PEPA: Performance Enhancement of Embedded Processors through HW Accelerator Resource Sharing

Qilin Si
qilin.si@UTDallas.edu
The University of Texas at Dallas
Deptm. of Electrical and Computer Engineering
Richardson, Texas, USA

Benjamin Carrion Schaefer
schaferb@UTDallas.edu
The University of Texas at Dallas
Deptm. of Electrical and Computer Engineering
Richardson, Texas, USA

ABSTRACT

To improve the performance while reducing the power consumption, embedded processors in Systems-on-Chip (SoC) often now include tightly coupled hardware accelerators that can execute dedicated tasks orders of magnitude more efficiently (faster and lower power). These hardware accelerators though require significant hardware resources as one of the main reason for their efficiency is that they extensively exploit the parallelism of these dedicated tasks mapped on them. The question that we address in this work is if these hardware resources can be re-used by the CPU when executing a different application.

To address this, in this work we propose an integrated methodology that adapts automatically the CPU architecture with the tightly integrated hardware accelerator(s) such that any applications that will run on the CPU (different from the accelerator) can benefit from the additional hardware resources available in the hardware accelerator(s), such that the execution of these applications is accelerated. We also propose a VLIW compiler backend that based on the resources shared, re-generates the assembly instructions such that the new applications can benefit from this new architecture. Experimental results show that our proposed methodology is very effective, achieving average speed up of 1.7×.

CCS CONCEPTS

• **Computer systems organization** → **High-level language architectures**; • **Hardware** → **Hardware-software codesign**.

KEYWORDS

Hardware accelerator, SW acceleration, resource sharing.

ACM Reference Format:
Qilin Si and Benjamin Carrion Schaefer. 2023. PEPA: Performance Enhancement of Embedded Processors through HW Accelerator Resource Sharing. In *Proceedings of the Great Lakes Symposium on VLSI 2023 (GLSVLSI '23), June 5–7, 2023, Knoxville, TN, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3583781.3590277

1 INTRODUCTION

Most Integrated Circuits (ICs) are now heterogeneous System-on-Chip (SoC) that include multiple embedded processors, on-chip memory, different types of interfaces and a growing number of dedicated hardware accelerators. These hardware accelerators are often the differentiating factor between competing SoCs.

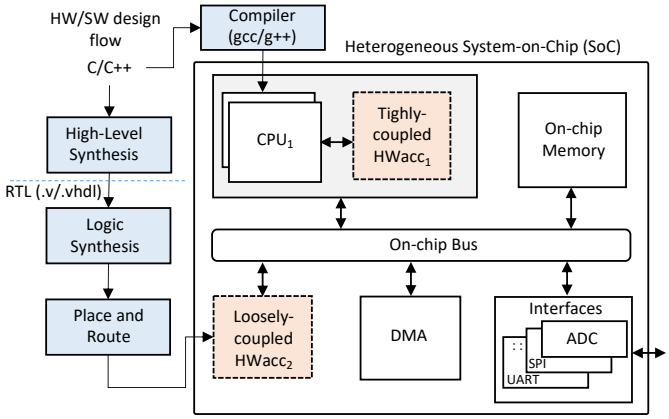

Figure 1: Target architecture and typical VLSI design flow.

Fig. 1 shows an example of a heterogeneous SoC similar to the one targeted in this work that contains two different types of HW accelerator configurations that can be classified as either loosely-coupled or tightly-coupled accelerators. Loosely-coupled accelerators have the advantage that every master in the SoC can access them [1], while in the tightly-coupled case [2, 3], they are faster and more energy efficient as data does not have to be shuttled across the SoC [4], but only one master can access them.

Fig. 1 also shows a typical VLSI design flow highlighting that the hardware accelerators can be directly generated from C/C++ descriptions synthesized through High-Level Synthesis (HLS) into RTL (Verilog or VHDL) or manually generated in any two Hardware Description Language (HDL) like Verilog or VHDL. The flow also shows that the SW running on the embedded processors is compiled using compilers like gcc or g++.

In this work we mainly target the tightly-coupled HW accelerators as the main goal is to re-use their HW resources to accelerate any application running on the embedded CPUs to which they are directly connected to. Intuitively one can see that the tightly-coupled accelerator will not always be in use due to the flexible nature of the CPU that will require it to run many other applications. It can also happen that due to changing workloads when the SoC is being deployed or if the SoC is being re-purposed for other applications, that the accelerators are no longer used. Thus, architectures and methodologies are required that enable to tap on the HW accelerators' resources. In this work we target the functional units (FUs), but future work could include internal memories

GLSVLSI '23, June 5–7, 2023, Knoxville, TN, USA

ACM ISBN 979-8-4007-0125-2/23/06.
https://doi.org/10.1145/3583781.3590277

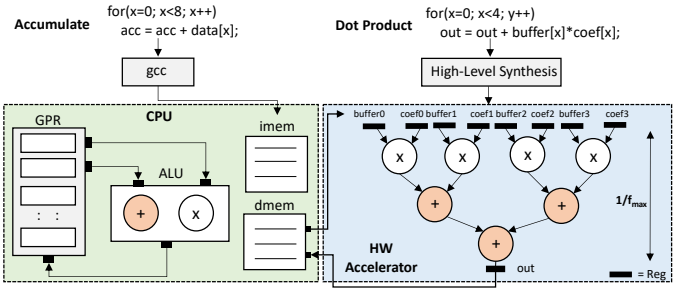

Figure 2: Motivational example of CPU+HW Accelerator. HW accelerator executes the Dot product, while CPU accumulates 8 values.

and registers in the accelerator. One of the reasons that HW accelerators can execute applications order or magnitudes faster than general-purpose CPUs is that they fully parallelize the dedicate application by using multiple hardware resources in parallel (e.g., multiple multipliers and adders). This opens the door to re-using these FUs by the CPU, which typically only contains one Arithmetic Logic Unit (ALU) with a single FU of each type. Based on this, the main contributions of this work are:

- Introduce a methodology to enable internal resources of tightly-coupled hardware accelerators to be shared with embedded CPUs.
- Present a custom instruction flow that makes use of the newly shared HW resources and present extensive experimental results showing the effectiveness of the proposed approach for different types of applications.

2 MOTIVATIONAL EXAMPLE

Fig. 2 shows a motivational example for this work. In particular it shows the target hardware architecture composed of a general-purpose processor (CPU) and a dedicated tightly-coupled HW accelerator. The processor is composed of general-purpose registers (GPR), an ALU, where only the adder and multiplier are shown for simplification and instruction and data memories (imem and dmem). The processor can execute any application compiled on it giving its traditional flexibility, albeit having low performance and energy efficiency. The HW accelerator on the other hand performs a fixed function. In this case a 4-vector dot product, which in this particular example is generated through HLS, but which can be also generated using Verilog or VHDL. Fig. 2 shows the dot product high-level code snippet and its RTL implementation. In this case it uses 4 multipliers and 3 adders as the synthesized circuit structure follows a classic tree height reduction structure to reduce the circuit delay.

The question that we try to address in this work is how to speed up the execution of other tasks running on the CPU by using the hardware resources of the accelerator. As shown in Fig. 2, the CPU has to run an application to accumulate 8 values. Using the processor's ALU would require at least 8 clock cycles (8 individual additions), but considering that the accelerator has three additional adders that are not being used, these could be used to reduce the computation time by half making use of the adder in the ALU and the three additional adders in the accelerator. Based on this, we can define the goal of this work as follows:

Problem Definition: Given a general-purpose CPU (CPU_{RTL}) enhanced with a tightly coupled hardware accelerator ($HWacc_{RTL}$), develop an automated flow that given a new application to be executed on the CPU different from the hardware accelerator, C_{app}, extract the individual resources in $HWacc_{RTL}$ = {FUs = [add, sub, mul, div]} and make them sharable between the CPU and the accelerator such that the execution of C_{app} is accelerated. This also involves adding a new compiler back-end aware of these shared resources in order to generate custom instructions that make use of these resources.

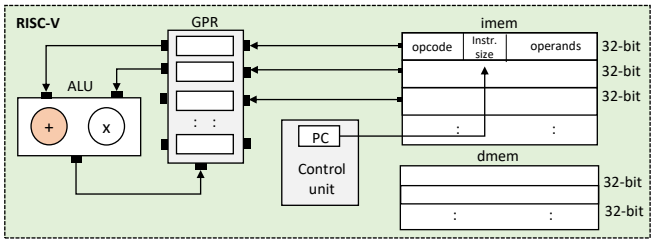

Figure 3: CPU based on modified RISC-V architecture.

3 RELATED WORK

CPUs are ubiquitous in every electronic system. Thus, much research has been done to improve their performance and reducing their energy consumption. Some popular techniques that are being used in all modern processors include pipelining and Instruction Level Parallelism [5] where multiple instructions are executed concurrently, with superscalar architectures doing this dynamically and VLIW processors statically (at compile time). Other orthogonal approaches include Application Specific Processors (ASIPs) which allow to build custom instructions based on specific applications running on the processor [6, 7]. This approach is well-known as shown by the fact that the two major EDA vendors offer commercial ASIP flows [8, 9]. Another approach is to use tightly-coupled dedicated HW accelerators. These accelerators perform complete sub-tasks orders of magnitude faster and energy efficiently than the processors [2, 3, 10–13]. The common denominator across these applications to be accelerated is that they have embarrassingly high amounts of parallelism.

The problem is that embedded processors are often in a wide range of applications due to their flexibility, many of which do not require the use of the hardware accelerator. The question that this work address, is how to re-use these resources.

4 CPU ARCHITECTURE DESCRIPTION

The CPU used in this work is based on a 32-bit pipelined RISC-V architecture taken from [14, 15]. The key enabler of our flow is that the RISC-V processor is generated using HLS. This architecture was modified to be able to accommodate the execution of multiple concurrent instructions as shown in Fig. 3. In particular, first the general-purpose register (GPR) bank was fully expanded so that multiple registers can be read/written in the same clock cycle. Secondly, because the instructions to be executed concurrently are grouped together into a single VLIW style instruction, the extensible RISC-V ISA is leveraged by encoding the length of the instruction using 3-bits (we noticed empirically that more bits were not needed). Based on this, the control unit knows the size of the instruction. This, combined with the expanded GPR allows to

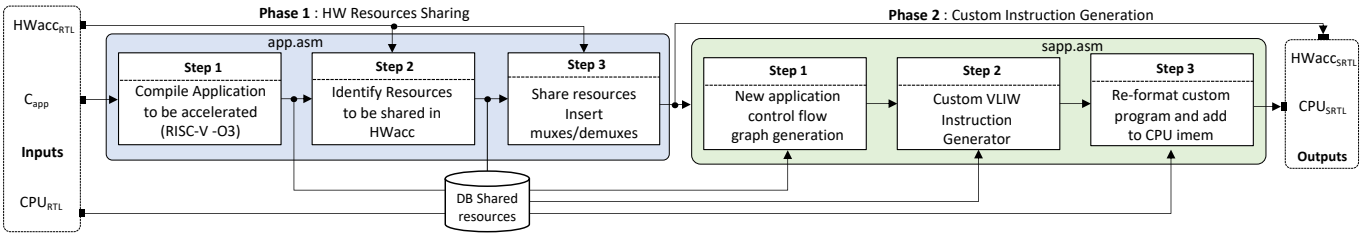

Figure 4: Overview of complete flow composed of two phases. Phase 1: Identify and share resources in HW accelerator. Phase 2: Generate custom instructions that make use of those resources.

execute multiple LOAD/STORE instructions in parallel. As shown in Fig. 3 the Program Counter (PC) in the control unit of the processor points to the next instruction in the instruction memory (imem) to be executed. It then extracts the bits that represent the size of the instruction and based on this fetches concurrently the consecutive N instructions that form this VLIW instruction, where each instruction still remains 32-bit to simplify the instruction decoding process. This modified processor is in turn synthesized into Verilog using HLS.

5 PROPOSED ACCELERATION FLOW

Fig. 4 shows an overview of the proposed complete flow that we call PEPA: Performance Enhancement of Embedded Processors through HW Accelerator Resource Sharing. Our proposed flow can be executed in two modes. **Mode 1** involves that we do not know what other SW applications might be executed on the processor and hence, all of the resources in the hardware accelerator will be shared. In **mode 2**, we do know the SW applications that will be executed on the CPU and hence can more precisely target which resources in the accelerator to share, hence, reducing the overhead associated with the sharing logic. In **mode 1** the inputs of our flow are the RTL description of the processor (CPU_{RTL}) and that of the HW accelerator ($HWacc_{RTL}$), while in **mode 2** an additional input are the SW application that we want to accelerate (C_{app}).
In both cases the outputs are the new HW accelerator RTL code that enables sharing its resources ($HWacc_{SRTL}$) and a customized compiler backend that generates VLIW instructions for new application that can leverage the shared resources. The flow itself is composed of two main phases. Phase 1, as shown, identifies which HW resources to be shared from the accelerator, while phase 2 recompiles the program to make use of these resources. Basically phase 1 prepares the underlying HW, while phase 2 compiles the SW to make use of these shared resources. The next subsections describe these two phases in detail.

Phase 1: HW Resource Sharing: This first phase modifies the HW accelerator's RTL description in order to allow its main resources to be shared. These include FUs like adders, multipliers and dividers. This is basically done by inserting muxes/demuxes at the inputs and outputs of each resources to be shared. In **mode 1** this sharing logic is inserted at each of the resource of the accelerator. This makes the resultant architecture very flexible as any SW application to be executed on the processor can potentially use them, but has a larger area and delay overhead as the cost of the muxes is often not negligible as we will show later. The other option is to take as input the application(s) that will be executed on the processor (C_{app}), and only share the resources that are needed by this particular application(s) (**mode 2**) . This obviously reduces the overhead associated with the sharing logic, but makes the approach less flexible as any new application that was not considered during this phase will not be able to leverage un-shared resources. This first phase can be further sub-divided into three steps as follows.

Step 1 Compile Application to be Accelerated on CPU: This first step is only required if we only want to share the resources needed by a given SW application. Thus, this steps takes as input the application in a high-level language, C_{app}, and compiles it using a compiler for the target CPU, e.g., gcc-riscv. The output is the assembly code (app.asm) for this application that includes all of the arithmetic operations required and the total number of distinct registers used. It should we noted that by default we use gcc's -O3 compiler option (optimizes for performance).

Step 2 Identify Resources to be Shared in HWacc: This second step takes as input the RTL description of the HW accelerator ($HWacc_{RTL}$) and the compiled application to be executed on the CPU obtained in step 1 (app.asm) and identifies which resources in the accelerator can accelerate the execution of the application. As mentioned previously, this step can also be executed without the application. In this case, all of the FUs are shared.

The HW accelerator ($HWacc_{RTL}$) is first parsed generating creating a data dependency graph (DDG). This graph is traversed performing a static timing analysis following [16] for each the paths. The process then extracts all of the *sharable* resources starting with the resources that are not in the critical path. The timing of the different HW resources is obtained from a pre-characterized library similar to what HLS tools do that contains the area and delay of different RTL components. Currently we only consider sharing FUs. The type and also the bitwidth of these resources is annotated into a data base of shared resources. This is important because when building the custom accelerator FUs might have reduced bitwidths to optimize the area, power and delay of the accelerator. This implies that arithmetic operations to be used need to match the minimum bitwidth of the processor in order to be usable.

Step 3 Shared Resources: This last step takes as input the data base of resources generated in step 2, and adds the necessary sharing logic. This basically includes modifying the datapath of the hardware accelerator by inserting muxes/demuxes at the inputs and outputs of the shared resources. Fig. 5 shows an example of one adder from the motivational examples being shared.

One of the main issues with this approach is that the sharing logic inserted at the RTL can now affect the timing of the circuit by increasing the critical path delay. Having sorted the sharable FUs based on being on the critical path or not in step 2 helps minimizing the risk of not achieving timing closure. The resultant

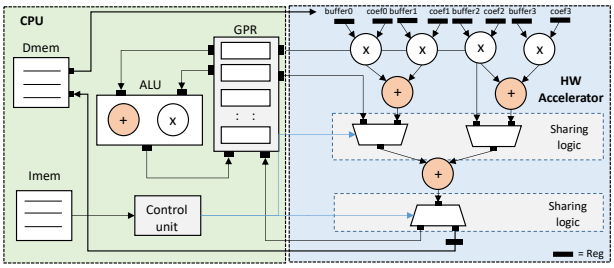

Figure 5: Processor structure with shared logic inserted: (i) Control Unit update; (ii) Mux/demux; (iii) GPR.

HW accelerator ($HWAcc_{SRTL}$) is in turn synthesized once (logic synthesis) with the target technology library and timing constraints. If the timing is not met then one option is to re-synthesize the HW accelerator using different HLS synthesis constraints such that a circuit with more positive timing slack is generated and then repeating this step. We did not encounter this problem during our experimental evaluation due to the relative lower timing delay of the muxes as compared to the FUs.

In addition, the control unit that decodes the instructions also needs to be updated in order make use of these resources so that it can set the control signal for the muxes that share these units at runtime. The output of this phase is the new RTL description of the HW accelerator with shared hardware resources ($HWAcc_{SRTL}$).

Phase 2: Custom Instruction Generation: This second phase generates the custom VLIW instructions that enable the processor to make use of the shared resources extracted in phase 1. Fig. 6 shows an overview of our proposed flow using an FFT as example. The inputs to this phase are the application to be executed on the processor (C_{app}) and the number and type of shared hardware resources. The output is the assembly code with the new custom instructions and the machine code that can be directly used in the instruction memory of the RTL description of the hardware (for verification). This phase follows three steps described in detail as follows:

Step 1 Application Control flow Graph Generation: This first step takes as input the assembly code of the compiled application to be accelerated on the CPU (app.asm) and generates a data dependency graph (DDG) of the instructions. It should be noted, as shown in Fig. 6, we use gcc's compiler option -O3 to generate the fastest possible compiled code. This DDG is then passed the next step that will merge these instructions into custom VLIW instructions.

Step 2 Custom Instruction Generation: The DDG generated in step 1 is then analyzed in order to group instructions together similar to a VLIW processor with the objective to maximize the instructions executed in parallel based on the resources available in the CPU (ALU) and HWacc . This problem can be formulated as a resource constraint instruction scheduling. Different approaches have been demonstrated in the past to e.g., maximize performance [17, 18], reduce power consumption [19] or the code size [20]. In our particular case we focus on increasing the performance by grouping as many instructions as possible based on the new resources available to reduce the execution time.

In this work we use a greedy algorithm that traverses the control flow graph and tries to parallelize as many instructions as possible based on the available resources. A depth first search (DFS) is performed first annotating any data dependencies. In particular, Read-after-Write (RAW), and because we also allow out-of-order execution, we also annotate Write-after-Write (WAW) and Write-after-Read (WAR) dependencies. Once these dependencies are annotated, our processes traverses the control flow graph again grouping as many instructions as possible ensuring that the data dependencies are kept.

As shown in the example in Fig. 6, 4 multiplications in the FFT code snippet and 2 additions are grouped together as a custom vector instruction (vmul and vadd), highlighted in red in the figure. This is possible because the processor now has 4 multiplier and 4 adders as shown in the example. The output of this step is hence, the new assembly code with the custom instructions (sapp.asm).

Step 3 Application Specific Program Re-formatting: This last step is the backend of our proposed flow, which re-formats the instruction generating the new machine code for the custom vector instructions. The output of this step is the new program that uses certain shared FUs from the tightly coupled hardware accelerator.

Table 1: Experimental Setup

HLS Tool	NEC CyberWorkBench 6.1.1
Logic Synthesis	Synopsys Design Compiler 2018.06-SP1
Power estimator	Synopsys Prime Power 2019.03-SP5
RTL Simulator	Synopsys VCS 2018.06
Synthesis Technology	Nangate open cell 45nm
Target Frequency	250MHz

6 EXPERIMENTAL RESULTS

Table 1 shows an overview of the experimental setup used to test our proposed flow. CyberWorkbench v. 6.1.1 from NEC is used as HLS tool. Synopsys Design Compiler (DC) for logic synthesis and Synopsys Primer power for power estimation. Synopsys VCS 2018.06 is used as RTL simulator. The target technology is Nangate 40nm open cell technology library, and the target synthesis frequency is set to 250MHz.

The embedded CPU used is a RISC-V processor taken from [14, 15] and synthesized into RTL using HLS. As tightly coupled hardware accelerators we use four benchmarks from the S2CBench benchmark suite [21] from different domains and different complexities. In all cases we synthesize the designs setting the HLS options to maximizing their performance (unroll loops, and inlining functions).

Table 2 shows an overview of the different configurations (RISC-V+HW accelerator) generated. E.g., S3 contains the RISC-V CPU and a three-stage interpolation filter instantiated as tightly integrated HW accelerator. The table also shows the number of sharable functional units that each HW accelerator has, e.g., in the S3 case 4 adders and 5 multipliers.

To test our proposed flow we run these application in software on the different systems, e.g., running a sobel filter on S3 as well as additional applications as shown in Table 3. This will help us to measure the performance and energy improvement of our flow (PEPA) vs. running that same application purely on the embedded processor (SW_only) as well as running a given application directly on the HW accelerator (HW_{only}). This can obviously only

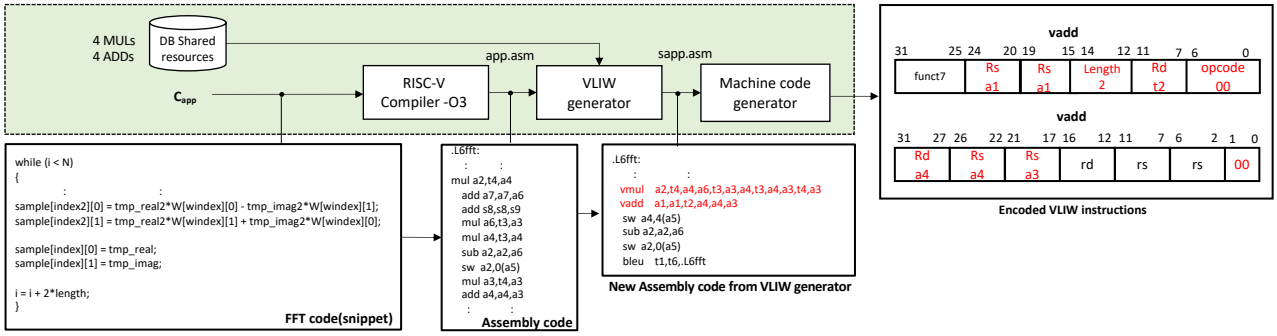

Figure 6: Overview of custom VLIW instruction generation

Table 2: CPU+tightly-coupled HW accelerator configurations

System		S1	S2	S3	S4	#Add	#Mul
CPU (RISC-V)		•	•	•	•	1	1
HWacc	sobel	•				2	3
	fir		•			3	3
	interp			•		4	5
	matinv				•	8	8

be done when the application running is also implemented as HW accelerator in the system, e.g., in S3 running the interpolation filter. Table 3 shows the main characteristics of these applications to be accelerated in terms of their lines of C code (C), compiled assembly instructions (ASM), number of assembly instructions after our flow generates the custom instructions (VLIW ASM), and finally the average number of instructions grouped together (avg. parallel).

Table 3: SW applications running to be accelerated with PEPA

Benchmark	Lines C	ASM	VLIW ASM	Avg. parallel
sobel	108	215	185	2.76
fir	72	110	95	2.58
interp	130	255	218	3.1
matinv	127	333	266	2.59
fft	182	205	177	2.96
dct	60	173	148	2.7
idct	217	309	266	3.68
ann	228	361	323	2.59

Fig 7 shows the experimental results in terms of speedup and energy for our proposed method (PEPA) taking the SW only execution on the RISC-V processor (SW_{only}) as baseline (1.0) and also showing when the application being executed matches the HW accelerator (HW_{only}). The figure also shows the average speedup results for all of the benchmarks in the last entry (**AVG.**), ignoring the HW_{only} case.

Several interesting observations can be made from these results. Observation 1: In all cases, our proposed flow leads to smaller runtime (speedups > 1.0) compared to running the application on the original embedded processor (SW_{only}). The average overall speedup for all of the applications ranged between 1.4 to 1.7×. Observation 2: As expected, the more FUs the hardware accelerator has, the larger the speedup is as more resources can be shared. From table 2 we can observe that system S4 has 8 sharable adders and 8 multipliers and thus, also leads to the largest speedups, on average 1.7×. On the other side system S1 only has 1 sharable adder and 1 multiplier leading to an average speedup of only 1.4×. Observation 3: Running the application on the dedicate HW accelerator leads to obviously the best results with speedups ranging from 25× to 35× and significant energy reductions. Observation 4: The energy reduction obtained by our method is relatively modest in three of the systems (S1-S3) while leading to larger average energy in the S4 case. This is mainly because of the size of the accelerator. The larger the accelerator is the larger the overall power consumption is. This is only partially compensated with the runtime reduction. Much better results could be obtained if the rest of the unused accelerator could be *turned off*.

Table 4: Avg. area and delay overheads introduced by PEPA

Metric	System				Avg.
	S1	S2	S3	S4	
Area [%]	10.86	6.72	7.22	4.82	7.10
Delay [%]	1.27	0	0	2.47	1.78

Table 4 measures the overheads associated with our proposed flow in terms of area and delay. Average values are shown here when all of the benchmarks given in table 3 are mapped on the different systems. Adding the sharing logic and additional control structure to execute the VLIW instructions leads on average to 7.1% additional area used. In terms of delay, it increases by 1.78% on average, while in two cases there is not delay increase because the additional logic inserted by our method was not in the critical path. It should be noted that in all cases the target synthesis frequency of 250MHz was met after logic synthesis.

Table 5: PEPA runtime overview

System		Runtime [s]			
		S1	S2	S3	S4
PEPA	Phase1	0.1	0.1	0.2	0.2
	Phase2	15.42	12.74	7.79	4.86
Total		15.43	12.75	7.81	4.88

Finally, table 5 reports the running time of our proposed flow reporting the runtime of each phase individually (phase 1 and phase 2) and the combined runtime (total). The values reported are the average values when the different applications are mapped onto the individual systems. From the results we can observe that the running times are very consistent across the different systems with phase 1 being extremely fast and phase 2 taking between 4.86s to 15.42s. The main reason for the higher runtime of phase 2 is the generation VLIW instructions which requires grouping individual

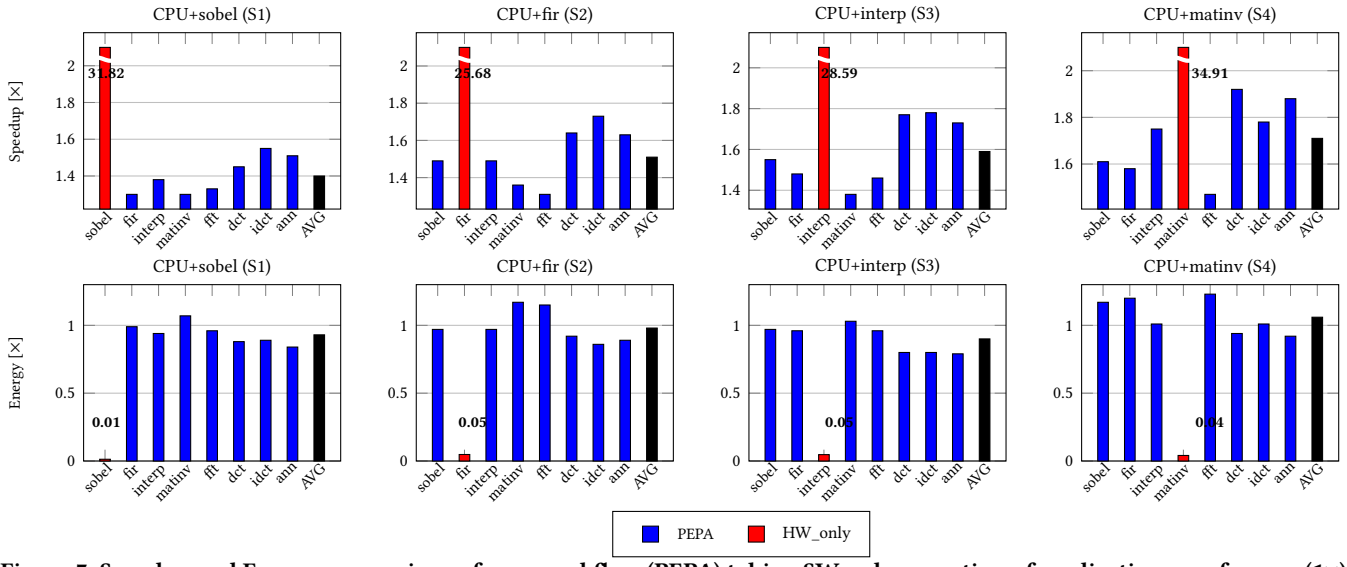

Figure 7: Speedup and Energy comparison of proposed flow (PEPA) taking SW only execution of application as reference (1×) and also running on HW accelerator (only systems shown in table 2).

instructions. We do believe that these running times are acceptable, especially considering the benefits introduced by our flow.

These results allow us to conclude that our proposed method is effective in accelerating the execution of applications re-using the hardware resources of tightly coupled hardware accelerators.

7 CONCLUSION

In this work, we have introduced a framework that leverages the hardware resources of tightly coupled hardware accelerators in order to accelerate the execution of other applications running on the processor. This is done by modifying the hardware accelerator's architecture sharing its FUs. A compiler back-end that knows the type of FUs is also presented. This compiler back-end generates new instructions for any application that can leverage these shared resources.

Experimental results show that our proposed automated flow is very effective. Future work will share a wider range of resources apart from the functional units like registers and memories and also steer the HLS process when generating the hardware accelerators to maximize resource sharing for these other applications that might be executed on the processor.

REFERENCES

[1] L. Piccolboni, P. Mantovani, G. D. Guglielmo, and L. P. Carloni, "COSMOS: Coordination of High-Level Synthesis and Memory Optimization for Hardware Accelerators," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 150:1–150:22, Sep. 2017.

[2] M. Dehyadegari, A. Marongiu, M. R. Kakoee, S. Mohammadi, N. Yazdani, and L. Benini, "Architecture support for tightly-coupled multi-core clusters with shared-memory hw accelerators," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2132–2144, 2015.

[3] Y. Janin, V. Bertin, H. Chauvet, T. Deruyter, C. Eichwald, O.-A. Giraud, V. Lorquet, and T. Thery, "Designing tightly-coupled extension units for the stxp70 processor," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013, pp. 1052–1053.

[4] A. Boroumand *et al.*, "Google workloads for consumer devices: Mitigating data movement bottlenecks," *SIGPLAN*, vol. 53, no. 2, p. 316–331, Mar. 2018.

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Morgan Kaufmann, 2019.

[6] M. Gries and K. Keutzer, "Building ASIPs: The Mescal Methodology." Springer, 2005.

[7] L. Zhang, S. Li, Z. Yin, and W. Zhao, "A Research on an ASIP Processing Element Architecture Suitable for FPGA Implementation," in *International Conference on Computer Science and Software Engineering*, vol. 3, 2008, pp. 441–445.

[8] Synopsys, "ASIP designer," 2022. [Online]. Available: https://www.synopsys.com/dw/ipdir.php?ds=asip-designer

[9] Cadence, "Tensilica processor," 2022. [Online]. Available: https://www.cadence.com/en_US/home/tools/ip/tensilica-ip.html

[10] S. Parameswaran, N. Cheung, and S. L. Shee, "Novel architecture for loop acceleration: a case study," in *2005 Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'05)*, 2005, pp. 297–302.

[11] L. Liu, Z. Yang, S. Li, and M. Yan, "Implementation of high-throughput fft processing on an application-specific reconfigurable processor," in *Proceedings of 2012 2nd International Conference on Computer Science and Network Technology*, 2012, pp. 1284–1288.

[12] D. Rossi, C. Mucci, M. Pizzotti, L. Perugini, R. Canegallo, and R. Guerrieri, "Multicore signal processing platform with heterogeneous configurable hardware accelerators," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 9, pp. 1990–2003, 2014.

[13] P. D. Schiavone, D. Rossi, A. Di Mauro, F. K. Gürkaynak, T. Saxe, M. Wang, K. C. Yap, and L. Benini, "Arnold: An efpga-augmented risc-v soc for flexible and low-power iot end nodes," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 4, pp. 677–690, 2021.

[14] Comet RISC-V. (2022). [Online]. Available: https://gitlab.inria.fr/srokicki/Comet

[15] S. Rokicki *et al.*, "What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications," in *ICCAD 2019*. IEEE, 2019, pp. 1–8. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02303453

[16] Y.-T. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 12, pp. 1477–1487, 1997.

[17] M. Lam, "Software pipelining: An effective scheduling technique for vliw machines," vol. 23, no. 7, p. 318–328, jun 1988.

[18] M. V. Eriksson and C. W. Kessler, "Integrated modulo scheduling for clustered vliw architectures," in *High Performance Embedded Architectures and Compilers*, A. Seznec, J. Emer, M. O'Boyle, M. Martonosi, and T. Ungerer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 65–79.

[19] C. Lee, J. K. Lee, T. Hwang, and S.-C. Tsai, "Compiler optimization on vliw instruction scheduling for low power," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 8, no. 2, p. 252–268, apr 2003.

[20] S. Haga, A. Webber, Y. Zhang, N. Nguyen, and R. Barua, "Reducing code size in vliw instruction scheduling," *J. Embedded Comput.*, vol. 1, no. 3, p. 415–433, aug 2005.

[21] B. Carrion Schafer and A. Mahapatra, "S2CBench:Synthesizable SystemC Benchmark Suite," *IEEE Embedded Systems Letters*, vol. 6, no. 3, pp. 53–56, 2014.