# HLS Compilation for CPU Interlays

Jose Raul Garcia Ordaz and Dirk Koch
School of Computer Science, The University of Manchester, United Kingdom
{raul.garcia, dirk.koch}@manchester.ac.uk

## ABSTRACT

The idea of coupling reconfigurable fabrics with general-purpose processors has been extensively studied during the last couple of decades. Custom instructions targeting those reconfigurable fabrics had to be handcrafted because tools capable of high level synthesis were not available at the time. Nowadays, high level synthesis tools have matured to a state allowing system designers to automatically generate hardware implementations from software applications.

At the end of Moore's era, it is required to reinvestigate reconfigurable custom instructions by taking full advantage of the latest HLS compilers. In this paper, we introduce the concept of CPU interlays which are FPGA-like fabrics that are integrated directly into the core of a hardened processor. This enables the customization of an instruction set at run-time. While CPU interlays will show best performance with hand-optimized custom instructions, this paper suggests a semi-automated flow that does not need the expertise of an FPGA designer. Using automatic profiling together with HLS tools allows the acceleration of user programs with very little human interaction during application design.

By replacing the NEON SIMD unit of an ARM Cortex-A9 with an interlay taking the same die area, we could demonstrate speedups as high as 68× for individual function kernels without touching any RTL code. Furthermore, we show that while HLS compilers can enhance design productivity, it is in some cases required to follow a HLS-friendly coding style for maximizing performance.

## 1. INTRODUCTION

Many CPU families have gained significant complexity (and corresponding real estate) throughout their many years of development. For instance, the ARMv5 ISA provided less than 40 instructions. Later, the ARMv6 ISA provided around 100 instructions including SIMD instructions such as USAD8. More recently, the ARMv7 ISA introduced a media instruction set extension (ISE) known as NEON. In total, the ARMv7 ISA including the NEON (ISE) provides around 250 instructions. The most recent ARM ISA (i.e. ARMv8) boasting almost 300 instructions, introduced a SIMD ISE for crypto applications [1].

However, instead of adding more and more complexity to a CPU core, which may have negative impact on energy efficiency and overall instruction throughput, *interlays* add a reconfigurable fabric inside an otherwise hardened CPU core. Interlays can be seen as customized FPGA fabrics to add just the reconfigurablility that is needed to gain substantial performance gains through customizing an instruction set. By configuring an interlay with an interlay bitstream, a customized instruction set can be adapted and used while the system is running. This approach follows the ideas first introduced in seminal work like in [2, 3].

Currently, the trend is to embed a processor SoC into a relatively large FPGA fabric (e.g., as done for Xilinx Zynq FPGAs). Instead of this, interlays are rather tiny and they are embedded inside a hardened CPU core. Furthermore, interlays are very tightly coupled to its CPU (allowing to call acceleration instructions directly in user mode). With this, interlays enable a path for building low cost and energy efficient systems in a software centric (and consequently faster) way while still providing a path for substantial acceleration through instruction set customization. We believe that in particular very heterogeneous applications domains (such as Internet of Things systems) will benefit from this approach.

Typically, hand-optimized interlay instructions will result in best acceleration, but this design path needs hardware design expertise and takes substantially longer to design. In this paper, we investigate a mostly automated compilation flow that exploits an interlay fabric entirely through a software design path using high-level synthesis (HLS).

This work was presented in part at the international symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2017) Bochum, DE, June 7-9, 2017.

## 2. RELATED WORK

In order to meet the design challenges of future SoC architectures which incorporate FPGA fabrics, tools for automatically compiling high level code to FPGAs have been developed. Some of these tools are based on *mapping data flow graphs* (DFG) into hardware. A recent example of design frameworks using the DFG-based approach is FISH [4], which can extract compute intensive subgraphs directly from C/C++ applications. These DFGs are then mapped into custom instructions. Similar approaches are discussed in [5–7].

Alternatively to the DFG-based approach, there is the *software-defined* design approach. Here, the framework's design flow allows an user to profile software applications and to select time-consuming functions aimed to be mapped to hardware. A high-level synthesis tool is then used to transform the function's algorithm described in C/C++ into an RTL equivalent. Finally, the existing application code is instrumented with function calls and a customized compiler is used to generate the executable. Examples of these software-defined tools include Legup from the University of Toronto [8] and Vivado HLS [9] developed by the vendor Xilinx. In both cases, the accelerators generated by each framework communicate to a CPU through a bus interface. In contrast, in this work we propose a design flow aimed to map custom instructions to a mixed-grained reconfigurable interlay fabric that directly accesses a register file. Such an interlay is integrated tighter into the CPU and allows calling accelerators at much lower latency than what is possible by a bus-coupled accelerator.

## 3. HLS-GENERATION OF CIS

### 3.1 Design Flow

The design flow of the framework proposed in this work is depicted in Figure 1. The description of the design flow is as follows: 1) First an application is developed in a high-level programming language such as C/C++. 2) A conventional compiler such as GCC is used to generate an executable of the application which is then executed on the Gem5 simulator. We leverage the built-in ability of Gem5 to generate program traces and statistics [10]. 3) The program trace is analysed to detect time-consuming functions (i.e. functions that consume most execution cycles). These kernels are selected as custom instruction candidates. 4) The Vivado HLS tool from the vendor Xilinx is used to synthesize the selected kernels into its RTL equivalents. Note that the use of an HLS tool eliminates the need for complex heuristics to generate DFGs or the need for an FPGA hardware expert in order to generate the custom instructions. 5) The area (FPGA primitives) and latency (execution cycles) reports produced by the Vivado HLS tool corresponding to each CI generated are analysed. Since we are aiming for a relatively small CPU interlay, we only select CIs that are below the area constrain. The reported CI's execution cycles are used to calculate the potential speedups that can be achieved by replacing the existing software function with the hardware candidate CI. Note that in general, a hardware implementation of a kernel consumes less execution cycles than its software counterpart. However, as the here proposed interlay fabric operates at a much slower clock frequency than the hardened CPU, this difference must be taken into account when calculating the execution time of the hardware CI. We incorporate this also in adapting the simulation model of the ARM CPU within the Gem5 simulator to precisely simulate the ARM CPU-interlay hybrid.

6) The generated CIs can be tuned to achieve a desired performance by modifying a) the structure of the existing software kernel in order to enable succesful hardware compilation and b) to use options provided by the HLS tool to perform hardware optimizations such as loop unrolling. Here it is also possible to limit the amount of DSPs used for the design which impacts the final distribution of LUTs and DSPs consumed by the CI.

7) Once the application-specific CIs are tuned as needed, they are saved into a library which will contain CIs for different applications. 8) The repository stores the interlay configuration and all metadata that is needed by the runtime system. 9) The custom instructions can then be called by the software application by instrumenting the existing source code with inline function calls. 10) Finally, the instrumented code will be compiled. The resulting executable will be able to then call the CIs configured to the CPU interlay.

### 3.2 Case Study 1

We will explain our approach in more detail using a practical example throughout the following paragraphs. We exemplary selected the ADPCM application benchmark and the AES application benchmark from the CHStone suite [11]. The CHStone suite is a collection of C-based application benchmarks specifically developed to be used to perform research in the area of high level synthesis (HLS). These applications represent signal processing tasks as well as compression which are quite common for IoT or mobile systems and that are quite compute heavy.

#### 3.2.1 Interlay Emulation

Normally, an interlay fabric (including its size, primitives and routing architecture) will be optimized for instruction set extensions. In this paper we follow the approach described in [12] that uses a commercial-off-the-shelf FPGA

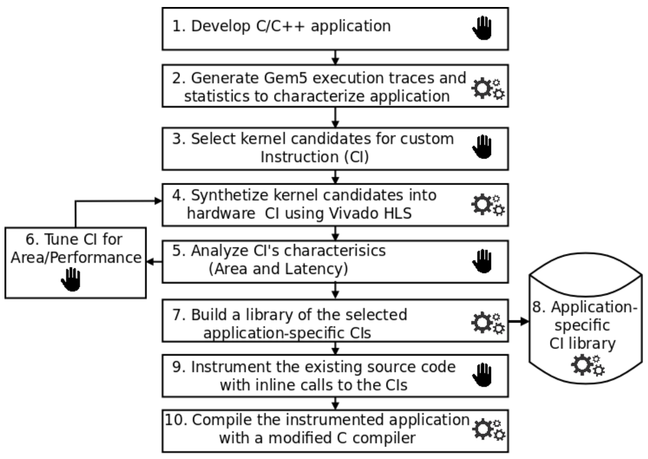
1. Develop C/C++ application
2. Generate Gem5 execution traces and statistics to characterize application
3. Select kernel candidates for custom Instruction (CI)
4. Synthetize kernel candidates into hardware CI using Vivado HLS
5. Analyze CI's characteristics (Area and Latency)
6. Tune CI for Area/Performance
7. Build a library of the selected application-specific CIs
8. Application-specific CI library
9. Instrument the existing source code with inline calls to the CIs
10. Compile the instrumented application with a modified C compiler

Figure 1: Interlay HLS design flow. The hand symbol represents a manual execution of the corresponding step. The gear symbol represents a mostly automatic execution of the corresponding step.

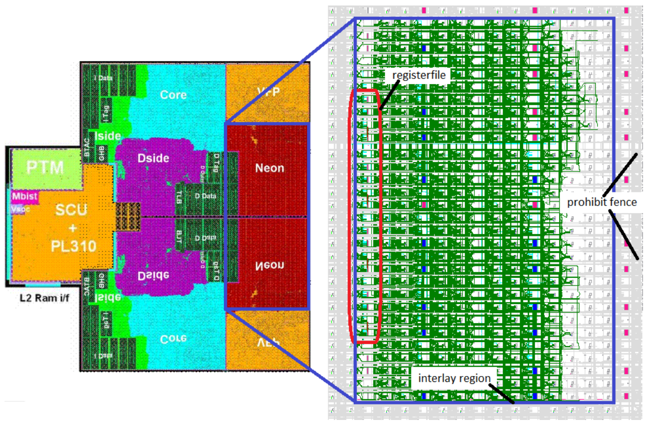

Figure 2: Emulated CPU interlay embedded as a NEON replacement in an ARM processor. The implemented interlay configuration implements the functions upzero, filtez, quantl, and uppol1 of the ADPCM example.

instead. This will result in weaker area and performance numbers that we will report here but allows using industrial tools and to experiment with real hardware. We chose an FPGA area providing 2400 6-input LUTs, 6 BRAMs, and 18 DSP blocks as our interlay. This FPGA region corresponds very closely to the die area that the NEON SIMD unit is occupying within the ARM SoC of a Cortex-A9 that is implemented on a Zynq SoC. We used 4 BRAMs as the assumed hardened register file providing two 128-bit vector operands and one 128-bit vector result (and a few more control signals). As shown in Figure 2, we used strict resource and routing bounding boxes to precisely emulate an interlay.

#### 3.2.2 Custom Instruction Selection

We used the Gem5 simulator to obtain profiling and execution traces of the ADPCM and the AES benchmarks. The Gem5 simulator was configured to simulate the characteristics of the ARM Cortex-A9 processor (which we have been using for our experiments) similarly as described in [13]. Figure 3 shows a segment of the function trace corresponding to the execution of the ADPCM benchmark. Similarly, Figure 4 shows a segment of the function trace corresponding to the execution of the AES benchmark. These figures help to visualize the amount of execution time of the different kernel functions.

The ADPCM application uses 15 functions whereof 7 ("encode", "decode", "upzero", "filtez", "uppol2", "quantl", "uppol1") contribute to 90% of the total execution of the application, as shown in Figure 3. These are suitable candidates to be mapped to the interlay as one set of *ADPCM-specific custom instructions*. Please note that one interlay may implement multiple different custom instructions in one configuration if permitted by the resources. Likewise, the AES application uses 11 functions with 5 candidate functions ("AddRoundKey_InversMixColumn", "Mix Column_AddRoundKey", "KeySchedule", "ByteSub_ Shift Row", and "InversShiftRow_ByteSub") that together represent 90% of the total AES execution time. In general, we follow a greedy approach that tries to boost acceleration by mapping the most compute intense functions first until we are running out of interlay resources.

If we compare the two traces in Figure 3 and Figure 4 with each other, we see that for the ADPCM application, we call rapidly a sequence of different functions, while for the AES application, we have two compute intensive kernels alternating with relatively low periodicity. This means that for the ADPCM case, run-time reconfiguration is basically infeasible and we can only map the most benefiting functions to one interlay configuration. Opposed to this, the situation is different for the AES case and depending on how long each of the function bursts lasts, reconfiguration may be beneficial. For example in Figure 4, we could map the AES "KeySchedule" function in one interlay configuration and the "MixColumn_AddRoundKey" function into another interlay configuration. A further observation is that there are other functions called between these bursts, which can be used to hide some of the configuration latency (e.g., by using configuration prefetching [14]).

In general, run-time reconfiguration is beneficial if the execution time of the hardware accelerated program plus the configuration time is less than the original software execution time ($t_{hw} + t_{config} \leq t_{sw}$). The actual reconfiguration time depends on the system (e.g., how it can deal with the extra configuration data burst) and the actual application (e.g., if configuration prefetching is applicable).

The assumed interlay as shown in Figure 2 equates to a bitstream that is about 196KB in size. This would take at least 49K cycles at a 32 bit wide configuration port for reconfiguration. Considering that the configuration port on a Zynq FPGA runs 6.5 times slower than the CPU, a full interlay reconfiguration process will correspond to about 320K CPU cycles. Therefore, a custom instruction has to save this number of cycles over the original CPU implementation before a gain can be achieved. However, the very slow configuration speed and the large configuration bitstream size will very likely be substantially smaller when building a real interlay. For example, in [15] a configuration speed of 2.2GB/sec @550MHz was demonstrated on a Virtex-5 device. This would correspond to 58K CPU cycles for one interlay configuration process.

### 3.2.3 Characteristics of Custom Instructions

We took the candidate software functions previously selected from the ADPCM and AES application benchmark, and used the Vivado HLS tool to generate RTL functional units that perform the same tasks as those kernels. We only consider custom instructions that fit into the number of resources that our assumed interlay fabric provides. Because we aim to cause minimal architectural disruptions when integrating the fabric into the considered hardened ARM processor, the remaining functional units that we implement as custom instructions should have the same 2-vector input, 1-vector output interface as the existing ARM instructions.

We define the gain of a custom instruction ($CI_{gain}$) as the overall execution time improvement. With $t_r$ being the relative portion from the overall execution time and $s$ being the speedup, this is:

$$CI_{gain} = \frac{1}{(1 - t_r) + \frac{t_r}{s}}$$

And if an interlay configuration implements k instructions:

$$int_{gain} = \frac{1}{(1 - \sum_k t_{r,k}) + \sum_k \frac{t_{r,k}}{s}}$$

The gain is, like we know from Amdahl's law, bound by how much a custom instruction contributes to the overall problem. In general, the custom instruction selection process has to consider 1) resource constraints, 2) speedups and gains, and optionally 3) reconfiguration times. The complexity of this optimization problem is more complex as HLS tools allow generating different resource/speedup trade-offs. Also when implementing more than one custom instruction in one interlay configuration, some resources maybe shared. As a first approach, we have automated the selection process by implementing a brute force algorithm in Python script that finds the combination of custom instructions (both optimized for area and optimized for performance) that *yield the maximum accumulated interlay gain and which accumulated resource utilization is below the interlay resource constrains* (see Algorithm 1).

**Algorithm 1** Maximum Interlay Gain

```
1: procedure MAX_INT_GAIN($ci\_list[0 : n - 1]$ )    ▷ List of custom instructions
2:     $combinations\_list \leftarrow get\_combinations(ci\_list)$
3:     for each combination in combinations_list do
4:         $acc\_res \leftarrow calculate\_accumulated\_resources$
5:         if acc_res <= int_res then
6:             $int\_gain \leftarrow calculate\_accumulated\_gain$
7:             if int_gain > tmp_max then
8:                 $tmp\_max \leftarrow int\_gain$    ▷ Store temporary maximum
9:                 $tmp\_comb \leftarrow combination$    ▷ Store combination index
10:            end if
11:        end if
12:    end for
13:    $max\_int\_gain \leftarrow tmp\_gain$
14:    $comb \leftarrow tmp\_combination$
15:    return ($max\_int\_gain, comb$)    ▷ Return the maximum interlay gain and its associated CI combination
16: end procedure
```

The results of our experiments are presented in Table 1 and Table 2 for the ADPCM and the AES function, respectively. The tables show the area (expressed in FPGA primitives) used by the candidate kernel functions. A check symbol is used to mark the candidate CIs that fit into the CPU interlay as reported by the Vivado HLS tool. Additionally, we provide potential speedup numbers derived from the execution time for each CI as reported by the Vivado HLS tool and the numbers provided by the Gem5 simulator. Finally, we present the relative execution contribution to the overall application execution time corresponding to each candidate CI.

Table 1: Characteristics of the HLS-Generated ADPCM custom instructions. Two versions are synthesized: one optimized for area (OA), and 2) one optimized for performance (OP).

| CI | Primitives | | | | | | Speedup | | Exec |
|---|---|---|---|---|---|---|---|---|---|
| | OA | | | OP | | | OA | OP | % |
| | LUT | DSP | Fit | LUT | DSP | Fit | | | |
| upzero | 337 | 4 | ✓ | 1087 | 24 | ✗ | 1.4 | 3.3 | 19.1 |
| filtez | 281 | 8 | ✓ | 143 | 24 | ✗ | 1.1 | 3.8 | 10.4 |
| uppol2 | 217 | 8 | ✓ | 300 | 8 | ✓ | 2.8 | 3.1 | 4.3 |
| quantl | 108 | 2 | ✓ | 1097 | 54 | ✗ | 1.5 | 14.6 | 4.2 |
| uppol1 | 234 | 4 | ✓ | 296 | 4 | ✓ | 2.2 | 2.9 | 3.4 |

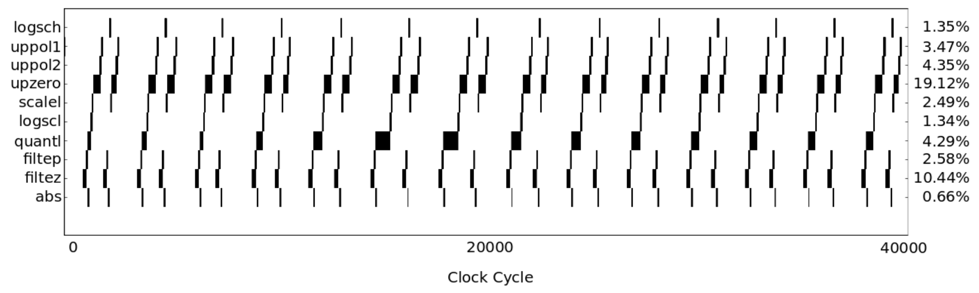

Figure 3: Segment of the ADPCM application function trace. The percentage values state the relative execution time per function.

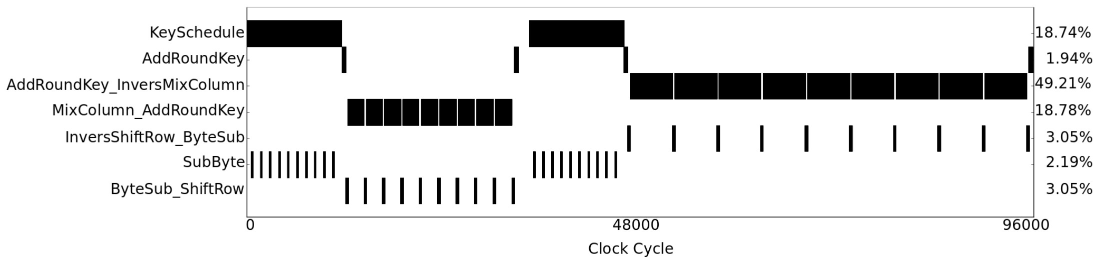

Figure 4: Segment of the AES application function trace. The percentage values state the relative execution time per function.

Table 2: Characteristics of the HLS-Generated AES custom instructions. Two versions are synthesized: one optimized for area (OA), and 2) one optimized for performance (OP).

| CI | Primitives | | | | | | Speedup | | Exec |
|---|---|---|---|---|---|---|---|---|---|
| | OA | | | OP | | | OA | OP | % |
| | LUT | DSP | Fit | LUT | DSP | Fit | | | |
| AddRound Key_Invers MixColumn | 2276 | 4 | ✓ | 4971 | 4 | ✗ | 8.9 | 18.1 | 49.2 |
| MixColumn_ AddRound Key | 2156 | 4 | ✓ | 2156 | 4 | ✓ | 10.3 | 10.3 | 18.8 |
| KeySchedule | 2310 | 1 | ✓ | 4284 | 1 | ✗ | 3.9 | 4.6 | 18.7 |
| ByteSub_ ShiftRow | 1291 | 0 | ✓ | 1291 | 0 | ✓ | 20.4 | 20.4 | 3.1 |
| Invers ShiftRow_ ByteSub | 1197 | 0 | ✓ | 1197 | 0 | ✓ | 20.4 | 20.4 | 3.1 |

#### 3.2.4 *Discussion*

The total speedup gains reported in [8] (i.e. the ratio between the execution time of the software implementation and the pure hardware Legup implementation) for the ADPCM and the AES applications are 3.2× and 4.2×, respectively. While these speedup gains are substantial, they come at the cost of a relatively large area consumption. The reported hardware Legup implementation of the ADPCM application is 22605 LUTs. Similarly, the hardware Legup implementation of the AES application is 28490 LUTs. Considering that the softcore used to run the pure software implementation of those applications is 12243 LUTs, implementing these applications with the Legup tool represents an area overhead of 1.8 and 2.32 for the ADPCM and the AES applications, respectively. In contrast, with our interlay approach, we obtain more modest total speedup gains at a much lower area cost. As an example we take the most time consuming function kernel that fits in the interlay (i.e. "upzero" for the ADPCM application and "AddRoundKey_InversMixColumn" for the AES application). In this case, we have area ratios (considering the 2400 LUT contraint) of 0.45 and 0.94, for the "upzero"" and the "AddRoundKey_InversMixColumn" function kernels, respectively. With this area consumption the ADPCM application can achieve a $CI_{gain}$ of 1.2×. Note that the "upzero" function optimized for area consumes only 45% of the interlay resources. This makes it possible to implement more lightweight ADPCM-specific CIs in the same CPU interlay configuration to achive even more performance gains and avoid interlay resource waste. In this case, it would be possible to feed the list of HLS-generated CIs (and its associated area and speedup characteristics) to our Python script to find the combination that achieves the maximum interlay gain. According to our results, implementing the "upzero", "filtez", "quantl", and "uppol1" CIs in their versions optimized for area would yield a maximum interlay gain of 1.45×. Similarly, the AES application can achieve a $CI_{gain}$ of 1.7×. Note that the "AddRoundKey_InversMixColumn" consumes 94% of the interlay resources, restricting the option to implement more CIs in the same interlay configuration. In this case, run-time reconfiguration could be used to enhance the performance of the CPU interlay.

### 3.3 Case Study 2

It has been demonstrated that applications that heavily depend on bitwise operations can greatly benefit from the implementation of advanced bit-manipulation custom instructions [16]. The reason for this is that in order to perform some complex bitwise operations, long sequences of instructions (e.g. load/store, basic boolean, shift instructions etc.) are required on general purpose CPUs. In some cases, a relatively straightforward but bitwise-intensive algorithm implemented in C code can easily be translated into several lines of assembly instructions. In contrast, reconfigurable fabrics, such as FPGAs, execute bitwise operations much more efficiently. This is because on FPGAs, bitwise operations can be performed in a more direct way as input operands are treated as bit vectors that can be easily accessed individually. Additionally, the architecture of FPGAs allow for a highly parallel execution of bitwise operations.

### *3.3.1 CRC Algorithm*

Consider the CRC algorithm, which is used extensively by storage and network devices to detect errors in digital data [17]. As the block diagram in Figure 5 shows, the CRC algorithm heavily depends on the XOR operation to generate the CRC value. In this case, load-eor-store instruction sequences are executed by the CPU (other instructions such as addition and shift are also required). As the instruction trace presented in Figure 6 shows, the lack of a CRC-specific custom instruction on general purpose CPUs translates into a sub-optimal computation of CRC values. This is normally mitigated by using different optimizing software techniques to force the compiler to compute CRC values faster [18]. Alternatively, a CRC custom instruction could be implemented by the here proposed CPU interlay. The code for this CI could be developed using a hardware description language (HDL). This would require some degree of expertise in HDL code development. Additionally, manually producing a CRC CI in HDL code would consume a relatively longer time than developing the same algorithm in C language. Instead, we propose leveraging our interlay HLS design flow to generate a high performance CRC CI without touching any HDL code.

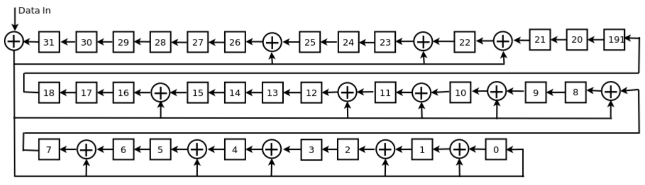

Figure 5: Block diagram corresponding to the CRC-32 algorithm (generator polynomial G=0x04C11DB7).

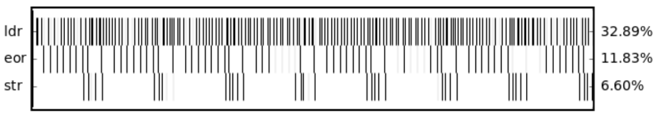

Figure 6: A segment of the instruction trace corresponding to the execution of the CRC32 program running on an ARM Cortex-A9 CPU. Load (ldr), store (str), and exclusive or (eor) instructions are frequently used (other instructions are omitted for clarity).

As a first approach, 4 different versions of a CRC-32 kernel were studied. These kernels developed in C language were derived from the algorithms described in [19]. The Vivado HLS tool was used to translate these 4 CRC-32 kernels into HDL code. An examination of the generated HDL code showed that the Vivado HLS compiler translated the existing C code into several HDL sequential blocks emulating the C program flow. According to our analysis, the Vivado HLS compiler was failing to detect that the CRC operation could be computed more efficiently if it was expressed as a combinational hardware structure consisting of wires and XOR gates (see Figure 5). Even for the CRC kernel that more closely follows the logical circuit, the HLS compiler was unable to generate an efficient hardware description of the CRC circuit without an overhead caused by a substantial usage of sequential blocks. We measured the actual speedup provided by the HLS-generated CRC-32 CIs by comparing their execution time to the execution time of their software counterparts. In our experiment we measured the execution time of each of the 4 versions of the CRC-32 function (crc32_v1-crc32_v4). These software kernels were compiled with optimization level 3 (i.e. O3) and were executed on Gem5 configured to simulate the architectural characteristics of an ARM Cortex-A9 CPU. The execution time of each hardware CRC-32 CI was derived from the report provided by the Vivado HLS tool. For each CRC-32 hardware kernel, two variants were synthesized: 1) one optimized for area (OA), and 2) one optimized for performance (OP). The execution time for all the software and hardware implementations, and the speedups achieved by each hardware CRC-32 CI over their software counterparts are presented in Table 3. Additionally, the same table presents the resource utilization corresponding to each CRC-32 CI.

Listing 1: HLS-friendly C code to compute CRC values.

```
#include "ap_cint.h"
unsigned int crc_ap( int m  ) {

uint1 d_in_0 , d_in_1 , d_in_2 , d_in_3;
uint1 d_in_4 , d_in_5 , d_in_6 , d_in_7;
int i0 = 0, i1 = 1, i2 = 2, i3 = 3;
int i4 = 4, i5 = 5, i6 = 6, i7 = 7;

d_in_0  = apint_get_bit(m, i0);
d_in_1  = apint_get_bit(m, i1);
d_in_2  = apint_get_bit(m, i2);
d_in_3  = apint_get_bit(m, i3);
d_in_4  = apint_get_bit(m, i4);
d_in_5  = apint_get_bit(m, i5);
d_in_6  = apint_get_bit(m, i6);
d_in_7  = apint_get_bit(m, i7);

uint1 crc_q_0= 1, crc_q_1= 1, crc_q_2= 1, crc_q_3= 1;

uint1 crc_c_0 = crc_q_2^crc_q_3^d_in_0^
                d_in_1^d_in_2^d_in_6^d_in_7;
uint1 crc_c_1 = crc_q_2^d_in_0^d_in_3^d_in_6;
uint1 crc_c_2 = crc_q_0^crc_q_3^d_in_1^d_in_4^
                d_in_7;
uint1 crc_c_3 = crc_q_1^crc_q_2^crc_q_3^d_in_0^
                d_in_1^d_in_5^d_in_6^d_in_7;

uint2  crc1_out = apint_concatenate(crc_c_1, lfsr_c_0);
uint3  crc2_out = apint_concatenate(crc_c_2, crc1_out);
uint4  crc3_out = apint_concatenate(crc_c_3, crc2_out);

return crc3_out;
}
```

Table 3: Comparison of execution time between software and hardware implementations of 4 versions of the CRC-32 algorithm. For each hardware version, two variants were synthesised, one optimized for area (OA) and one optimized for performance (OP). A CRC-32 CI developed with a HLS-friendly C coding style is also presented. The resource utilization of each CRC-32 CI is shown.

| CI | Latency SW (ns) | Latency HW (ns) | | Speedup $t_{sw}/t_{hw}$ | | Resources LUT, DSP, BRAM | |
|---|---|---|---|---|---|---|---|
| | O3 | OA | OP | OA | OP | OA | OP |
| crc32_v1 | 35189.0 | 22221.5 | 4825.6 | 1.58× | 7.29× | 238, 0, 1 | 1018, 0, 1 |
| crc32_v2 | 22895.2 | 29141.9 | 5005.4 | 0.79× | 4.57× | 227, 0, 1 | 770, 0, 1 |
| crc32_v3 | 17708.5 | 9594.6 | 4842.2 | 1.85× | 3.66× | 309, 0, 2 | 828, 0, 2 |
| crc32_v4 | 22701.1 | 6369.6 | 3190.3 | 3.56× | 7.12× | 721, 0,1 | 721, 0, 1 |
| crc32_hf | 17708.5 | - | 260.0 | - | 68.11× | - | 318, 0, 0 |

According to our results, modest speedups of up to 7.29× were obtained for the HLS-generated CIs. Based on the previously analysed HDL code, it was possible to observe that the coding style used to implement the CRC-32 software kernels was hampering the synthesis of a more efficient hardware implementation. The structure of the hardware synthesized by the Vivado HLS tool can be greatly influenced by the coding style used to develop a software kernel. We performed an experiment to show how a more "HLS-friendly" C coding style can positively influence the synthesis of a more efficient CRC-32 custom instruction. This experiment was performed as follows: First, HDL code to compute CRC-32 values was generated as a reference. This code implements a boolean equation to compute each of the 32 1-bit CRC output values. Then C code that makes use

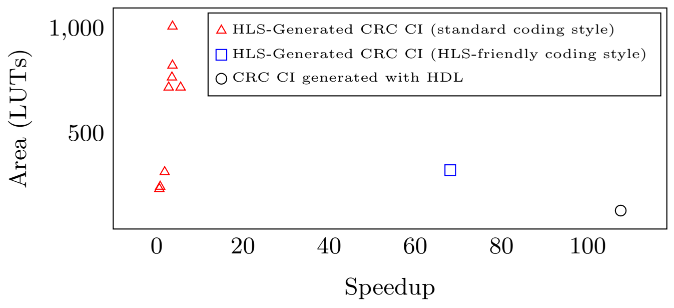

Figure 7: Area vs Speedup values for HLS-Generated Custom Instructions and a HDL-generated Custom Instruction targeting the CRC-32 application.

of the "apint_get_bit( )" and the "apint_concatenate( )" bit-level manipulating functions, is developed. These functions are found in the "ap_cint.h" library provided by the Vivado HLS compiler. With this approach, the function input can be processed as a bit-vector and the result is returned as an integer value. To illustrate this approach, Listing 1 shows a snipet of the HLS-friendly C code to compute CRC values with a small generator polynomial (G $= x^3 + x + 1$) [19].

Finally, the HLS-friendly C code was synthesized with the Vivado HLS tool. An inspection of the generated HDL code showed that this time, the resulting hardware structure consisted mainly of an array of XOR gates performing the computation of each of the 32 1-bit CRC return values. The Vivado HLS tool reported a resource utilization of 318 LUTs, 0 DSPs, and 0 BRAMs for this custom instruction. The latency numbers reported for this CI were used to derive speedups against the fastest software kernel (see Table 3). Figure 7 shows a graphical comparison between the different HLS-generated CRC-32 CIs, including the 4 kernels developed with standard C coding style, and the CRC-32 kernel developed with HLS-friendly C coding style. For comparison purposes, the area and speedup achieved with a handcrafted HDL CRC-32 custom instruction is also presented. Note that the CRC CI generated with HLS-friendly C code is about 10× faster than the CIs generated with conventional C coding style. This in turn results in an overall HLS-enabled speedup of 68× over the fastest CRC-32 software implementation.

## 4. CONCLUSION

In this article we presented a semi-automatic design flow for CPU interlays. We demonstrated that a software-defined path to generate custom instructions leveraging HLS tools is an alternative to complex design flows based on DFG extraction heuristics or RTL handcrafted custom instructions. The here presented design flow is targeted to a CPU interlay consisting of a FPGA fabric embedded at the core of an otherwise hardened CPU. Our case study showed that individual kernels could be accelerated by as much as 68× on a 2400 LUT interlay and full applications by up to 1.7× (AES application) without touching any HDL code. Additionally, we showed how a software kernel developed with an HDL-friendly C coding style allows the Vivado HLS tool to synthesize efficient custom instructions that are up to 10× faster than the custom instructions generated with a conventional C coding style. And 68× over the fastest software CRC implementation. With this, we demonstrated a fully software-driven design flow utilizing CPU interlays.

## 5. ACKNOWLEDGMENT

The first author acknowledges the support from the Mexican National Council for Science and Technology (CONACyt) through scholarship 381920.

## 6. REFERENCES

[1] "ARM Infocenter," www.arm.com.

[2] Z. A. Ye *et al.*, "Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit," in *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2. ACM, 2000, pp. 225–235.

[3] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *IEEE FCCM 1997*. IEEE, pp. 12–21.

[4] Atasu *et al.*, "FISH: Fast instruction synthesis for custom processors," *IEEE VLSI Systems*, vol. 20, no. 1, pp. 52–65, 2012.

[5] K. Seto and M. Fujita, "Custom instruction generation with high-level synthesis," in *IEEE ASAP 2008*, pp. 14–19.

[6] J. Cong *et al.*, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[7] K. Atasu *et al.*, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proceedings of the 40th annual Design Automation Conference*. ACM, 2003, pp. 256–261.

[8] A. Canis *et al.*, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *ACM FPGA 2011*, pp. 33–36.

[9] V. Kathail *et al.*, "SDSoC: A Higher-level Programming Environment for Zynq SoC and Ultrascale+ MPSoC," in *ACM FPGA 2016*. ACM, 2016, pp. 4–4.

[10] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[11] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "CHStone: A benchmark program suite for practical C-based high-level synthesis," in *2008 IEEE International Symposium on Circuits and Systems*, pp. 1192–1195.

[12] "Making a Case for an ARM Cortex-A9 CPU Interlay Replacing the NEON SIMD Unit," Blinded, Submitted to the 2017 FPL Conference.

[13] F. A. Endo, D. Couroussé, and H.-P. Charles, "Micro-architectural simulation of in-order and out-of-order arm microprocessors with gem5," in *IEEE SAMOS 2014*, pp. 266–273.

[14] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys (csuR)*, vol. 34, no. 2, pp. 171–210, 2002.

[15] S. G. Hansen, D. Koch, and J. Torresen, "High speed partial run-time reconfiguration using enhanced ICAP hard macro," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, May 2011, pp. 174–180.

[16] Y. Hilewitz and R. B. Lee, "Performing advanced bit manipulations efficiently in general-purpose processors," in *ARITH'07*. IEEE, pp. 251–260.

[17] W. W. Peterson and D. T. Brown, "Cyclic codes for error detection," *Proceedings of the IRE*, vol. 49, no. 1, pp. 228–235, 1961.

[18] S. E. Anderson, "Bit twiddling hacks," *URL: http://graphics. stanford. edu/˜ seander/bithacks. html*, 2005.

[19] H. S. Warren, *Hacker's delight*. Pearson Education, 2013.