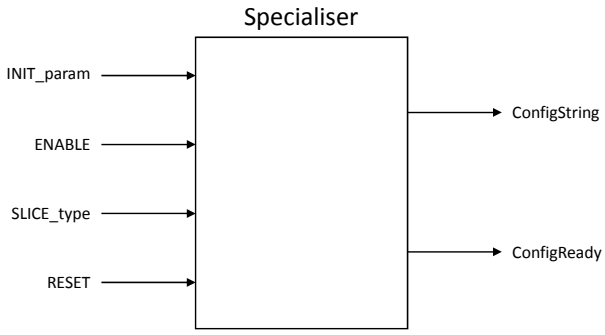


Figure 1: Block diagram of the top-level bitstream specialiser initialisation

based on the initialisation parameter (*INIT_param*) and LUT construct type (*SLICE_type*). Because of the asynchronous nature of the specialiser, the *ConfigReady* pin is used to establish handshaking with the rest of the hardware.

4 LOOKUP TABLE RECONFIGURATION

To illustrate the functionality and advantages of the specialiser, a fixed-point distributed MAC was selected and implemented, as shown in Figure 2. Distributed arithmetic performs multiplication operations using lookup table based schemes (Peled & Liu, 1974; White, 1989). It specifically targets the sum of products computation, predominately featuring in many important digital signal processing (DSP) filtering and frequency transformation functions. It can be used to implement a wide variety of applications, such as signal processing (Lu, Duan, Halak, & Kazmierski, 2019), filters (Kalaiyarasi & Reddy, 2019; Kumar, Shrivastava, Tiwari, & Mishra, 2019), control systems (Chan, Moallem, & Wang, 2004, 2007), system-on-chip (SoC) applications (Khawam, Arslan, & Westall, 2004) and discrete cosine transforms (Pan, Shams, & Bayoumi, 1999; Sowmya & Mathew, 2019; Yu & Swartzlander Jr., 2001). All these implementations utilise lookup tables to store certain aspects of the computation. This is of particular interest, because le Roux et al. (2019)'s bitstream specialiser specifically targets LUTs. This implies that the configuration of any of these designs can be specialised.

5 IMPLEMENTATION

Traditionally, if an application requires a MAC with different outputs, each datapath has to be implemented in parallel with the output being dependent on the path currently selected. This is referred to as a static application, because the hardware remains constant for each MAC output. Depending on the number of datapaths required, this implementation usually yields a high functional density, because no additional time is added to the execution time.

https://doi.org/10.18489/sacj.v31i2.786

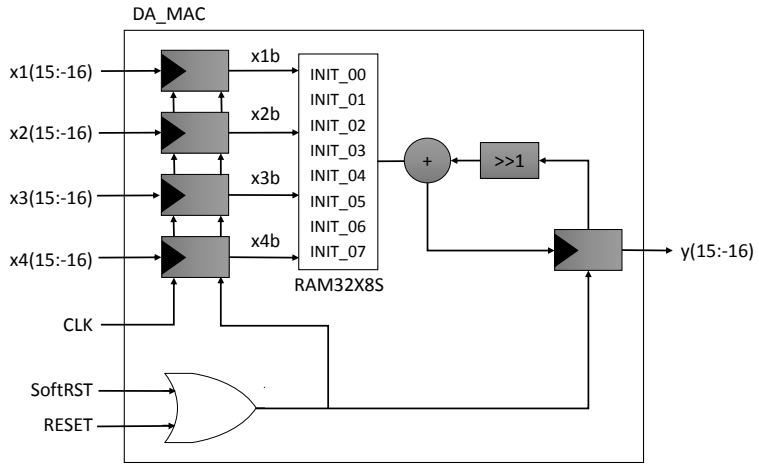


Figure 2: Block diagram representation of a multiply-accumulate implemented using distributed arithmetic

5.1 Base design: Static implementation

A MAC with nine possible datapaths was selected as the base design. Since it is not reconfigured, it is also referred to as a static, or generic, application. To cater for different outputs, each datapath has to be implemented in parallel, with a switch selecting the required path. This is shown in Figure 3. Even though many more datapaths could exist in practise, it is impossible to cater for each configuration. For this reason, only eight alternative paths were selected for comparison. However, keep in mind that some of the reconfiguration methods could allow for an indefinite number of configurations. The total area utilization of this application is 813 LUTs. This includes additional logic for the configuration selector and clock management.

5.2 Reconfiguration methods

Using this static implementation as a comparison base, the distributed arithmetic MAC shown in Figure 2 was implemented and reconfigured using different techniques to provide different outputs, and their functional densities compared. The reconfiguration methods used are:

1. **Configuration swapping with run-time FPGA toolflow:** Uses module-based reconfiguration to swap between configurations. In this implementation, the different configurations are stored in CompactFlash and the MAC reconfigured by swapping these configurations with the one currently on the device. This works fine for a small number of configurations, but for implementations with a larger set of configurations, these have to be generated at run-time using the Xilinx® tool flow. is also discussed for this implementation.

2. **Configuration swapping with software specialiser:** Generating new configurations at run-time adds significant overhead to the reconfiguration process. To mitigate this, the specialiser shown in Section 3 was implemented in software and added to the aforementioned configuration

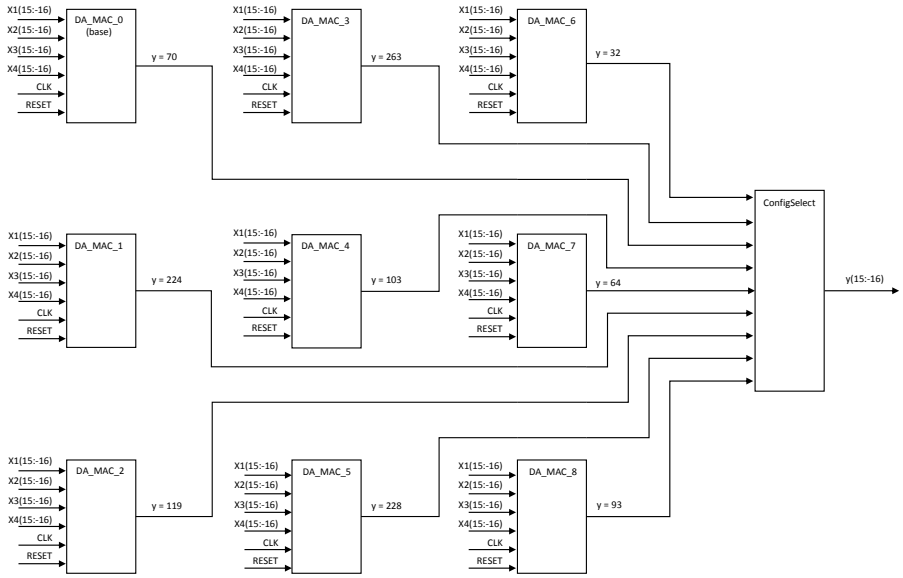


Figure 3: Architecture of the parallel static multiply-accumulate

swapping method. In this instance, instead of using the FPGA toolflow during run-time to generate new configurations, the software specialiser is used.

3. **CLB bit toggle:** This implementation uses a Xilinx® reconfiguration method by allowing direct manipulation of bits inside a CLB using an embedded processor. Like the previous implementation, this reduces configuration overhead by not having to transfer an entire bitstream to the configuration memory.

4. **Shift registers:** In this implementation the shift register functionality of modern Xilinx® FPGA's LUTs are used to adapt their configuration. In this setup, the configuration of a LUT can be changed by simply shifting a new value into the shift register lookup table (SRL).

5. **Hardware-based reconfiguration:** In this implementation, the bitstream specialiser of Section 3 is implemented in the FPGA fabric, along with the necessary circuitry to manage the reconfiguration process. Not only does this technique provide reconfiguration with the least amount of overhead, it also allows the reconfiguration logic to be clocked at frequencies above the 100 MHz recommended by Xilinx®.

To maintain uniformity between the different reconfiguration methods, each process was clocked at 100 MHz, except for the last design, which was also clocked at 200 and 300 MHz respectively to illustrate the increase in functional density at higher clock speeds. For designs with an embedded PowerPC® (such as the second reconfiguration method above), the processor bus was clocked at 200 MHz. These processors also have instruction and data cache built into the silicon of the hard processor, which, when enabled, provides a performance increase without compromising the area utilization or

https://doi.org/10.18489/sacj.v31i2.786

timing Fletcher, 2005. As such, the cache was enabled for each design with an embedded PowerPC® to provide a best case comparison.

5.2.1 Method 1: Configuration swapping with run-time FPGA toolflow

The first reconfigurable implementation assumes that the datapaths are, just like in the static implementation, known beforehand. The difference is that only one datapath is implemented at a time and in order to switch paths, the device has to be reconfigured. Therefore, all eight alternate configurations were generated beforehand and stored in external memory. When a different datapath is required, the specific section of the FPGA is reconfigured with the matching configuration. Two types of reconfiguration are available; module-based and difference-based (Eto, 2007; Kshirsagar & Sharma, 2011). The former configures an entire section of the device and requires larger configurations to be stored in memory. The latter reconfigures only the portion of the device that differs and therefore typically requires smaller configurations. An additional benefit is that a smaller configuration requires less time to be transferred from the external memory-space and will result in shortened reconfiguration time. For these reasons difference-based reconfiguration was selected to implement this first reconfiguration method.

The architecture used is shown in Figure 4 and requires a total of 2008 LUTs to implement this design. As can be seen, a PowerPC® 440 is used to facilitate the reconfiguration process. This is a hard-core processor embedded into the FPGA fabric, and uses no LUTs for its implementation. If a MicroBlaze™ soft-core processor, which is implemented using FPGA fabric, is used in lieu of the PowerPC® processor, an additional 600 LUTs are required. As such, the embedded PowerPC® was used for all implementations to reduce the area utilisation.

This implementation only works for a small number of configurations (because of the external memory requirements). This means that for applications requiring larger configuration sets, only a subset can be stored. The solution would then be to generate the required configurations at run-time using the conventional FPGA tool flow, however, this will either add a significant overhead in terms of time, or result in non-optimal configurations. For instance, running this process on an Intel® Core™ i7 M620 clocked at 2.67 GHz, with 4.00 GiB RAM and hyper-threading enabled in the toolset, requires an average of 521 seconds to generate the full configuration. If difference-based reconfiguration is used, this time can be reduced to 81 seconds, which is still significant for an application executing in real-time.

5.2.2 Method 2: Configuration swapping with software specialiser

The second reconfiguration method aims to alleviate this time-overhead by adding the bitstream specialiser proposed in Section 3 in software to the PowerPC®. As such, it uses a similar architecture shown in Figure 3, with the exception of the specialiser that is implemented in software.

5.2.3 Method 3: CLB bit toggle

Both the aforementioned reconfiguration implementations require a configuration to be transferred across a bus from the external memory-space to the configuration memory. As such, both these

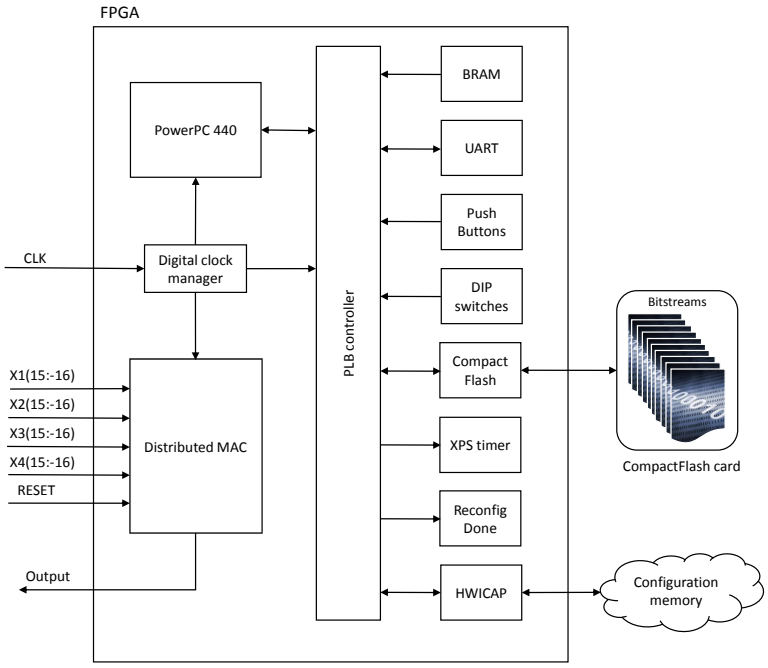


Figure 4: Architecture of the distributed MAC reconfigured using configuration swapping

implementations have additional overhead. In order to mitigate this, the Xilinx® predefined functions *XHwIcap_SetClbBits* and *XHwIcap_GetClbBits* were used to toggle the configuration of the LUTS. The result is an extremely fast adaptation of the LUT configuration, but this method is only capable of targeting specific LUTs. The resulting architecture requires 1562 LUTs to implement and is shown in Figure 5.

5.2.4 Method 4: Shift registers

A similar approach to toggling the CLB-bits, is using shift registers to change the configuration of the LUTs (Davidson, Abouelella, Bruneel, & Stroobandt, 2012; Glette, Torresen, & Hovin, 2009; Heyse, Farisi, Bruneel, & Stroobandt, 2012). In this architecture, the configuration bits are arranged as a shift register and shifted into the SRL as a new truth table. The result is a significant improvement in reconfiguration time with no unnecessary overhead. An added benefit is that this reconfiguration method also allows multiple LUTs to be reconfigured in parallel.

5.2.5 Method 5: Hardware-based reconfiguration

The final implementation utilises the hardware controlled reconfiguration controller discussed in Section 3 to facilitate the reconfiguration process. This controller is implemented in the FPGA fabric to form the high-level architecture shown in Figure 6. When new MAC constants are required, the specialisation process is triggered and new LUT values calculated. These new values are then sent

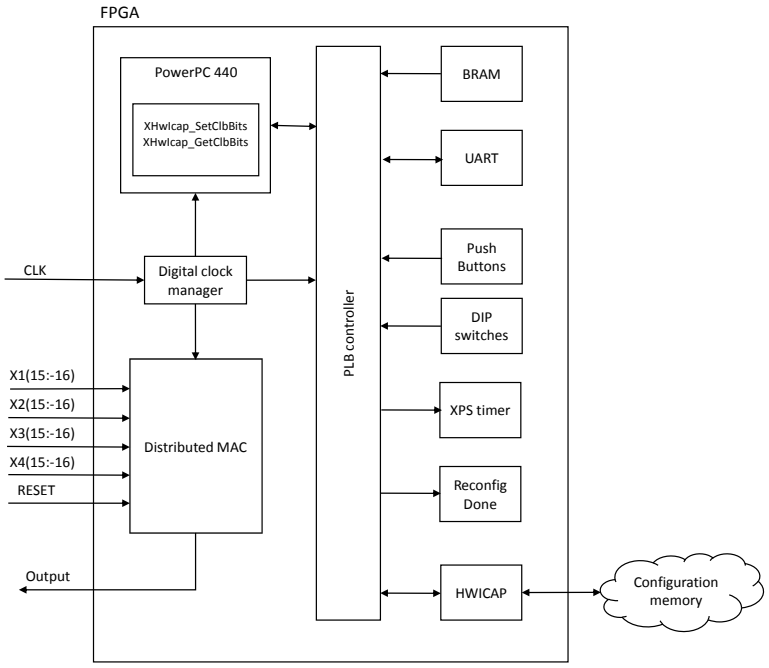


Figure 5: Architecture of the distributed MAC reconfigured using Set/GetClbBits

to the *Reconfiguration controller*, which is responsible for injecting these values into the bitstream read from the block RAM, and transferring it to the configuration memory via the ICAP. The primary advantage of this architecture is that it is not only possible to reconfigure with the least amount of overhead, but it is also possible to generate configurations for any number of constants.

6 RESULTS

The MAC was implemented on a Xilinx® ML507 development board (Xilinx Inc., 2011), whereafter the specialisation time (i.e. the time it takes to generate new hardware) and the reconfiguration time (i.e. the time it takes to load the new hardware on the device) were measured using an oscilloscope. The output of each reconfiguration process was verified using an "if"-statement to compare the predicted output with the actual result. If the two values match, an LED on the development board is illuminated. Conversely, the LED is off when the values are mismatched. Because the reconfiguration process temporarily causes these two values to differ, the reconfiguration time can be measured from the time the active LED turns off and back on again. The specialisation response was measured by illuminating a second LED once the reconfiguration process is triggered and turning it off once a new configuration is ready and being transferred to the device.

The rigidity of the timing results was determined using a command line batch tool, or the PowerPC® XPS timer. In some cases the reconfiguration time can also be calculated by using the ICAP clock frequency and the size of the configuration time. The results are given in Figures 7a to 7d for

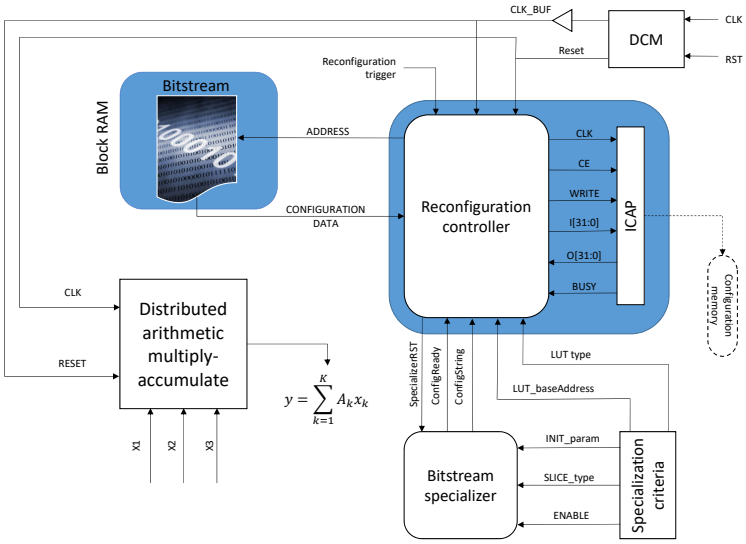


Figure 6: Architecture of the MAC reconfigured with hardware-based reconfiguration

the conventional reconfiguration methods, and Figures 7e and 7f for the proposed hardware-based reconfiguration. Using this information the functional density of each of the designs can be calculated and compared to the static design.

The execution time of the static design can be calculated from the number of clock cycles it takes to complete a single MAC-instruction and the clock period. In this instance, 35 clock cycles are required to process one instruction. Clocking the application at 100 MHz yields an execution time of 350 ns. Because the static design requires 813 LUTs to be implemented, the functional density can be calculated using Equation 3, with T_e the execution time of the application and A the utilized area:

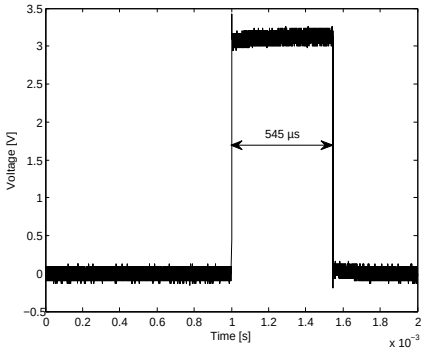
$$D_{static} = \frac{1}{AT_e} = \frac{1}{813(350 \times 10^{-9})} = 3514\,\text{operations/s} \tag{3}$$

Table 1 shows a summary of the functional density for each implementation, and Figure 8 illustrates the functional density of each design as a function of the average number of executions between hardware changes.

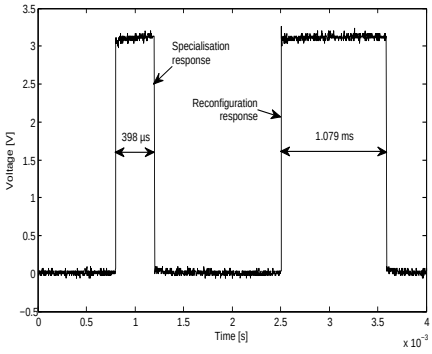
7 DISCUSSION

The static implementation with multiple datapaths yields the best functional density, because no additional overhead is required to switch between paths. However, this comes at the cost of an increase in area utilisation to accommodate each parallel datapath.

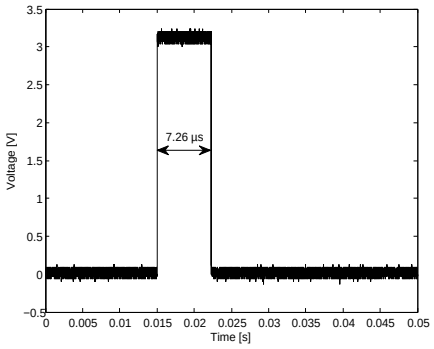
When an embedded PowerPC® is added to facilitate the reconfiguration process, the functional density is immediately reduced by the significant amount of resources required to implement these designs. Despite having decent configuration times, the time required to generate new hardware



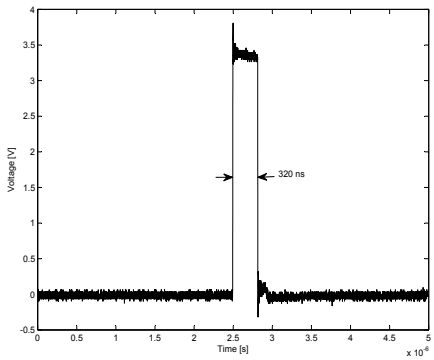
(a) Reconfiguration response of the configuration swapped design.



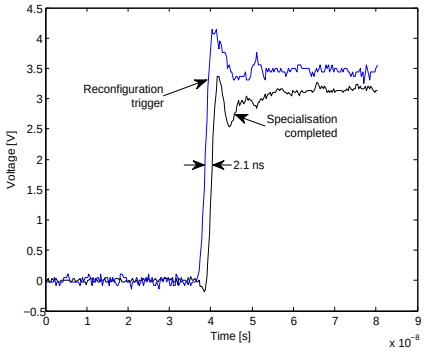
(b) Specialisation and reconfiguration response of the configuration swapped design with specialiser.



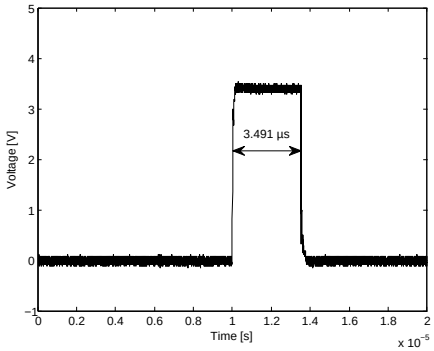
(c) Reconfiguration response of the CLB bit toggle functions.



(d) Reconfiguration response of the SRL reconfiguration method.



(e) Specialisation response of the hardware-based reconfiguration.



(f) Reconfiguration response of the hardware-based reconfiguration.

Figure 7: Oscillograms of the measured specialisation and/or reconfiguration response of the different reconfiguration methods

https://doi.org/10.18489/sacj.v31i2.786

Table 1: Functional density for each of the designs

Nr.	Implementation	Area (A) [#LUTs]	Execution time (T_e)	Generation time (T_{gen})	Configuration time (T_{conf})	Functional density (D)[1]
	Generic (Static) multiply-accumulate	813	350 ns	0	0	3,514.00
1	Configuration swapping (run-time)	1,685	350 ns	80.6 s	544.97 µs	7.36×10^{-6}
2	Configuration swapping (with software specialiser)	1,589	350 ns	398.00 µs	1.08 ms	425.86×10^{-3}
3	CLB bit toggle reconfiguration	1,562	350 ns	78.22 s	7.26 ms	8.18×10^{-6}
4	SRL reconfiguration	177	350 ns	21.16 s	320 ns	267.00×10^{-6}
5	Dynamic reconfiguration (100 MHz)	362	350 ns	2.10 ns	3.49 µs	718.73
6.1	Dynamic reconfiguration (200 MHz)	362	350 ns	2.10 ns	1.76 µs	1,311.00
6.2	Dynamic reconfiguration (300 MHz)	362	350 ns	2.10 ns	1.17 µs	1,814.00

[1] In operations per second per unit hardware resources

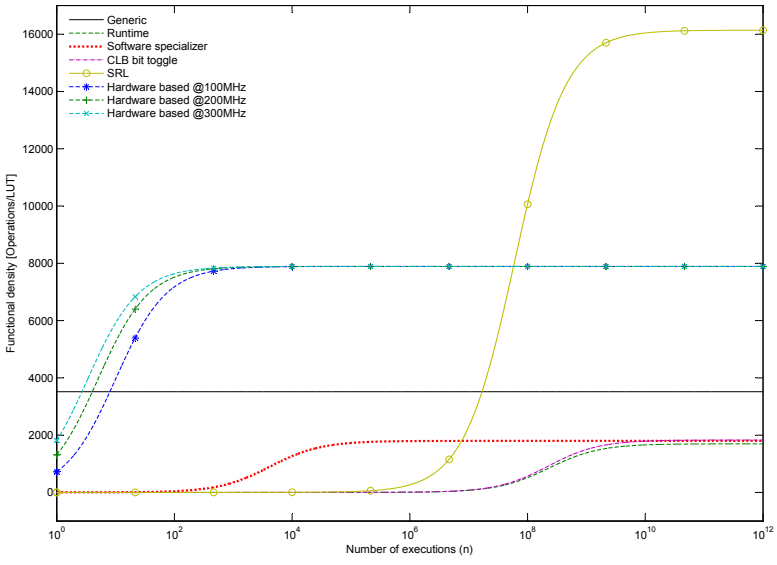


Figure 8: Illustration of functional density as a function of the number of executions

https://doi.org/10.18489/sacj.v31i2.786

for these implementations is, in most cases, significantly longer. The only exception is when configuration swapping is used in conjunction with the software specialiser. However, this is still significantly less than the functional density of the static design and the dynamic reconfiguration implementations. In this latter case, both the generation and configuration times are comparable to the static implementation. Unfortunately, the overhead induced by the reconfiguration process still results in a lower functional density than their static counterpart.

The main reason behind this is because the hardware has to be reused a minimum number of times before reconfiguration becomes feasible. By reusing the hardware multiple times, the overhead of the reconfiguration is amortised over each iteration and a break-even point is reached. This is defined by Equation 4, with T_{gen} the time to generate new hardware, T_{conf} the time to configure the device, $T_{r,exec}$ the execution time of the application, $T_{s,exec}$ the execution time of the static implementation, A_s the static area and A_r the reconfigurable area:

$$n = \frac{A_r(T_{conf} + T_{gen})}{A_s T_{s,exec} - A_r T_{r,exec}}. \tag{4}$$

As seen in Figure 8, none of the conventional reconfiguration methods are able to reach this break-point, because of the fast execution time of the MAC. The only conventional reconfiguration method capable of reaching the break-even point, and eventually exceeding the static implementation's functional density, is the SRL reconfiguration method. The overhead from the other conventional techniques is simply too large to improve the functional density.

Conversely, the hardware-based reconfiguration yields a significant improvement over the other reconfiguration techniques. Even though it does not obtain the same functional density as the SRL reconfiguration after multiple hardware reuse, it only requires the hardware to be changed once every eight execution cycles when clocked at 100 MHz to obtain a functional density advantage. This can be improved even further by clocking the design at higher frequencies. At 200 MHz only four execution cycles are required and at 300 MHz this is reduced to 2.7.

It is worth noting that each graph in the figure is relative. For this study, the MAC was bound by nine datapaths in the static implementation. To expand on the number of datapaths more hardware has to be added in parallel, which will increase the functional density, because of an increase in area. For example, if another nine datapaths are required, the area would increase to about 1,600 LUTs—halving the functional density. The same applies to the reconfigurable designs. In this particular study, only the constants of the MAC are reconfigured. If the reconfiguration processes are adapted to change the architecture of the MAC as well, the area would again influence the functional density of each design.

8 CONCLUSION

The work presented in this paper used the proposed hardware-based reconfiguration method of le Roux et al. (2019) to improve the functional density of an application that is typically not reconfigured. These applications in general have strict time constraints and short execution times, which give them

excellent functional densities. As a representative application, a distributed MAC was selected as a baseline, static application, which was then reconfigured using different techniques.

The results showed that the static application yielded the highest functional density, which was immediately reduced once the application was reconfigured. This can be attributed to the overhead involved in the process. This is particularly true for the configuration swapping methods that use the conventional Xilinx® toolflow, due to their long hardware specialisation and reconfiguration times. None of these methods ever reach a break-even point where the overhead is amortised over multiple executions.

Only the hardware based and SRL reconfiguration methods reach break-even points and exceed the functional density of the static application. Even though the SRL reconfiguration initially reduces the functional density significantly, this method eventually quadruples the functional density of the static application. However, this point is only reached if the hardware is reused 10^9 times between reconfigurations. Even though the hardware-based reconfiguration method does not obtain the same functional density advantage as the SRL reconfiguration, it improves on the break-even point by requiring hardware to be reused only eight times when clocked at the Xilinx® recommended 100 MHz. This result is significant since it is an indication that hardware-based reconfiguration could provide a functional density advantage to a real-time application.

The latest 7-series FPGAs from Xilinx® have a significantly higher performance than the Virtex-5 FPGA this study was based on. Since this will affect the functional density, le Roux et al.'s reconfiguration method and its effect on functional density for these devices need to be investigated.

9 ACKNOWLEDGEMENTS

This research was done under the Technology and Human Resources for Industry Programme (THRIP) and Oppenheimer Memorial Trust Grant (Ref. 19328/01).

References

Bruneel, K. (2011). *Efficient circuit specialization for dynamic reconfiguration of FPGAs* (Doctoral dissertation, PhD thesis, Ghent University). Retrieved from http://www.iwls.org/iwls2013/invited/bruneel.pdf

Bruneel, K., & Stroobandt, D. (2010). TROUTE: A reconfigurability-aware FPGA router. In P. Sirisuk, F. Morgan, T. El-Ghazawi, & H. Amano (Eds.), *Reconfigurable Computing: Architectures, Tools and Applications* (pp. 207–218). https://doi.org/10.1007/978-3-642-12133-3_20

Chan, Y. F., Moallem, M., & Wang, W. (2004). Efficient implementation of PID control algorithm using FPGA technology. In *2004 43rd IEEE Conference on Decision and Control (CDC) (IEEE Cat. No.04CH37601)* (Vol. 5, pp. 4885–4890). https://doi.org/10.1109/CDC.2004.1429572

Chan, Y. F., Moallem, M., & Wang, W. (2007). Design and implementation of modular FPGA-based PID controllers. *IEEE transactions on industrial electronics*, *54*(4), 1898–1906. https://doi.org/10.1109/TIE.2007.898283

Davidson, T., Abouelella, F., Bruneel, K., & Stroobandt, D. (2012). Dynamic circuit specialisation for key-based encryption algorithms and DNA alignment. *International journal of reconfigurable computing*, *2012*, 1–13. https://doi.org/10.1155/2012/716984

Eto, E. (2007). *Difference-based partial reconfiguration* (tech. rep. No. XAPP290 (v2.0)). Xilinx Inc.

Fletcher, B. H. (2005). FPGA embedded processors: Revealing true system performance. In *Embedded Systems Conference San Francisco 2005* (p. 18). ETP-367. Memec.

Glette, K., Torresen, J., & Hovin, M. (2009). Intermediate level FPGA reconfiguration for an online EHW pattern recognition system. In *2009 NASA/ESA Conference on Adaptive Hardware and Systems* (pp. 19–26). https://doi.org/10.1109/AHS.2009.46

Heyse, K., Farisi, B. A., Bruneel, K., & Stroobandt, D. (2012). Automating reconfiguration chain generation for SRL-based run-time reconfiguration. In O. C. S. Choy, R. C. C. Cheung, P. Athanas, & K. Sano (Eds.), *Reconfigurable Computing: Architectures, Tools and Applications* (pp. 1–12). https://doi.org/10.1007/978-3-642-28365-9_1

Heyse, K., & Stroobandt, D. (2015). Avoiding transitional effects in dynamic circuit specialisation on FPGAs. In *Proceedings of the 52nd Annual Design Automation Conference* (159:1–159:6). DAC '15. https://doi.org/10.1145/2744769.2744802

Kalaiyarasi, D., & Reddy, T. K. (2019). Design and implementation of least mean square adaptive FIR filter using offset binary coding based distributed arithmetic. *Microprocessors and microsystems*, 102884. https://doi.org/10.1016/j.micpro.2019.102884

Khawam, S., Arslan, T., & Westall, F. (2004). Synthesizable reconfigurable array targeting distributed arithmetic for system-on-chip applications. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* (pp. 150–158). https://doi.org/10.1109/IPDPS.2004.1303136

Kshirsagar, R. V., & Sharma, S. (2011). Difference based partial reconfiguration. *International journal of advances in engineering & technology*, *1*(2), 194–197.

Kumar, P., Shrivastava, P. C., Tiwari, M., & Mishra, G. R. (2019). High-throughput, area-efficient architecture of 2-D block FIR filter using distributed arithmetic algorithm. *Circuits, systems, and signal processing*, *38*(3), 1099–1113. https://doi.org/10.1007/s00034-018-0897-2

le Roux, R., van Schoor, G., & van Vuuren, P. (2014). A survey on reducing reconfiguration cost: reconfigurable PID control as a special case. *IFAC Proceedings Volumes*, *47*(3), 1320–1330. 19th IFAC World Congress. https://doi.org/10.3182/20140824-6-ZA-1003.01544

le Roux, R., van Schoor, G., & van Vuuren, P. (2015). Block RAM-based architecture for real-time reconfiguration using Xilinx® FPGAs. *South African computer journal*, *56*(1). https://doi.org/10.18489/sacj.v56i1.252

le Roux, R., van Schoor, G., & van Vuuren, P. (2019). Parsing and analysis of a Xilinx FPGA bitstream for generating new hardware by direct bit manipulation in real-time. *South African computer journal*. https://doi.org/10.18489/sacj.v31i1.620

Lu, Y., Duan, S., Halak, B., & Kazmierski, T. J. (2019). A cost-efficient error-resilient approach to distributed arithmetic for signal processing. *Microelectronics reliability*, *93*, 16–21. https://doi.org/10.1016/j.microrel.2018.12.007

Pan, W., Shams, A., & Bayoumi, M. A. (1999). NEDA: A new distributed arithmetic architecture and its application to one dimensional discrete cosine transform. In *1999 IEEE Workshop on Signal Processing Systems. SiPS 99. Design and Implementation (Cat. No.99TH8461)* (pp. 159–168). https://doi.org/10.1109/SIPS.1999.822321

Peled, A., & Liu, B. (1974). A new hardware realization of digital filters. *IEEE transactions on acoustics, speech, and signal processing*, *22*(6), 456–462. https://doi.org/10.1109/TASSP.1974.1162619

Sowmya, K. B., & Mathew, J. A. (2019). Adept-disseminated arithmetic-based discrete cosine transform. In A. Abraham, P. Dutta, J. K. Mandal, A. Bhattacharya, & S. Dutta (Eds.), *Emerging Technologies in Data Mining and Information Security* (pp. 379–384). https://doi.org/10.1007/978-981-13-1951-8_34

White, S. A. (1989). Applications of distributed arithmetic to digital signal processing: a tutorial review. *IEEE ASSP magazine*, *6*(3), 4–19. https://doi.org/10.1109/53.29648

Wirthlin, M. J., & Hutchings, B. L. (1998). Improving functional density using run-time circuit reconfiguration. *IEEE transactions on very large scale integration (VLSI) systems*, *6*(2), 247–256. https://doi.org/10.1109/92.678880

Xilinx Inc. (2011). *ML505/ML506?ML507 evaluation platform* (User guide No. XAPP347 (v3.1.2)). Xilinx Inc. Retrieved from https://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf

Yu, S., & Swartzlander Jr., E. E. (2001). DCT implementation with distributed arithmetic. *IEEE transactions on computers*, *50*(9), 985–991. https://doi.org/10.1109/12.954513