Making a Case for an ARM Cortex-A9 CPU Interlay Replacing the NEON SIMD Unit

DOI:
10.23919/FPL.2017.8056806

Document Version
Accepted author manuscript

Making a Case for an ARM Cortex-A9 CPU Interlay Replacing the NEON SIMD Unit

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*—As an alternative of adding more and more instructions to CPU cores in order to address a wide range of applications, this paper examines to use a mixed grained CPU interlay fabric to provide reconfigurable instruction set extensions. In detail, we are examining to replace the hardened NEON SIMD unit of an ARM Cortex-A9 with an identical sized FPGA fabric. We show that by applying a set of optimizations, we are able to emulate original applications using NEON instructions at the same hardware cost and at very little performance drop by an interlay. Moreover we are demonstrating examples where special custom instructions running on a CPU-Interlay-hybrid are substantially outperforming the original hardened CPU-NEON-system, hence making a strong case to embed reconfigurability as a beneficial feature in future processors.**

I. Introduction

As new mobile applications continue to appear, existing processing architectures for mobile devices are under constant pressure to deliver higher performance at very low power and very low cost. Additionally, mobile chip vendors strive to differentiate their products by providing specialized hardware in the form of accelerator engines or instruction set extensions (ISE) to support one or many of the aforementioned emerging mobile applications. For example, to accelerate applications from the media domain, the ARM architecture introduced an application domain-specific SIMD unit known as NEON [1]. As a result of this arms race, more and more specialized hardware units are cramped into mobile chips. Although more performance can be achieved with this approach, adding hardened application-specific acceleration units comes at the cost of an increased area and larger energy consumption. This points out the core dilemma in the design of mobile processing architectures: *the need for specialization and acceleration on the one hand and the many variations needed for the heterogeneous application domains on the other*. For this reason, *alternative processing architectures capable of achieving better performance without increasing the die-area or energy consumption* of future mobile devices, are required.

The main purpose of this article is to address the problem previously described by adding a mix-grained reconfigurable fabric into an otherwise hardened CPU and combining this with reconfiguration to swap instructions as needed in order to improve on-device processing performance. We call this concept a *CPU interlay* as it adds a programmable layer between a hardened CPU and customizable instructions.

As a case study, we replace the NEON SIMD unit of an ARM Cortex-A9 CPU with a CPU interlay consisting of a constrained mix of fine-grained bit-level operations and coarse-grained ALU primitives. We select the ARM Cortex-A9 processor because it is a widely used IP. Additionally, the ecosystem provided by ARM and its partners facilitates the access to development platforms, IDEs and software with support for this IP. We contend that with our approach, instead of using a CPU's real estate to allocate a domain-bound SIMD unit, a CPU interlay can be allocated in the same area to dynamically implement different sets of custom instructions. The contribution of this work is threefold: 1) We introduce the concept of CPU interlay, which revisits the idea of embedding a reconfigurable fabric into an otherwise hardened CPU. We discuss important aspects such as design, usage, and integration of a CPU interlay. 2) We show that the functionality of the hardened NEON unit can be emulated by a reconfigurable implementation of the NEON ISE running on top of an CPU interlay. We demonstrate that this replacement is realistic and that it can be achieved by applying optimization techniques such as ISA subsetting and vector width customization. 3) We present examples where a CPU interlay can potentially be used to provide the functionality from different instruction sets and even from new custom instructions to boost the performance of a CPU at run-time.

TABLE I: Comparison of related architectures and the CPU interlay.

Architecture	Design Characteristics			
	Implementation	Coupling Type	Interface Characteristics	Target Size
GARP [2]	Hybrid	Tight	RF↔RA	Little
Chimaera [3]	Hybrid	Tight	RF↔SRF↔RA	Little
Legup [4]	Hybrid/Soft	Loose	Avalon Interconnect	Large
Nios II [5]	Soft	Tight	RF↔RA	Small
RISPP [6]	Soft	Tight	RF↔RA	Large
CPU Interlay	Hybrid	Tight	RF↔RA	Small

II. Related Work

The idea of extending a hardened CPU with reconfigurable functional units to accelerate applications has been extensively studied. Classic examples of these types of reconfigurable processing systems are DISC [7], which provides hardware support for a dynamic selection of instructions streams. Garp [2], which couples a hardened CPU register file (RF) directly with a reconfigurable array, and Chimaera, which uses a shadow register file (SRF) to share data between a hardened CPU and a reconfigurable array (RA) [3]. More recently, architectures like CCA [8] and RISPP [6] aim to improve the adaptability of embedded systems by providing a set of specialized functional units that can be dynamically selected at run-time. Existing hybrid architectures such as Legup provide a relatively large amount of reconfigurable resources for implementing large accelerators. While substantial gains can be achieved by offloading large kernels into those accelerators, the associated area and power cost makes their application on the mobile/IOT domain unfeasible for cost-sensitive markets. In contrast, CPU interlays consider a much smaller mix-grained fabric. The rational behind this is that the CPU interlay is targeting small

kernels that can provide significant performance gains at very small area overhead. To simplify a comparison between the here proposed CPU interlay and closely related reconfigurable architectures, Table I is presented.

III. CPU Interlays

Interlays are reconfigurable fabrics allowing the customization of an instruction set after fabrication and while a system is running. Consequently, it is located logically between the software instruction stream and the physical hardware. Interlays can be seen as specialized FPGA fabrics that are of small capacity and that interface directly to a CPU and/or cache. With this, interlays allow for a much tighter integration without the need of complex drivers. Furthermore, interlays target a software centric design flow which will commonly result in better design productivity than following a hardware centric design methodology which includes the development of complex accelerators and their integration. While relatively small interlays may obviously not provide massive acceleration, they still provide an opportunity for substantial performance boosts, lower power, and faster time-to-market by allowing customization of probably low-cost mass produced chips.

Interlays provide a path for accessing lower abstraction levels (i.e. the ISA) that are commonly hardened. In order to use interlays at an industrial level, an ecosystem with different parties will be needed, each of them providing a different set of skills, as described briefly in the following subsections.

1) Interlay Fabric Design: From a distance, an interlay fabric should look very similar to an FPGA fabric. However, there are several differences that require further investigation including routing architecture, the number of clock networks used, the interlay primitives required and the fabric layout.

2) CAD and Compilation Tools: An interlay fabric will need tools for transforming RTL and/or HLS specifications into interlay configuration bitstreams. Because we assume that interlay fabrics will provide orders of magnitude less logic and routing than recent FPGAs, more optimization at compile time will be feasible. Alternatively, because a small interlay will demand little resources for tools, even just-in time compilation could be a feasible option for interlays.

3) SoC Integration: We assume to integrate an interlay into a given hardened SoC which should omit complex tasks as much as possible. This includes VLSI aspects such as floorplanning and manufacturing as well as changes in the hardware architecture of the SoC (e.g., the interface to the cache/memory subsystem and the integration with the original scalar processor).

4) Interlay Configurations: The design of interlay configurations can follow an IP-core model as it is known from the FPGA industry (with corresponding vendors) as well as automated HLS compilation. IP-cores will typically be bundled with software libraries that provide an easy usable API (similar to how major CPU vendors use their embedded hardware accelerators, like for example for encryption).

The complexity of this ecosystem is enormous and we use the following approaches for our investigation: Instead of designing an optimized interlay fabric and corresponding tools, we take commercial off-the-shelf FPGAs and add constraints for emulating some of the characteristics of an interlay fabric (e.g. using bounding boxes and adding restrictions on the routing fabric). We anticipate that this will give us conservative numbers for area, performance, and reconfiguration time that could be improved by optimizing the interlay fabric.

IV. Replacing the NEON Unit with a CPU Interlay

A. Implementation

To measure the area that a reconfigurable implementation of NEON would occupy on a FPGA-like fabric, a design compatible with the NEON ISE was developed in RTL. It is based on the specification described in the ARMv7 architecture reference manual [1] Our design is a 3-stage pipeline where 1) the two 128-bit operands are fetched from the register file, 2) the operation is executed by the SIMD ALU, and 3) one 128-bit result vector is written back to the register file. To reduce the LUT count, the register file was implemented with BRAM blocks. The single-precision floating-point operations supported by NEON were implemented using the Vfloat floating library which is specifically designed for FPGAs [9]. Where possible, the design uses DSP blocks instead of LUTs to reduce area and to enhance the speed of the design. The synthesis tool was configured to target the Zynq device from the FPGA vendor Xilinx. Table II shows the resource breakdown for our reconfigurable NEON implementation.

TABLE II: Utilization breakdown for our NEON RTL implementation.

Functional Unit	FPGA Primitive		
	LUT	DSP	BRAM
– NEON ALU	10968	275	0
– arithmetic-ops	640	64	0
– boolean-ops	388	4	0
– comparison-ops	926	0	0
– shift-ops	718	0	0
– multiply-ops	819	88	0
– miscellaneous-ops	7699	119	0
– NEON Register File	0	0	4
– NEON Unit	11360	275	4

B. CPU Interlay - hardened NEON Gap Analysis

Since we are aiming to replace the hardened NEON with a CPU interlay, we measured the gap between both implementations to quantify and to mitigate the effects of this replacement. First, we determined the area for a single hardened NEON unit in terms of FPGA primitives. For our experiment we used the Zynq chip because it provides a dual-core ARM Cortex-A9 processor (featuring a NEON unit in each core) embedded in a Xilinx Artix-7 FPGA fabric. Both the ARM core and the FPGA fabric are fabricated on the same die using a 28 nm process technology.

TABLE III: Area utilization breakdown for a Dual-Core ARM Cortex-A9 processor and the NEON unit on a Xilinx Zynq chip.

Functional Unit	FPGA Primitive Equivalent		
	LUT	DSP	BRAM
Dual Core ARM Cortex A-9 Processor	10400	80	40
Single Core ARM Cortex A-9 Processor	5200	40	20
Two NEON Units	2080	16	8
Single NEON Unit	1040	8	4

By analysing the floorplan of the Zynq chip using EDA tools from the vendor Xilinx, it is possible to accurately determine the amount of FPGA primitives that can fit in the area occupied by the ARM processor system. Table III summarizes our measurements.

1) Area Gap: The area gap measured between our RTL NEON implementation and the existing hardened NEON using the methodology described above is summarized in Table IV. The area gap is expressed as the number of times that our RTL NEON implementation is *bigger* than the area reference that we determined for the existing hardened NEON. Our results

show that our RTL NEON implementation would occupy an area 10.9× (LUTs) and 34.4× (DSPs) bigger than the existing hardened NEON. The reasons for the high DSP count are that 1) these coarse-grained multipliers and adders were heavily used to obtain lower LUT utilization, and 2) the Xilinx Zynq DSP blocks are not well tailored for exploiting different levels of sub-word parallelism.

TABLE IV: Area an Latency characteristics of our RTL NEON implementation and a hardened NEON unit.

FPGA Resources				Operating Frequency	
Hardened NEON		RTL NEON		Hardened NEON	RTL NEON
LUT	DSP	LUT	DSP	MHz	MHz
1040	8	11360 (10.9×)	275 (34.4×)	650	164 (3.9×)

2) Latency Gap: The latency gap measured between our RTL NEON implementation and the hardened NEON is summarized in Table IV. We measured a gap of 3.9×. This means that that our soft NEON implementation runs at a 4× lower clock speed than the existing NEON unit. It can be noted that despite the fact that hard arithmetic blocks (DSP blocks) are used, the support of single-precision floating-point operations adds substantial complexity to the design of our RTL NEON implementation.

V. Closing the Gap

To close the gap between our RTL NEON implementation and the hard implementation of the NEON ISE, we applied ISA subsetting [10], vector width customization [11], and a design technique known as "folding" [12]. The former two optimization techniques consist of supporting only the instruction subsets and vector widths required by each particular application. The third technique is extensively used in the domain of DSP architectures and consists of transforming an operation that is normally executed in a single time unit, into N steps executed on the same hardware operator over N time units. To measure the impact of applying these optimization techniques, we used the following methodology: First we analysed profiling data from a set of media and security application benchmarks. Then based on the statistics previously analysed, we defined *stripped-down application-specific NEON ISE subset configurations*. Finally, we obtained area and latency numbers for the different application-specific RTL NEON configurations.

A. NEON Utilization

We measure the percentage of NEON ISE that is used by media and security applications. For portability, the selected benchmarks don't make explicit use of ARM/NEON-specific instructions. Instead, we instruct the GCC compiler (arm-linux-gnueabi-gcc version 5.4.0) to perform automatic vectorization on the application's code. A static analysis of the code produced by GCC shows that the studied benchmarks use on average 13% of the general-purpose NEON ISE. Figure 1, shows the relative amount of NEON instructions used by each application. Figure 2, shows the distribution of the different types of NEON instructions used by each application. Likewise, our analysis showed that 8-bit and 32-bit operations are more heavily used than 16-bit and 64-bit operations. Hence providing an opportunity for further area optimization. Figure 3 shows the distribution of the size of operations performed by each application.

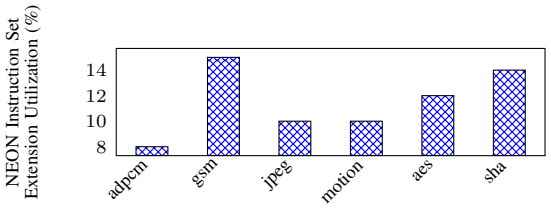

Fig. 1: Relative NEON ISE utilization by different applications.

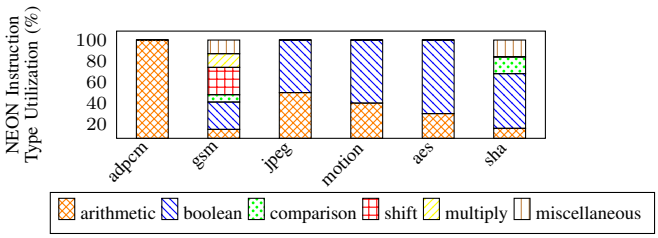

Fig. 2: Distribution of the different types of NEON instructions used by media and security applications.

B. Application-Specific Optimization

We generated optimized stripped-down configurations of our RTL NEON implementation. Table V shows the area occupied by these optimized configurations (expressed in terms of FPGA primitives). In parenthesis the area gap and the folding factor (N=2, 4) used is presented for each configuration. With these optimizations, the original area gap (10.9× LUT, 34.4× DSP) for the full RTL NEON design is substantially reduced to between 0.5× and 1.0× for LUTs, and between 0.5× to 2.8× for DSPs. Figure 4 shows the relative area savings that were achieved with these techniques combined.

Similarly, we measured the latency gap between the optimized RTL NEON configurations and the existing hardened NEON. Our experimental results, presented in Table V, show that the latency gap between the full CPU interlay (i.e. 3.9×) and the hardened NEON, is reduced to between 2.2× and 2.9× for the optimized NEON configurations. Because only a few NEON instructions and vector sizes are used by each application, the corresponding NEON configuration becomes less complex which improves the critical path delay. Figure 5 shows the relative latency gap reductions achieved by the application-specific NEON configurations.

C. Performance Considerations

We tuned Gem5 to simulate the characteristics of an ARM Cortex-A9 processor, then we executed the application bench-

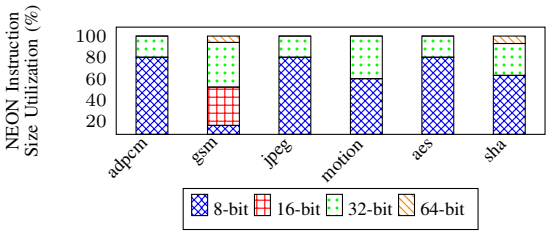

Fig. 3: Distribution of the vector sizes of NEON instructions used by media and security applications.

TABLE V: CPU Interlay / Hardened NEON Gap

Application	Resources		Frequency
	LUT	DSP	MHz
adpcm	933 (0.9×) (N = 2)	12 (1.5×)	247.6 (2.6×)
gsm	960 (0.9×) (N = 4)	22 (2.8×)	220.2 (2.9×)
jpeg	993 (1.0×) (N = 4)	18 (2.3×)	250.2 (2.5×)
motion	706 (0.7×) (N = 2)	22 (2.8×)	280.7 (2.3×)
aes	528 (0.5×) (N = 2)	6 (0.8×)	303.6 (2.2×)
sha	473 (0.5×) (N = 2)	4 (0.5×)	302.9 (2.2×)

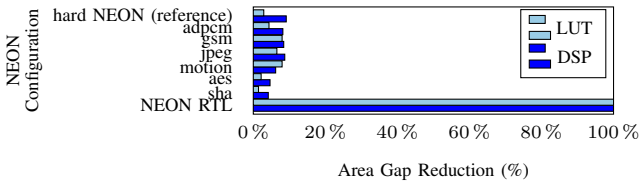

Fig. 4: Relative area gap reductions achieved by different application-specific NEON configurations.

marks on Gem5 using this baseline configuration. To simulate the impact on the performance caused by replacing the existing hardened NEON with our RTL NEON implementation, we manually increased the latency values defined for the NEON operations. We were able to do this by modifying the Gem5 configuration file where the latencies of the different functional units of the ARM processor, including the NEON unit, are defined (i.e. the O3_ARM_v7a.py configuration file). Because our earlier experiments showed that the slowest NEON configuration is 3.0× slower than the hardened NEON, we incremented the latency value of the NEON operations in the Gem5 simulator by that factor. Next, we executed the benchmark applications on Gem5 with this modification. Finally, we compared the execution time corresponding to the baseline and the modified configuration reported by Gem5. Table VI summarizes the performance values obtained for the different benchmark applications.

TABLE VI: Performance Gap (CPU Interlay / Hardened NEON)

Application	Execution Time (μs)	
	Hardened CPU + Hardened NEON	Hardened CPU + CPU Interlay
adpcm	259	285 (1.10×)
gsm	65	69 (1.06×)
jpeg	5411	6096 (1.13×)
motion	72	74 (1.03×)
aes	202	213 (1.05×)
sha	597	793 (1.33×)

VI. CPU Interlay Applications

In this section, we will show how the reconfigurability introduced by the interlay fabric can be leveraged to obtain substantial performance gains.

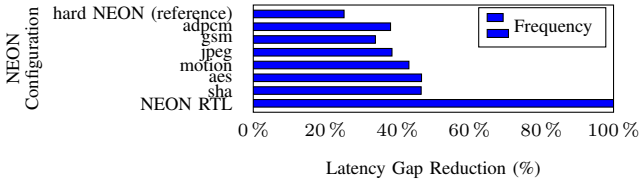

Fig. 5: Relative latency gap reductions achieved by different application-specific NEON configurations.

1) CRC Operator: We implemented a handcrafted CRC-32 operator for the CPU nterlay and compared this with a software implementation running on the ARM Cortex-A9. We reduced execution time from 17708.5 ns to just 260 ns (i.e. a 68× faster). This operator only takes 121 LUTs.

2) RGB2Y Operator: According to our experiments, the RGB2Y function running on an ARM processor using existing NEON instructions consumes 1701 ns to process a 360x270 RGB image. In contrast, a handcrafted RTL RGB2Y operator takes 34 ns (i.e. a 50× faster). The RGB2Y operator consumes 136 LUTs.

3) Human Skin Colour Classifier Operator: With a resource utilization of 132 LUTs, a hardware Human Skin Colour Classifier Operator achieves 3.6× faster classification on a 360x270 image over a software implementation.

A summary of the here presented custom instructions targeting the CPU interlay is presented in Table VII.

TABLE VII: Summary of Custom Instructions targeting the CPU Interlay.

Algorithm	Resources	Latency	Speedup
CRC-32	121 LUTs	260 ns	68×
RGB2Y	136 LUTs	34 ns	50×
Skin Colour Classifier	132 LUTs	1726.2 ns	3.6×

VII. Conclusion

In this article we introduced the concept of an CPU interlay which is aimed to be embedded into the core of an otherwise hardened CPU. We demonstrated that a relatively small FPGA can substitute a hardened vector unit at the same cost (i.e. die area) and at almost the same performance when using FPGA design specific optimizations. With the possibility to customize the processor, we provide all the opportunities that come with (run-time) reconfiguration which includes in particular in-field updates and specialized instructions.

Acknowledgment

This work is kindly supported by the Mexican National Council for Science and Technology (CONACyT) under grant 381920.

References

[1] "ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition ," www.arm.com.

[2] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," in *IEEE FCCM 1997*.

[3] Z. A. Ye *et al.*, "CHIMAERA: A High-performance Architecture With a Tightly-coupled Reconfigurable Functional Unit," in *ACM Computer Architecture News*, vol. 28, no. 2, 2000, pp. 225–235.

[4] A. Canis *et al.*, "LegUp: An open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems," *TECS*, 2013.

[5] Altera Corp., "Nios II Custom Instruction. User Guide."

[6] L. Bauer, M. Shafique, and J. Henkel, "RISPP: A run-time adaptive reconfigurable embedded processor," in *IEEE FPL 2009*, pp. 725–726.

[7] M. D. Nemirovsky *et al.*, "DISC: Dynamic instruction stream computer," in *Proceedings of the 24th annual international symposium on Microarchitecture*. ACM, 1991, pp. 163–171.

[8] N. Clark *et al.*, "An architecture Framework for Transparent Instruction Set Customization in embedded processors," *IEEE ISCA 2005*.

[9] X. Wang and M. Leeser, "VFloat: A Variable Precision Fixed- and Floating-Point Library for Reconfigurable Hardware," *ACM Trans. Reconfigurable Technol. Syst.*, 2010.

[10] P. Yiannacouras, J. G. Steffan, and J. Rose, "Exploration and customization of FPGA-based Soft Processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2007.

[11] P. Yiannacouras *et al.*, "VESPA: Portable, Scalable, and Flexible FPGA-based Vector Processors," in *ACM CASES*, 2008.

[12] K. K. Parhi *et al.*, "Synthesis of Control Circuits in Folded Pipelined DSP Architectures," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 1, 1992.