A Low-Cost BRAM-Based Function Reuse for Configurable Soft-Core Processors in FPGAs

Pedro H. Exenberger Becker[1](✉), Anderson L. Sartor[1], Marcelo Brandalero[1], Tiago Trevisan Jost[1], Stephan Wong[2], Luigi Carro[1], and Antonio C. Beck[1]

[1] Institute of Informatics, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
{phebecker,alsartor,mbrandalero,ttjost,carro,caco}@inf.ufrgs.br

[2] Computer Engineering Laboratory, Delft University of Technology, Delft, The Netherlands
J.S.S.M.Wong@tudelft.nl

Abstract. Many modern FPGA-based soft-processor designs must include dedicated hardware modules to satisfy the requirements of a wide range of applications. Not seldom they all do not fit in the FPGA target, so their functionalities must be mapped into the much slower software domain. However, many complex soft-core processors usually underuse the available Block RAMs (BRAMs) when comparing to LUTs and registers. By taking advantage of this fact, we propose a generic low-cost BRAM-based function reuse mechanism (the BRAM-FR) that can be easily configured for precise or approximate modes to accelerate execution. The BRAM-FR was implemented in HDL and coupled to a configurable 4-issue VLIW processor. It was used to optimize different applications that use a soft-float library to emulate a Floating-Point Unit (FPU), and an image processing filter that tolerates a certain level of error. We show that our technique can accelerate the former by 1.23x and the latter by 1.52x, with a Reuse Table that fits in the BRAMs (that would otherwise be idle) of five tested FPGA targets with a marginal increase in the number of slice registers and LUTs.

Keywords: FPGAs · Soft-core processors · Function reuse Approximate

1 Introduction

The implementation of soft-core processors in Field-Programmable Gate Arrays(FPGA) has known benefits such as architecture customization, hardware acceleration, and obsolescence mitigation [1]. These processors have gained space in solutions to specific purpose problems: by using modules that can be configured at synthesis time, they combine the ease of high-level programming for end

N. Voros et al. (Eds.): ARC 2018, LNCS 10824, pp. 499–510, 2018.
https://doi.org/10.1007/978-3-319-78890-6_40

users with performance gains in dedicated tasks. Because many of them require high performance for a wide range of applications, specific hardware like Floating Point Units(FPUs), security and cryptography modules, and coders/decoders for multimedia commonly surround the processor (e.g., Multiprocessor Systems on Chip(MPSoCs)) [2]. However, FPGA designs require more area and energy compared to Application Specific Integrated Circuits(ASICs) [3]. Therefore, in many cases, the resources available in an FPGA will be a limiting factor. In case specific hardware cannot fit inside the FPGA, its functionality must be mapped into the software domain, which is significantly slower.

However, there is one class of resources in FPGAs that is often underutilized when implementing complex logic driven designs: Block Random Access Memories(BRAMs). For instance, in the *OpenSparc T1* (a single-issue, six-stage pipeline that supports up to four concurrent threads), BRAMs are not utilized in the same proportion as registers and Look-Up Tables(LUTs). This comes from the observation that BRAMs usually present a limited number of ports (in most cases, two for reading and one for writing). This feature may forbid many possible uses for BRAMs: for example, the register file in multiple-issue processors usually need multiple read ports to feed all the available functional units adequately [4]. Hence, BRAMs are typically used only to implement moderately sized caches, common in the scope of soft-cores running in embedded environments.

Considering this scenario, this paper proposes BRAM-FR: a function reuse-based technique that leverages those idle BRAMs, resulting in a low-cost and generic hardware solution to speed up specific software parts without the need for implementing dedicated hardware components. Each time a function executes, its results are dynamically stored in a BRAM Reuse Table (RT) and, when the same function with the same input arguments is called again, the output can be directly fetched, avoiding re-calculation and improving performance. Going one step further, we also show that, by tuning how the RT is accessed, it is possible to gracefully switch, by using the same hardware structure, from precise to approximate reuse, which can significantly increase reuse rates and performance at an expense of output quality in some specific classes of applications. Therefore, the proposed reuse mechanism exploits BRAMs that would otherwise be idle to optimize the execution of any given software library, avoiding its ASICs counterpart implementation, which results in significant savings in design time, LUTs and registers.

BRAM-FR was coupled to a complex configurable 4-issue Very Long Instruction Word (VLIW) soft-core at Hardware Description Language (HDL). We investigate six applications that process a significant amount of Floating Point (FP) operations in different scenarios, including one where implementing a FPU in hardware would prevent the inclusion of any new dedicated hardware because of the limited amount of available resources. In this case, BRAM-FR is used to optimize a soft-float library that uses integer units to emulate double precision FP operations. We also evaluate an image processing filter software that tolerates a certain error level, showing that one can switch to approximate mode and trade-off performance and quality.

We demonstrate that an average speedup of 1.23x in the precise mode and 1.52x in the approximate one is achieved when considering an RT that fits in five different test targets. For targets with larger BRAMs, this number can be as high as 1.38x and 2.97x, respectively. Meanwhile, the usage of slice registers and slice LUTs by our generic reuse mechanism increases by 17% and 3% respectively, compared to 140% and 48% for an FPU or 11% and 13% for a dedicated Sobel filter. It is important to note that BRAM-FR is generic, so its cost in registers and LUTs is fixed regardless the number of different applications that it can optimize.

The upcoming sections are organized as follows. Related work about different reuse approaches is covered in Sect. 2. Section 3 discusses the implementation and the particularities of BRAM-FR. Results are presented and discussed in Sect. 4. Section 5 states conclusions and future work.

2 Related Work

Many works have discussed reuse of computation [5]. Implementations vary from software (also known as memoization [6]) to hardware-based solutions, covering different granularities of code. In [7], dynamic instruction reuse is presented with execution-driven simulation. The goal is to avoid re-execution of instructions in an out-of-order processor. In this case, instructions' source registers are the inputs, and its result is the output. Authors in [8] proposed the reuse of FP instructions focusing on multimedia applications, considering only those that take more than one cycle to execute. Average speedup between 1.08x and 1.22x is achieved. Despite a hardware scheme being discussed, the results are taken from an instruction-level simulator.

Going a step further, [9] considers reuse of basic blocks, simulating with SimpleScalar. The source operands (registers or memory) of each instruction inside a basic block are considered as part of the input, while the values written to any register or memory location are considered as part of the output. Their work shows performance improvements of up to 1.14x. A similar system is proposed at trace level (a set of sequential basic blocks) in [10]. In this case, less reusability is found compared to instruction reuse, but more speedup is obtained since larger chunks of code are involved.

The authors in [11] introduced the concept of dynamic function reuse. In this case, only pure functions (which do not use global variables and make no I/O operations, so the global state of the program is not altered) can be reused, so that the return value depends only on the function's input parameters. The study presented 10% to 67% of reusability on a variety of applications. Finally, the authors in [12] implemented function reuse with a mechanism that intercepts calls to the dynamically linked math library. This modified library verifies reusability and returns the respective output value by reuse when available (otherwise, the original math library is called to solve the function).

A few works have explored the concept of approximate function reuse under distinct names. In [13], fuzzy memoization of FP instructions is presented. Similarly to the work developed in [8], only multiplication and division operations are saved in the table due to their high latency, and multimedia applications are used for evaluation. Approximation is achieved by discarding some Least Significant Bits(LSBs) from the input FP value's mantissa, causing close enough values to be grouped into the same table entry. The authors claim that 4x more energy can be saved by using fuzzy memoization compared to the precise reuse approach. Work in [14] presents the clumsy value cache, an instruction/block-level reuse technique targeting (GPUs) fragment shaders. The authors investigate the potential of dropping input bits to increase instruction reuse rates and show that by doing so is the only viable way to implement block reuse. No speedup results are presented in the work, but the technique reduces the amount of instructions executed, on average, by 13.5%. Dropping input bits is also assessed with approximate function reuse in software [15], where the authors achieve 50% reuse rate with less than 10% quality degradation in image benchmarks. The authors in [16] use memoization to accelerate application-specific circuits synthesized for FPGA using High-Level Synthesis(HLS), and show that it can achieve 20% energy savings with less than 5% of area overhead.

Differently from previous works, BRAM-FR specifically considers FPGAs and configurable soft-cores, taking into account their unique components, design constraints and intrinsic characteristics, such as the fact that BRAMs are usually underused. It can provide a generic solution for both precise and approximate computation, delivering a low-cost and flexible technique so the design requirements can be achieved with the FPGA at hand. To the best of our knowledge, this is the first hardware implementation of such technique targeted towards soft-core processors. Through this, this work provides an in-depth analysis of the area/resources consumption of the mechanism, and a level of accuracy that only actual implementations can provide. Our hardware implementation is free of any abstraction layers, leading to a solution independent of user space or operating systems, which are unavailable in bare metal designs. By presenting function reuse in FPGA for configurable soft-core processors, we open new possibilities for design space exploration and new tradeoffs for HW/SW co-design in such devices. For instance, low-price FPGAs may regain space in project decision, since our approach provides performance gains with low overhead in LUTs, occupying, instead, BRAMs that would otherwise be idle.

3 Implementation

BRAM-FR is implemented through a function Reuse Unit (RU) composed of the following:

– *Reuse Table (RT)*: a direct mapped table implemented in BRAM that stores dynamic information of *reusable functions* (frequently executed, likely-to-be-reused pure functions defined at design time). Each entry (Fig. 1) contains the function's address and the input (or a tag, in case it is approximate, as it will be further explained) and output contexts. Its size is defined at design time.

- *Functions Table*: a small (one entry per reusable function) and fully associative table with static information on the reusable functions. Each entry contains the function's address, execution mode (precise or approximate), number of parameters of the function, and number of the input bits for quality control (in the case it is configured to be approximate).
- *Reuse mechanism*: implements the process of accessing the reuse table, which involves the index calculation (using a hash); checking whether the entry in the RT is valid or not; and reusing it, if it is the case.

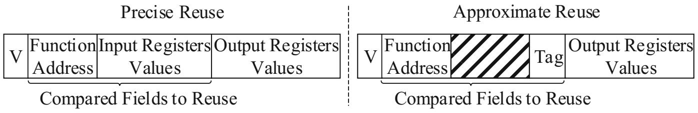

Fig. 1. Difference in RT structure between precise and approximate modes.

The RU is generic (can potentially be used with any application containing pure functions) and can switch between modes at runtime to perform both precise and approximate reuse.

3.1 Baseline Processor

BRAM-FR was coupled to the ρ-VEX VLIW soft-core processor [17] (a 32-bit five-stage pipeline, compatible with the VEX Instruction Set Architecture (ISA) [18], described in VHDL and configurable at design time), even though there are no restrictions whatsoever that would prevent its implementation to any other soft-core processor. In this work, we used the default 4-issue version consisting of 4 ALUs, 2 multipliers, 1 memory unit, and 1 branch unit (as shown in Fig. 2) and $8+8$ KB instruction and data caches. The VEX ISA defines that argument and return values for function calls are passed through registers *R3* to *R10*. If more than eight input or output registers are required, the memory is used. BRAM-FR considers only the first case (up to eight parameters) since we have found that the number of functions that do not fit in this case is not significant.

3.2 Reuse Mechanism

Figure 2 details the ρ-VEX organization integrated with the RU. Three phases are highlighted and correspond to (1) how the RU collects reuse information, (2) verifies and stores reuse information, (3) and applies reuse (when possible). Precise and approximate reuse use the very same hardware structure and differ only in the way they access the RT during Phase 2. Algorithm 1 details the three phases above implemented by the RU, which will be further discussed next.

Phase 1: When the pipeline decodes a *call*, the function and return addresses are captured by the RU, which checks in the *Functions Table* if the function is defined as *reusable* (*l. 3–5* in Algorithm 1). If so, it also fetches reuse information

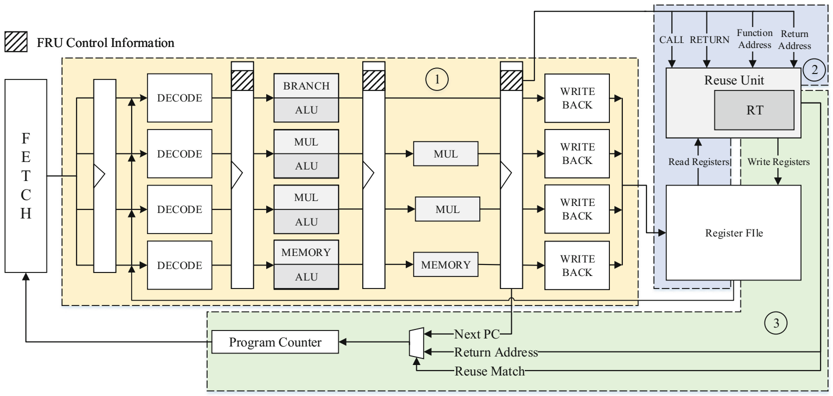

Fig. 2. Organization of a 4-issue ρ-VEX with a Reuse Unit.

(i.e., which input/output registers, whether the function was flagged for precise or approximate reuse, and the number of input bits) and goes to Phase 2. If the function is *not reusable*, the processor continues its regular operation.

Phase 2: In this phase, the current function's input parameters (the input context *ctx*) are collected by accessing the register file. The behaviour depends on whether the function was flagged for *precise* or *approximate* reuse in the *Functions Table*. In the former case, the RU generates a hash key (*l. 11–12*) by *XORing* every 16-bit of data in the current input context and function address, similarly to the approach in [12]. The resulting key's LSBs are used as the RT index to fetch a table entry (*l. 13–15*), which contains the fields shown in Fig. 1. In case the fetched entry is valid, the entry's function address and input parameters are compared with those of the current call (*l. 22–23*). If the comparisons match, there is an RT hit and phase (3) starts.

If the function was flagged for approximate reuse, the process of generating the hash key is almost the same, with one difference: some LSBs (given by the *fun.drop* field) is dropped before computing the hash to group close enough values (*l. 8–9*). With the resulting key generated, an entry in the RT is fetched (see Fig. 1). Then, the current call's function address and hash key are compared against the function address and hash key (tag) of the fetched RT entry (*l. 29–30*). The trade-off between reuse rates and quality can be easily tuned by changing the number of input bits to be discarded (*l. 9*).

For both cases (precise or approximate), a reuse miss happens if the valid bit of the current entry is not set or if the function address and inputs/tag do not match. In these cases, the RU waits for the function to execute regularly and then, with the input context (or hash value, if approximate) and outputs captured from the register file, stores a new entry (if the valid bit was not set, *l. 16–21*) or replace an entry in the RT (in case of data mismatch, *l. 26–28* and *33–35*). Therefore, the RT is dynamically filled as the program executes.

It is important to highlight that the same hardware, controlled only by a small set of multiplexers, and the same table (but with a slight change in how its data is structured) are used for both precise and approximate implementations. The way how RU is accessed from the Functions Table is what defines the mode, so both are available during the execution.

Phase 3: A match was detected in the previous phase, so the result of the whole function is available in the fetched RT entry. Therefore, the RU writes it to the register file, skipping the actual execution (*l. 24 and 31*), and notifies the reuse detection to the processor. Then, the instructions in the pipeline are flushed, and the return address (captured by the RU in Phase 1) is written to the program counter (*l. 25 and 32*). Since reusability can be checked before the pipeline commits any instruction, no rollback mechanism is required.

The register file was modified with the addition of extra reading ports (with marginal area impact, as our results will show). With this modification, there are no stalls in the pipeline when fetching the input parameters or the output values. The number of additional ports can be tuned according to the target functions: the more parameters, the more ports are needed to ensure no pipeline stalls (in our implementation, four reading ports were added). As already mentioned, when the reuse is applied, the pipeline is flushed, and the result of the function is written to the register file. The RU exploits the fact that the write ports of the register file would be idle due to the pipeline flush and uses them to perform this operation, so no additional write ports are necessary.

Algorithm 1. Algorithm Implemented by the Reuse Unit.

```
Require: Function address ($addr$), return address ($raddr$), function context ($ctx$).

 1: while program is executing do
 2:     if function CALL instruction then
 3:         $fun \leftarrow FunctionsTable.srch(addr)$
 4:         if $fun.reusable$=false then
 5:             continue
 6:         if $fun.precise$ then
 7:             $ictx \leftarrow ctx$
 8:         else if $fun.approx$ then
 9:             $ictx \leftarrow ctx$, dropping $fun.drop$ LSBs
10:         $h \leftarrow 0$
11:         foreach 16-bit words $w$ in {$addr$,$ictx$} do
12:             $h \leftarrow h \oplus w$
13:         $n \leftarrow \log_2(RT.size)$
14:         $i \leftarrow h[n-1:0]$
15:         $e \leftarrow RT.fetch(i)$
16:         if $e.valid$ = false then
17:             $octx \leftarrow Execute(fun)$
18:             if $fun.precise$ then
19:                 $RT.update(fun, i, ictx, octx)$
20:             else if $fun.approx$ then
21:                 $RT.update(fun, i, h, octx)$
22:         else if $fun.precise$ then
23:             if $e.addr$=$addr$ and $e.ictx$=$ictx$ then
24:                 $ReuseFunction(fun, e)$
25:                 $WritePC(raddr)$
26:             else
27:                 $octx \leftarrow Execute(fun)$
28:                 $RT.update(fun, i, ictx, octx)$
29:         else if $fun.approximate$ then
30:             if $e.addr$=$addr$ and $e.h$=$h$ then
31:                 $ReuseFunction(fun, e)$
32:                 $WritePC(raddr)$
33:             else
34:                 $octx \leftarrow Execute(fun)$
35:                 $RT.update(fun, i, h, octx)$
```

4 Results

4.1 Methodology

The soft-float library (case-study for precise function reuse), which emulates FP operations using integer hardware, was statically linked at compile time. We considered the four basic operations (add, sub, mult, div) in double FP precision as the *reusable functions* and evaluated the speedup of six benchmarks: five from the WCET benchmark suite [19]: *lms, ludcmp, minver, qurt, st*; and one from the Powerstone suite [20]: *fir*. The *sobel* image-processing filter (case-study for approximate function reuse) from the AxBench suite [21] was evaluated using 30 distinct images. In this case, only the convolutional kernel was considered as reusable. The benchmarks were compiled with LLVM [22] using the -O3 flag and cycle-accurate simulations were carried out using Mentor Graphics Modelsim 10. To measure performance, we compared the execution cycles of the benchmarks on ρ-VEX with and without the RU (either in precise or approximate mode), experimenting with RT sizes varying from two to 32K lines. We collected FPGA resource usage and timing information after synthesizing and mapping the VHDL of the processor to five FPGA targets from Virtex 4 (xc4vlx40; xc4vsx55), 5 (xc5vsx50t; xc5vsx95t), and 7 (xc7vx690t) Series, optimizing for area and using Xilinx ISE 14.7. Adding BRAM-FR to the design caused no changes in the critical path. Finally, the error metric used to assess the sobel benchmark's output quality was the root-mean-squared (RMS) pixel difference between the original and approximated computations, normalized to the range 0–100%, as defined in the AxBench suite [21].

4.2 Performance

Precise Reuse - Soft-Float: Figure 3 presents the reusability of the soft-float functions. Due to space limitations, we show only a subset of all RT sizes evaluated, including extrapolated results for the best case (when the RT is increased to a point when no replacements take place).

Grouped by benchmark, each column depicts the stacked RT hit rate of *add*, *sub*, *mul*, and *div* functions, according to the number of RT lines (x-axis). Naturally, reusability increases with the RT size, since more reuse information is available so a match attempt succeeds. Cases with significant reusability of *div* (e.g., *fir*) and *mul* (e.g., *lms* and *minver*) have more potential for improving performance, since these operations take longer than *add* or *sub*. For these

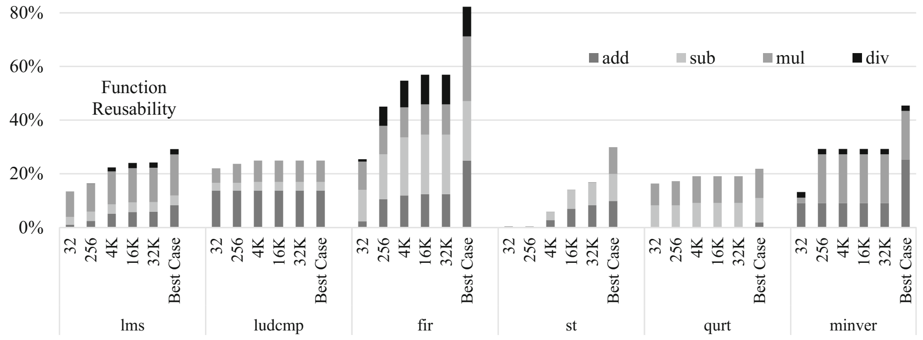

Fig. 3. Reusability of FP functions.

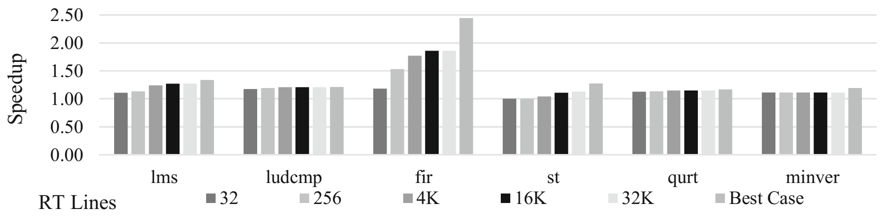

Fig. 4. Speedup for different RT sizes compared to the baseline.

benchmarks, reusability can vary from almost none (*st* with 32 lines) to more than 80% (*fir* in the best case).

Figure 4 depicts the speedup for different RT sizes, also grouped by benchmark. For the largest RT tested (32K lines) significant speedups are achieved: in *ludcmp* (1.21x), *lms* (1.28x) and *fir* (1.86x); and even in the worst cases (*minver*, 1.12x and *st*, 1.13x). When the RT is reduced to only 32 lines, five out of six benchmarks still improve by more than 1.1x (the only exception is *st*, given its small reuse rates). Also, for most benchmarks, note how a 16K or even a 4K-line RT already improves performance near to the theoretical maximum (the *best case*) of the technique. Comparing to a 4K-line RT, for example, the best case brings no improvement in *ludcmp*, and increases *qurt* and *lms* performance only marginally, by 2% and 7%, respectively. This fact highlights that function arguments very often repeat during execution and present limited variability. The exception is *fir*, in which the reuse rates increase significantly with larger table sizes. In the best case, its performance improves by 2.44x compared to the baseline (and 67% more than the 4K-line RT).

Considering the geomean speedup for the six benchmarks, considerable gains in performance can be achieved with low resource overhead. For example, 32-line and 256-line RTs can provide speedups of 1.12x and 1.18x, respectively. A 4K-line RT, which still fits in all five tested FPGAs (which will be discussed in subsection Resource Usage) provides speedups of 1.23x.

Therefore, even resource-limited FPGAs can benefit from the BRAM-FR. When it comes to high-end FPGA, the available BRAMs can be used to increase even more the RT size and consequently get closer to the maximum speedup possible for applications with high reusability rates.

Approximate Reuse - Sobel Image Filter: Figure 5 shows geomean speedup and error rate considering 30 distinct images, in an approximation scenario where 4 LSBs are dropped from the inputs values in the *sobel* benchmark. Modifying this value leads to distinct performance-error trade-offs, so we constrained ourselves to only one representative spot in the vast design space available, chosen after comprehensive experimentation.

The great benefit from approximate reuse is that speedup is achieved more easily than with precise reuse. Note that a 2-line RT, in this case, improves performance by 1.33x, with only 3% error. Error remains under 10% (common error threshold for approximate *sobel* benchmark [21]) for every table up to 4K

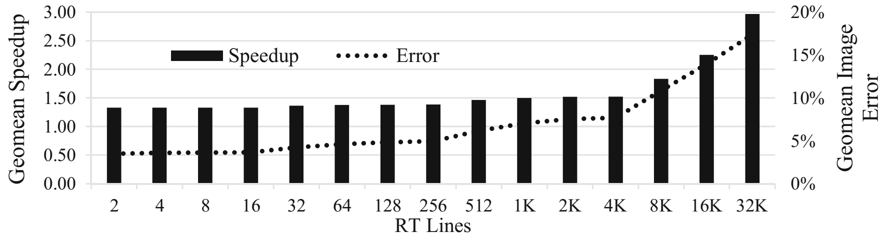

Fig. 5. Speedup for approximate sobel image filter.

lines, where 8% error meets 1.52x speedup. Since higher speedups mean that more reuse was possible, it also means that more errors will appear (since exact values will be exchanged for approximated ones). If higher error rates can be tolerated, the speedup can reach 2.25x with 14% of error (for a 16K line RT), or even 2.97x with 17% of error (32K line RT).

Resource Usage: We collected the usage of BRAM, Slice Registers, and Slice LUTs in four scenarios: baseline (ρ-VEX), ρ-VEX with RU, ρ-VEX with a double precision FPU [23], and ρ-VEX with a hardware sobel filter. They all were synthesized following the methodology explained before, with exception of the hardware implementation of Sobel, which data was taken from [24], covering Virtex 5 FPGAs only. Table 1 presents the comparison in the four scenarios with distinct targets using the largest RT that fits in each design. For example, the Virtex 5 - xc5vsx50t supports a maximum of 16K lines. Smaller tables, yet measured, were omitted. Although each table line for approximate reuse needs less information (a tag instead of all the input values, see Fig. 1) the results consider the size needed for the implementation of both modes (i.e., it considers the size for precise reuse), since it is possible to switch between them at run-time.

All targets can support an RT with at least 4K lines. In the Virtex 4 FPGA (the smallest available device), using the 4-issue ρ-VEX processor with an FPU would occupy nearly all FPGA resources (97% of the available Slice LUTs) and

Table 1. Usage of resources for different designs and targets

Series	Model	Design	Used Slice Registers	% Used Slice Registers	Used Slice LUTs	% Used Slice LUTs	Used BRAM	% Used BRAM
Virtex 7	xc7vx690t	ρ-Vex	3,015	0%	14,675	3%	16	1%
		ρ-Vex + RU (32K lines)	3,494	0%	15,176	4%	242	16%
		ρ-Vex + FPU	7,275	1%	19,926	5%	16	1%
ex 5	xc5vsx95t	ρ-Vex	3,012	5%	15,200	26%	16	7%
		ρ-Vex + RU (32K lines)	3,516	6%	15,717	27%	242	99%
		ρ-Vex + FPU	7,061	12%	23,349	40%	16	7%
		ρ-Vex + Sobel	3,351	6%	17,127	29%	16	7%
	xc5vsx50t	ρ-Vex	3,012	9%	15,200	47%	16	12%
		ρ-Vex + RU (16K lines)	3,516	11%	15,717	48%	129	98%
		ρ-Vex + FPU	7,061	22%	23,349	72%	16	12%
		ρ-Vex + Sobel	3,351	10%	17,127	52%	16	12%
Virtex 4	xc4vsx55	ρ-Vex	3,008	6%	23,986	49%	32	10%
		ρ-Vex + RU (16K lines)	3,511	7%	24,820	50%	258	81%
		ρ-Vex + FPU	7,403	15%	35,888	73%	32	10%
	xc4vlx40	ρ-Vex	3,008	8%	23,986	65%	32	33%
		ρ-Vex + RU (4K lines)	3,511	10%	24,820	67%	89	93%
		ρ-Vex + FPU	7,403	20%	35,888	97%	32	33%

restrict the addition of other hardware accelerators or even the modification of the issue-width (e.g., increase to the 8-issue version). As the original ρ-VEX uses a minimal amount of the available BRAMs, the RT can occupy the remaining ones as much as possible, leveraging these idle components which neither the FPU nor the Sobel hardware could exploit. In some cases, even an RT larger than 3K lines could be used (e.g., it only occupies 15% of the Virtex 7's BRAMs).

As for the logic resources that are usually scarce (slice registers and LUTs), we introduce a small overhead of 17% and 3%, respectively. In contrast, adding an FPU to ρ-VEX more than doubles the number of registers (140% overhead) and significantly increases LUTs usage (48%). The sobel hardware, likewise, increases by 11% the slice registers and 13% the slice LUTs. However, while these units are application-specific and are incremental in terms of resources (i.e. more LUTs and registers are necessary for each new application-specific hardware that is integrated), the overhead in LUTs and registers of our generic design is fixed, being only the RT variable (and thus BRAM usage). Therefore, costs can be amortized as the BRAM-FR encompasses more system features, enabling performance gains when any new robust hardware module does not fit.

Therefore, BRAM-FR can benefit both low and high-end FPGAs: in the former, the reuse mechanism allows performance improvements with minimum hardware overhead. In the latter, not only more hardware accelerators but also extra processors could be integrated into the system. For instance, three cores of the ρ-VEX processor could be instantiated alongside the RU in the Virtex 5 - xc5vsx95t. This would not be possible if a FPU was implemented in hardware.

5 Conclusions and Future Work

This work proposed a new function reuse approach as an alternative to logic-costly specific hardware in soft-core designs. We showed that it is possible to improve software libraries that substitute specialized hardware, using highly available BRAMs. Our scheme is able to perform precise and approximate reuse using much less logic than specific hardware, making low-cost targets to regain space in the design space. As future work, we will consider a multi-core environment, providing reusability for multiple programs at similar hardware cost.

Acknowledgments. This work was supported in part by CNPq, CAPES and FAPERGS.

References

1. Fletcher, B.H.: FPGA embedded processors. In: Embedded Systems Conference. p. 18 (2005)
2. Beck, A.C.S., Lisbôa, C.A.L., Carro, L.: Adaptable Embedded Systems. Springer, New York (2012). Springer-Link: Bücher
3. Kuon, I., Rose, J.: Measuring the gap between FPGAs and ASICs. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. **26**(2), 203–2015 (2007)

4. Xilinx, Inc.: 7 Series FPGAs Memory Resources User Guide (UG473) (2016)
5. Sastry, S.S., Bodik, R., Smith, J.E.: Characterizing coarse-grained reuse of computation. In: Feedback Directed and Dynamic Optimization, pp. 16–18 (2000)
6. Hall, M., McNamee, J.P.: Improving software performance with automatic memoization. Johns Hopkins APL Tech. Dig. **18**(2), 255 (1997)
7. Sodani, A., Sohi, G.S.: Dynamic instruction reuse. In: Proceedings of 24th Symposium on Computer Architecture (ISCA), vol. 25, no. 2, pp. 194–205 (1997)
8. Citron, D., Feitelson, D., Rudolph, L.: Accelerating multi-media processing by implementing memoing in multiplication and division units. ACM SIGPLAN Not. 33(11), 252–261 (1998)
9. Huang, J., Lilja, D.J.: Exploiting basic block value locality with block reuse. In: Proceeedings of Symposium on High-Performance Computer Architecture, pp. 106–114. IEEE (1999)
10. González, A., Tubella, J., Molina, C.: Trace-level reuse. In: International Conference on Parallel Processing, pp. 30–37. IEEE (1999)
11. Kavi, K.M., Chen, P.: Dynamic function result reuse. In: Proceedings of Conference on Advanced Computing, pp. 17–20 (2003)
12. Suresh, A., Swamy, B.N., Rohou, E., Seznec, A.: Intercepting functions for memoization: a case study using transcendental functions. ACM Trans. Archit. Code Optim. (TACO) **12**(2), 18 (2015)
13. Alvarez, C., Corbal, J., Valero, M.: Fuzzy memoization for floating-point multimedia applications. IEEE Trans. Comput. **54**(7), 922–927 (2005)
14. Keramidas, G., Kokkala, C., Stamoulis, I.: Clumsy value cache: an approximate memoization technique for mobile GPU fragment shaders. In: Workshop On Approximate Computing, P. 6 (2015)
15. Brandalero, M., da Silveira, L.A., Souza, J.D., Beck, A.C.S.: Accelerating error-tolerant applications with approximate function reuse. Sci. Comput. Program. (2017)
16. Sinha, S., Zhang, W.: Low-power FPGA design using memoization-based approximate computing. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **24**(8), 2665–2678 (2016)
17. Wong, S., van As, T., Brown, G.: ρ-VEX: a reconfigurable and extensible softcore VLIW processor. In: Conference on Field-Programmable Technology, pp. 369–372 (2008)
18. Hewlett-Packard Laboratories: VEX Toolchain (2009)
19. Gustafsson, J., Betts, A., Ermedahl, A., Lisper, B.: The Mälardalen WCET benchmarks: past, present and future. In: WCET, vol. 15, pp. 136–146 (2010)
20. Scott, J., Lee, L.H., Arends, J., Moyer, B.: Designing the low-power M$^{\bullet}$CORE$^{\text{TM}}$ architecture. In: Power Driven Microarchitecture Workshop, pp. 145–150 (1998)
21. Yazdanbakhsh, A., Mahajan, D., Esmaeilzadeh, H., Lotfi-Kamran, P.: AxBench: a multiplatform benchmark suite for approximate computing. IEEE Des. Test **34**(2), 60–68 (2017)
22. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of Symposium Code Generation and Optimization: Feedback-Directed and Runtime Optimization, p. 75. IEEE Computer Society (2004)
23. Lungdren, D.: FPU Double VHDL (2014)
24. Chaple, G., Daruwala, R.D.: Design of Sobel operator based image edge detection algorithm on FPGA. In: International Conference on Communications and Signal Processing, pp. 788–792. IEEE (2014)