UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

PEDRO HENRIQUE EXENBERGER BECKER

Function Reuse on a Multi-Core VLIW Soft-Core Processor

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engineering

Advisor: Prof. Dr. Antonio Carlos Schneider Beck
Coadvisor: Anderson Luiz Sartor

Porto Alegre
January 2018

“Navegar é preciso,
viver não é preciso.”
— FERNANDO PESSOA

ACKNOWLEDGMENTS

If I had to do this work without any help, I would probably need a couple more years and much patience to handle a very lonely and stressful life. Fortunately, I could count on great and supportive people to develop this work, whom I now want to acknowledge.

First of all, I would like to thank my parents, Raquel and Denis, for educating and supporting me ever since I recall. Also, for my advisor Antonio and his valuable guidance, and my colleagues from the Embedded Systems Laboratory of UFRGS, especially Anderson (my co-advisor) and Marcelo, whom I consulted incessantly. Furthermore, I would like to thank my classmates, whom I found to be a free and precious source of help. Finally, I would like to thank my lovely girlfriend Marina, for her every-day support and kindness.

ABSTRACT

Modern processors contain several specific hardware modules and multiple cores to ensure performance on a wide range of applications. When a project needs a processor regarding time to market and architecture customization, FPGAs are often used as the implementation platform. However, for FPGA-based processors, those modules may not fit in the target device, because of FPGA's resource constraints, so their functionalities must be mapped into the much slower software domain. In the same way, we exploit the fact that logic-driven designs usually underuse available BRAMs, memory blocks embedded in FPGAs. As an alternative, we propose a low-cost hardware-based function reuse mechanism, which can optimize software execution. This is accomplished by saving the inputs and outputs of the most recurring functions in a BRAM-based reuse table, so they can be reused in the next function calls, skipping actual execution, and improving performance. The technique was implemented in HDL and was coupled to a 4-issue VLIW processor. As a case study, we optimized six applications that use a soft-float library to emulate a floating-point unit in software, achieving 1.23x of geomean speedup. The analysis was extended to a multi-core environment, which was done with a simulator built for this work, where we observed the benefits of sharing the reuse table among similar applications running with different inputs. Sharing the reuse table proven promising, for example, leading one application to 1.9x speedup instead of 1.25x if the reuse table was not shared. Also, we present how the mechanism can be easily enhanced, so function reuse embraces the concept of approximate computing, increasing the scenarios where its use is beneficial and achieving 1.52x speedup in single-core image filter application at the cost of output quality.

Keywords: FPGAs. Soft-core processors. Function reuse. Multi-core systems.

Reuso de Funções em Processadores Soft-Cores VLIW Multi-Núcleo

RESUMO

Processadores modernos contém vários módulos específicos e multiplos núcleos para garantir desempenho em uma variedade de aplicações. Quando um projeto de processador depende de time-to-market e customização de arquitetura, FPGAS são frequentemente utilizados como a plataforma para a implementação. Entretando, para processadores baseados em FPGAs, esses módulos podem exceder a capacidade do dispositivo alvo, dadas as limitações de recursos dos FPGAs, de maneira que algumas funcionalidades precisam ser mapeadas para o domínio de software, mais lento. Da mesma forma, exploramos o fato de que projetos lógicos geralmente sub-utilizam a quantidade disponível de BRAMs, blocos de memória embarcados nos FPGAs. Como alternativa, propomos um mecanismo de reúso em *hardware* de baixo custo que pode otimizar a execução de programas. Isso é obtido ao salvarmos os parâmetros de entrada e valores de retorno de funções recorrentes, em uma tabela de reúso implementada em BRAM de maneira que as funções possam ser reutilizadas em chamadas futuras, evitando a sua reexecução, e acelerando a aplicação. A técnica foi implementada em linguagem de descrição de hardware e adicionada à um processador *VLIW 4-issue*. Como estudo de caso, otimizamos seis aplicações que usam uma biblioteca *soft-float* para simular uma unidade de ponto-flutuante em software, atingindo 1.23x de aceleração geométrica média. A análise foi estendida para um ambiente multi-núcleo, o que foi feito com um simulador construído para este trabalho, onde observou-se os benefícios de compartilhar a tabela de reúso entre aplicações similares executando sobre dados diferentes. Compartilhar a tabela se mostrou promissor, por exemplo, levando uma das aplicações à ganhos de 1.9x em contra partida de 1.25x se a tabela não tivesse sido compartilhada. Além disso, apresentamos como o mecanismo pode ser facilmente melhorado, de maneira que o reúso de funções faça uso da computação aproximativa, aumentando o número de cenários onde o reúso de funções é benéfico e atingindo 1.52x de aceleração em uma aplicação de filtro de imagem, ao custo de qualidade.

Palavras-chave: FPGAs. Processadores soft-core. Reúso de funções. Sistemas multi-núcleo.

LIST OF ABBREVIATIONS AND ACRONYMS

ALU Arithmetic and Logic Unit.

ASIC Application Specific Integrated Circuit.

BRAM Block Random Access Memories.

CLB Configurable Logic Block.

CPU Central Processing Unit.

DSP Digital Signal Processing.

FORMOSA FunctiOn Reuse MulticOre SimulAtor.

FP Floating-Point.

FPGA Field-Programmable Gate Array.

FPU Floating-Point Unit.

FU Functional Unit.

GPP General Purpose Processor.

GPU Graphics Processing Unit.

HDL Hardware Description Language.

HLS High-Level Synthesis.

ILP Instruction-Level Parallelism.

IO Input/Output.

IOB Input/Output Block.

IP Intellectual Property.

ISA Instruction-Set Architecture.

LSB Least Significant Bit.

LUT Look-Up Table.

MPSoC Multiprocessor System on Chip.

RT Reuse Table.

RU Reuse Unit.

SMT Simultaneous Multithreading.

VLIW Very Long Instruction Word.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

The implementation of processors in Field-Programmable Gate Arrays (FPGAs), also known as soft-core processors, provides known benefits when designing a computational system. For example, the reprogrammability of FPGAs guarantees good time-to-market, fast architecture customization and integration of hardware accelerators (either by writing new modules or using off-the-shelf components), as well as obsolescence mitigation since the hardware description can be easily ported to the latest FPGAs (FLETCHER, 2005). These processors have gained space in solutions to specific purpose problems: by using modules that can be configured at synthesis time, they combine performance gains in dedicated tasks with the ease of high-level programming for end users.

At the same time, nowadays systems require high performance for a wide range of applications, which increases the demand for logic resources. The use of multi-core processors, e.g., ARM Cortex-A53 (BOPPANA et al., 2015), together with dedicated hardware like Floating-Point Units (FPUs), security and cryptography modules, and coders / decoders for multimedia, are commonly adopted in any modern design, such as in Multiprocessor Systems on Chip (MPSoCs) (WOLF; JERRAYA; MARTIN, 2008). However, FPGA designs require more area and energy compared to Application Specific Integrated Circuits (ASICs) (KUON; ROSE, 2007). Therefore, in many cases, the resources available in an FPGA will be the limiting factor. In case specialized hardware can't fit inside the FPGA, some of its features must be mapped into the software domain, which is significantly slower.

In addition, Block Random Access Memories (BRAMs) are often underutilized when implementing such complex logic driven designs in FPGAs. Table 1.1 shows the utilization of Look-Up Tables (LUTs), Registers and BRAMs of three soft-core processors (arranged in order of complexity) implemented in a Virtex-5 xc5vlx110t. As one can observe, for complex soft-core processors, such as the OpenSparc T1 (a single-issue, six-stage pipeline supporting up to four concurrent threads), BRAMs are not utilized in the same proportion as registers and LUTs. This comes from the observation that BRAMs usually present a limited number of ports (in most cases, 2 for reading and 1 for writing), which forbids many possible uses for them, such as the register file in multiple-issue processors, which need multiple read ports to properly feed all the available functional units (XILINX; INC, 2016). Hence, BRAMs are usually used only to implement moderate size caches, common in the scope of soft-cores running in embedded environments.

Table 1.1: Resource Utilization of Three Soft-Core Designs

Design	% Slice LUT	% Slice Register	% BRAM
OpenRisc1200	5%	2%	7%
Leon 3	27%	16%	15%
OpenSparc T1	88%	56%	40%

Considering this scenario, this work proposes 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 of this function can be directly fetched from the RT, avoiding re-calculation and improving performance.

Additionally, considering the consolidation of multi-core environments, we extend our reuse concept within a soft-core multi-processor design, where it is possible to share the RT among cores. Thus, programs that are simultaneously running on the soft-core can all update the RT as they calculate function results, and fetch results calculated by other processes from the RT. Since multi-thread programs (or multiples instances of a program) run over the same code, we expect that they benefit from such arrangement, increasing the reuse possibilities. At the same time, the introduction of a shared RT is cheaper than if we would introduce a dedicated RT for each core, as we have a single and centralized (instead of replicated) control logic. By using this approach, we can accelerate multiple cores while proportionally reducing the impact of an RT in the FPGA design.

Going one step further, we also show that, by tuning how the BRAM 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 the expense of output quality in some specific classes of applications. This reuse mechanism that uses BRAMs and is configurable for both precise and approximate modes can be easily used to optimize the execution of any given software library, avoiding its hardware implementation counterpart and resulting in significant savings in design time, LUTs and registers.

We evaluate the technique by implementing it in a complex 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 hardware FPU prevents the

addition of any new dedicated hardware because of the limited amount of resources available. In this case, we apply our idea to optimize a soft-float library that uses integer units to emulate double precision FP operations that would otherwise have to be implemented in hardware. Then, with software-based simulations, we consider the execution of multiple instances of each of these benchmarks, running with different inputs, sharing the RT to optimize the same soft-float library. Over this, we can analyze the space exploration benefits in performance and area when using the reuse mechanism in a thread-collaborative way. In addition, we evaluate an image processing filter software that tolerates a certain error level, so we can switch to the approximate mode to increase reuse rates and, therefore, improve performance at the cost of quality.

For the single-core scenario, we show that an average speedup of 1.23x in the precise case 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 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 our mechanism is generic, so its cost in registers and LUTs is fixed regardless the number of different applications that it can optimize. When considering a multi-core system with a shared RT, we present cases where improvement goes from 1.25x to 1.9x, keeping the total amount of BRAM used. When BRAM is also a scarce FPGA resource, we show that our multi-core reuse approach can equate the performance of the single-core using less BRAM.

The upcoming sections are organized as follows. We present background in Chapter 2. Related work about different reuse approaches is covered in Chapter 3. Chapter 4 discusses the Implementation and Methodology of the work. Results are presented and discussed in Section 5. Section 6 states Conclusions and Future Work.

2 BACKGROUND

In the upcoming sections, we introduce some important concepts that support our study. They will be briefly defined together with its role in this work.

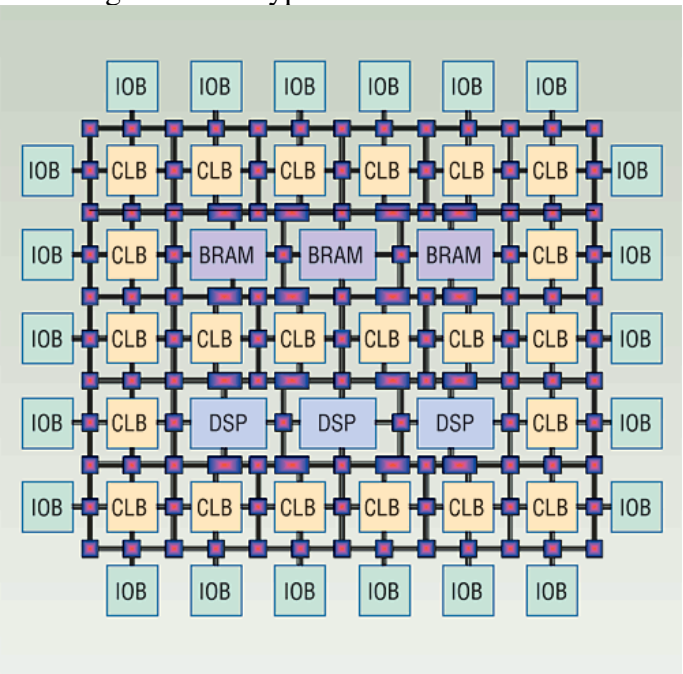
2.1 Field-Programmable Gate Array

FPGAs emerged as an intermediary solution between processors and ASICs, providing better control over the hardware implementation compared to processors, and more flexibility, testability, and time-to-market over ASIC. With its differentials, the FPGA market has reached about USD 6.36 billion by the year of 2015 with continuous growth expectation (Research Grand View, 2016). Thereby, this research is based on a technology that is already established and has a solid market.

FPGAs gained market share with its uniqueness: the idea of a reconfigurable circuit. Figure 2.1 depicts a traditional FPGA architecture. It is an array of Configurable Logic Blocks (CLBs), routing channels, BRAMs and Digital Signal Processing (DSP) units. The reconfigurability is a result of the programmability of the CLBs and routing channels, as we present below.

A CLB is the fundamental component on the FPGA architecture. Inside each

Figure 2.1: A typical FPGA Architecture.



Source: (BUELL et al., 2007)

CLB, there are a set of small tables (LUTs). With these tables, the FPGA can implement logical functions over the CLB inputs (using the LUTs as a truth-table). Thus, combining many CLBs through the routing channels can lead to implementations of very complex logical functions. Moreover, the LUTs can be used as memory elements for small data amount. Since both CLB and routing channels are programmable, the circuit that an FPGA implements can be changed by reprogramming those components.

Aside CLBs and routing channels, FPGA also has Input/Output Blocks (IOBs), to connect the FPGA with the outside world, and embedded blocks. Traditional architectures contain both embedded DSPs units, to speed up costly operations that are implemented by cascades of truth-tables (e.g., multiplications), and BRAMs, which will be detailed in the following subsection. More details can be found in (SKLYAROV et al., 2014).

2.1.1 Block Random Access Memories

BRAMs are embedded memory blocks used for storing large sets of data more efficiently than by LUTs, and are widely available in modern FPGAs. For example, the Xilinx 7 series FPGAs contain from 5 to 1880 dual-port BRAMs, depending on the model, each storing 36Kb of data. These blocks can be divided into two independent 18Kb BRAMs. In both cases, dual-port is assured, and each port is completely independent of another, sharing only the stored data (Xilinx Inc.; XILINX; INC, 2017).

These embedded blocks can also be configured in different associations (e.g., 32K 1-bit lines, 16K 2-bit lines, ..., 1K 32-bit lines, 512 64-bit lines), and can be interconnected to create wider and deeper memory structures (SKLYAROV et al., 2014). Finally, both read and write operations are synchronous, requiring an active clock edge.

As already mentioned since logic-driven designs, as soft-core processors, generally underuse available BRAMs (see Table 1.1), we propose to better occupy those components by implementing the RT for enhancing applications' performance.

2.1.2 Soft-Core Processors

A soft-core processor is a hardware description language model of a processor that can be customized and synthesized for an ASIC or FPGA target (TONG; ANDERSON; KHALID, 2006). In this study, however, we consider only FPGA-based soft-cores.

FPGA-based soft-cores became popular as they bring advantages such as (i) architecture customization, since FPGA allows non-standard implementation, according to the design requirements; (ii) obsolescence mitigation, as the hardware description perpetuates while hardware technologies advance; (iii) cost reduction, considering that multiple components can be replaced with a single FPGA; and (iv) hardware acceleration, since specific algorithms can be easily implemented in hardware, for example, to achieve better performance (FLETCHER, 2005).

There is a variety of commercial and academic soft-core processors as a demonstration of its representativeness. Examples from the industry comprehend Xilinx *MicroBlaze* soft processor, an Intellectual Property (IP) for Xilinx FPGAs (KALE, 2016), the Altera Nios II (NIOS, 2009), and the open sourced hardware description of Sun's UltraSparc T1 and T2, which were released by the OpenSparc Project (PARULKAR et al., 2008). From academia, we highlight the ρ-VEX VLIW processor (WONG; Van As; BROWN, 2008) from Delft University of Technology.

2.2 Modern Architectures

The required performance of computing devices has increased as technology advances. The transistor's scaling improved processors frequency, while the exploitation of Instruction-Level Parallelism (ILP) increased processors throughput. However, increased clock rates dissipate more power, which became a barrier. At the same time, the ILP exploitation in *superscalar* processors - where multiple independent instructions can be simultaneously executed every cycle - by deep pipelining and out-of-order execution, seems to have reached a plateau (DAS et al., 2008).

To overcome the above challenges and to guarantee computers ascendant performance, various solutions were proposed. We detail three solutions in the upcoming subsections, which are strongly related to this work: Very Long Instruction Word Processors, Multi-Core Processors, and Multiprocessor System on Chip.

2.2.1 Very Long Instruction Word Processors

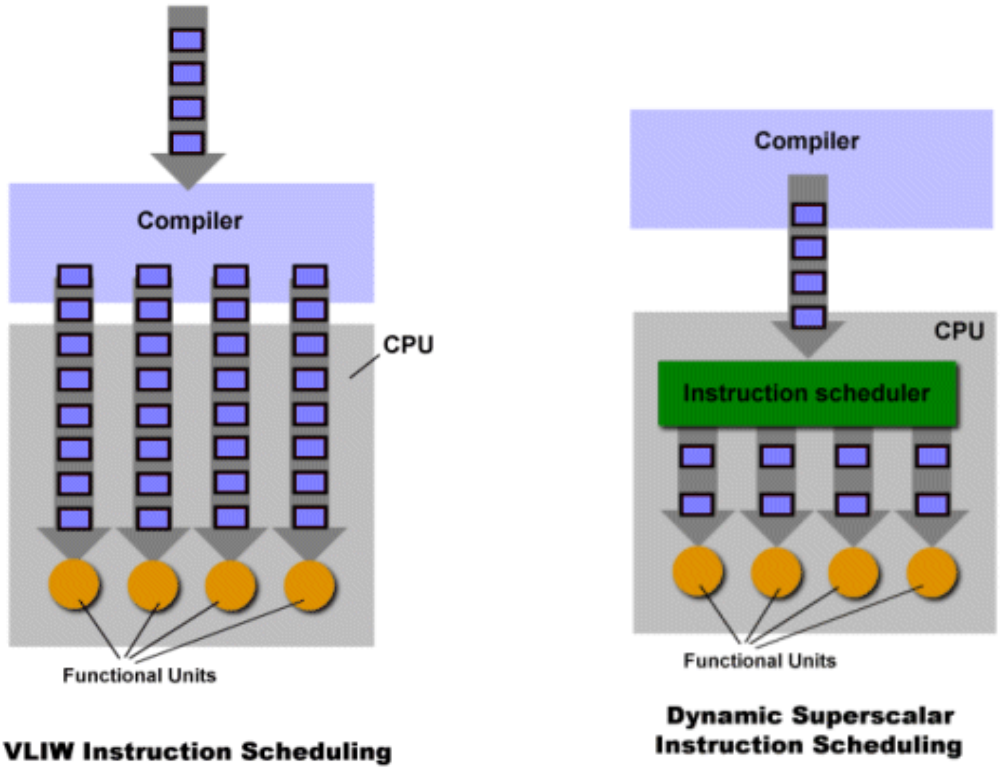
The extraction of ILP at execution time by *superscalar* processors enabled performance gains but introduced more hardware (and complexity) given the logic required to

find parallel instructions (OLUKOTUN; HAMMOND, 2005) dynamically. As alternative to on-the-fly ILP exploitation, the VLIW processors were proposed (FISHER, 1983).

A VLIW processor contains multiple execution pipelines, the issue-slots, so that it can execute more than one instruction at a time (just like a superscalar). The advantage of using these processors is that the instruction parallelism is extracted by the compiler (see Figure 2.2). Thus, a VLIW processor can benefit from ILP while maintaining a simple microarchitecture. The parallel instructions are disposed inside a *very long instruction word*, in a set of independent instructions that can be executed concurrently without any concern by the processor. The arrangement of the instruction even maps each instruction with the issue-slot on which it will be executed.

Because of its simple organization alongside its ILP exploitation, the VLIW processors are powerful yet less resource-consuming in comparison to a complex superscalar. These factors make the VLIW a good architecture option for resource-constrained FPGA-based soft-core processors.

Figure 2.2: ILP Exploitation by VLIW and Superscalar Architectures.



(a) A VLIW architecture. ILP is extracted by the compiler.

(b) A superscalar architecture. ILP is extracted by the hardware.

Source: (STOKES, 2000)

2.2.2 Multi-Core Processors

Optimizing a single core processor with pipeline, ILP exploitation and out-of-order execution became insufficient. Multi-core processors were proposed, observing that complex systems usually execute multiple tasks. Thus, in a multi-core environment, various tasks could be distributed among multiple processors, increasing the overall throughput of the system. Figure 2.3 presents a quad-core Intel i7 overview as example. Each core can execute independently of others and communicates by using the shared cache memory.

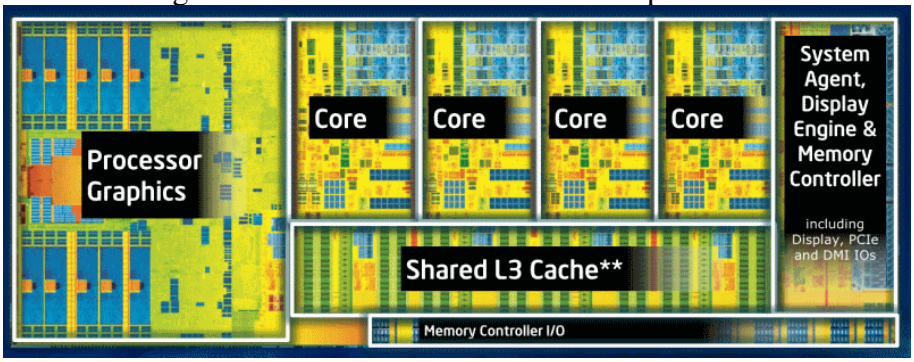
The first commercial multi-core processor was the IBM Power4 (TENDLER et al., 2002) released in 2001. After, other companies such as Intel, AMD, and Sun also turned to multi-core production (GEER, 2005). Ever since, multi-core designs became almost a standard for General Purpose Processors (GPPs), from Intel Pentium D up to the latest Intel i7 Series. Not enough, multi-core processors are highly used in nowadays embedded systems as smartphones (Van Berkel, 2009).

2.2.3 Multiprocessor System on Chip

Some computer tasks, like real-time video encoding, for example, execute a complex algorithm with very high throughput demand. In other cases, the constraints can be latency or power. When that occurs, using a GPP may not be the best option, either for performance or energy consumption, since a specialized hardware can meet the constraints more easily.

Let us consider the case of smartphones and tablets as an example. They are required to execute a broad range of concurrent tasks with minimum battery consump-

Figure 2.3: A four-core Intel i7 4770K processor.



Source: (LUDLOW, 2014)

tion. For this reason, these devices have specific hardware modules in their main chip die working together with multiple processors to run the applications efficiently (JÓŹWIAK, 2017). This chip arrangement is known as MPSoCs. For example, the Snapdragon 835, used in newest smartphones, contains a quad-core Central Processing Unit (CPU) with dedicated hardware for processing audio, graphics, camera image, security, location, and several other specialized modules (QUALCOMM, 2017).

2.3 Reuse

Here we introduce the concept of reuse in the computing scope. A more detailed overview of techniques and approaches will be covered in the Chapter 3 (Related Work).

2.3.1 General Reuse

Reuse of computation is based on the observation that deterministic execution - where a set of inputs always leads to the same result - often repeats within programs. Reuse exploits this by saving input and result (output) values of those executions in a RT. The RT is searched when reentering a given snippet of execution, checking whether the current and the saved input values are the same. In case of a hit, the result (output value) from the RT is fetched faster than recomputing it.

For example, consider the hypothetical three-address code bellow:

Figure 2.4: Example of a Reusable Code Snippet

$$r.1 = r.6$$
$$r.2 = 1 + r.1$$
$$r.3 = 1 + r.2$$

Source: The Author

The inputs are the values which are read before are written; in the example, the register $r.6$ is the only input. The outputs are the written values, which will be read further, in the case $r.1, r.2, r.3$. Consider $r.6$ = 1 in a given execution. Therefore, after the code executes, $r.1 = 1, r.2 = 2, r.3 = 3$. Note that any eventual execution in which $r.6$ = 1 will imply in the same result in the outputs $r.1, r.2, r.3$. In this scenario, saving the input and output values for consulting in a new occurrence could skip execution of three

instructions.

2.3.2 Function Reuse

A particular case for reuse is when the evaluation of reuse occurs in the grain of functions. In this case, the function parameters are the inputs, and the values returned by functions are the outputs.

The following example illustrates a simple scenario where function reuse could be used:

Figure 2.5: Example of a Reusable Function Result.

$$a = sin(\pi)$$
$$\vdots$$
$$b = sin(\pi)$$

Source: The Author

In this case, the second execution of sin could be skipped, given a repetition of the inputs (π in this case). Naturally, in this simple case, the explicit re-operation of $sin(\pi)$ could be avoided by better programming or even compiler optimization. Note, however, that in a more realistic situation the input parameters can, and often will, be variables, whose values are unknown at compile time and unpredictable by the programmer.

2.3.3 Approximate Function Reuse

When applications tolerate some error level, approximate function reuse can take place. For example, observe the scenario in Figure 2.6. In this case, we have three calls to the function sin. If we consider the former function reuse, none of the calls could benefit from previous calculations to skip the functions executions since the inputs always differ. When we perform approximate reuse, we can consider that close values are good enough for the final computation, increasing reuse possibilities, at the cost of quality.

Therefore, we must decide how much error we can accept: in the example, we can be conservative considering π very close to 3.14, but not close enough to 3. Thus we could reuse in one case, with a small loss in quality in the value of b. Being more

Figure 2.6: Example of a Reusable Function with Approximate Result.

$$
\begin{aligned}
a &= sin(3.14) \\
&\vdots \\
b &= sin(\pi) \\
&\vdots \\
c &= sin(3)
\end{aligned}
$$

Source: The Author

aggressive, we could consider π and 3.14 also very close to 3, hence, leading to one more reuse case (when calculating the value of c) but paying a higher cost in quality. In chapter 3, we present how different related works have implemented approximation to achieve a behavior like the one we discussed above.

3 RELATED WORK

This chapter presents previous works that are somewhat aligned with the problem we try to solve. We divide the works in three groups: Reuse in Single-Core Environments, Reuse in Multi-Core Environments, and Approximate Reuse in order to facilitate the understanding of our contributions in each of these research subfields.

3.1 Reuse in Single-Core Environments

A variety of works has discussed reuse of computation (SASTRY; BODIK; SMITH, 2000). Implementations vary from software (where reuse is also known as *memoization* (HALL; MCNAMEE, 1997)) to hardware-based solutions and cover different granularities of code. Sodani and Sohi (1997) presented dynamic instruction reuse is with execution-driven simulation. The goal is to avoid re-execution of instructions in an out-of-order processor. Instructions' source registers are the inputs, and its result is the output. The scheme is enhanced with control of dependency links among instructions, providing reuse of a set of dependent instructions. Citron, Feitelson and Rudolph (1998) proposed the reuse of FP instructions only, focusing on multimedia applications. For each function unit that takes more than a cycle to execute (like an FP divider or multiplier), a MEMO-TABLE is used to store the results. Average speed up between 8% and 22% is achieved. Despite a hardware scheme being discussed, the results are taken from an instruction-level simulator.

Reuse of a set of instructions within a basic block is considered by Huang and Lilja (1999) and simulated using SimpleScalar (AUSTIN; LARSON; ERNST, 2002). The source operands (registers or memory) of each instruction inside a basic block are considered as part of the input. The values written to any register or memory location are considered as part of the output. Their work shows performance improvements of up to 14%. A similar system is proposed over trace level (a set of sequential basic blocks) by González, Tubella and Molina (1999). In this case, less reusability is found compared to instruction reuse, but more speedup is obtained since larger chunks of code are involved.

Kavi and Chen (2003) introduced the concept of dynamic function result reuse. In this case, only *pure* functions (global variables free, no I/O requirement, nor any change in the global state of the program) can be reused, so that the return value depends only on the function's input parameters. The authors verify the impact of (i) reuse buffer size; (ii)

reuse buffer associativity; and (iii) amount of input parameters of functions. The study presented from 10% to 67% of reusability on a variety of applications, supporting the use of the function reuse concept. Finally, Suresh et al. (2015) implemented function reuse in the interface between programs and operating system. Their mechanism intercept calls to the dynamically linked math library by preloading a memoized version of it. 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).

3.2 Reuse in Multi-Core Environments

Historically, when data value reuse was first proposed, research focused in (simulated) single-core environments for skipping snippets of execution. Very few works have extrapolated the analysis to multi-core/multi-thread scenarios.

Molina, Gonzdalez and Tubella (2002) propose the execution of a program with two virtual threads - similar to Simultaneous Multithreading (SMT) - where one speculates execution while the other uses a *Verification Engine* to quickly verify misspeculations to reduce the penalty of skipping execution by *prediction*.

Some works have put effort into avoiding reexecution of computation by thread synchronization. Long et al. (2010) propose to synchronize threads of an application in a form that execution of threads occurs by aligning identical instructions, synchronizing with a fetch priority mechanism. This is possible thanks to the code similarities among program's threads. By this, it is often possible to use a single fetch unit (since instructions are equal), and when instructions operand are also equal, execute multiple instructions together in a single thread, and applying the results to all threads. Speedup is expected, as highlighted by the authors, because the instruction window is easier to keep full, fewer instructions need to be executed, and fewer cache accesses must occur. Indeed, 1.15x speedup is achieved comparing to a two-thread traditional SMT processor, and 1.25x comparing with a four-thread SMT. The work by Mckeown, Balkind and Wentzlaff (2014) uses a similar approach. However, their solution also works with multi-threads from different programs, even though they show that this scenario is *not* the sweet spot for the technique usage.

3.3 Approximate Reuse

A few works have explored the concept of approximate function reuse under distinct names. Alvarez, Corbal and Valero (2005), present fuzzy memoization of FP instructions. Similarly to the work developed by Citron, Feitelson and Rudolph (1998), where 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 dropping 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 by Keramidas, Kokkala and Stamoulis (2015) presents the clumsy value cache, an instruction/block-level reuse technique targeting Graphics Processing Units (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 number of instructions executed, on average, by 13.5%. Sinha and Zhang (2016) 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.

3.4 Work Contribuitions

Our work is the first to consider reuse specifically targeted to FPGAs and 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 either in single or multi-core environments, 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. Additionally, we propose to cover reuse with a shared RT in a multi-core environment, which (to the best of our knowledge) was never proposed before not only for FPGA-based processors but also in GPPs.

By presenting function reuse in FPGA for 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. Moreover, the multi-core analysis can update the knowledge of reusability in modern designs, and also explain the behavior of reuse when the RT is populated simultaneously from different applications.

4 IMPLEMENTATION AND METHODOLOGY

This chapter presents the addition of a function Reuse Unit (RU) to a single-core VLIW processor, by VHDL implementation, to skip redundant function execution in single-thread programs; this is accomplished by using available BRAMs for storing the reuse information of functions. After, it details a multi-core reuse proposal to share a single centralized RU (and RT) among multiple cores, to improve reuse by leveraging inter-core likeness of code, and the simulator built to extract the results. Finally, we demonstrate the minor changes in the former hardware so we can enhance the RU with approximate reuse.

For the sake of clarity, we will explain each of this incremental steps in the sections below.

4.1 Baseline Processor

The techniques we propose are for the ρ-VEX VLIW soft-core processor (WONG; Van As; BROWN, 2008) compatible with the VEX Instruction-Set Architecture (ISA) (Hewlett-Packard Laboratories, 2009) and described in VHDL, even though there are no restrictions that would prevent its implementation to any other soft-core processor. It has a Load/Store Harvard architecture with a five-stage pipeline: *Fetch*, *Decode*, *Execute 1*, *Execute 2*, and *Write-back*. The issue-width (e.g., 2, 4, or 8), type and organization of functional units, and register file size can be configured during design time. Each pipelane may contain different Functional Units (FUs) from the following set: Arithmetic and Logic Unit (ALU) (always present), multiplier, memory, and branch units.

In this work, a ρ-VEX core is set as the default 4-issue version consisting of 4 ALUs, 2 multipliers, 1 memory unit, and 1 branch unit (as shown in Figure 4.3) and 8+8KB 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. We consider only the first case (up to eight parameters) since we have found that the functions that do not fit in this case are not good for reuse: they are unlikely to be reused (many inputs to match) and lead to high latency to check/compare memory values.

Figure 4.1: A Reuse Table entry.

Source: The Author

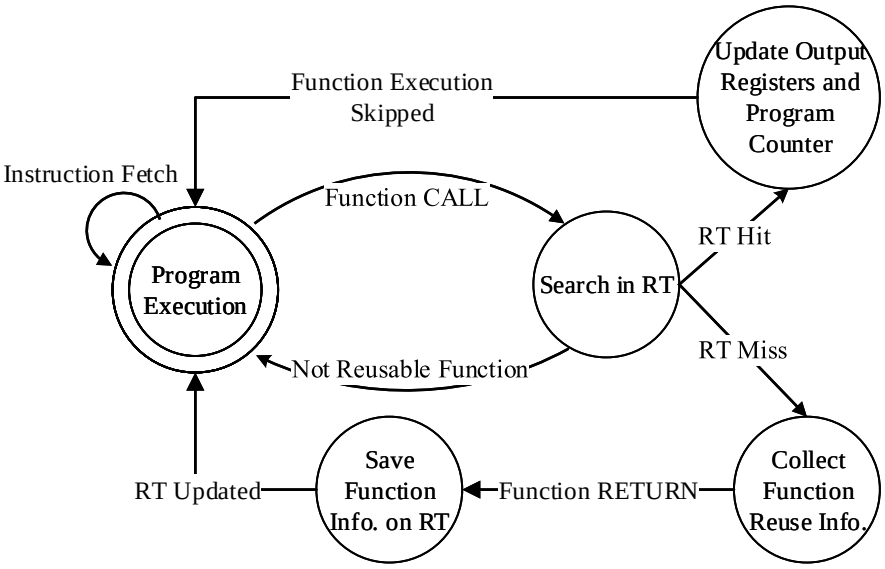
4.2 Implementing the Function Reuse Unit in a Single-Core

The proposed mechanism was implemented by modifying the ρ-VEX hardware description. An RU was attached to the processor and is composed of the following:

- *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 (Figure 4.1) contains the function's address as well as the input 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 and the number of parameters of the function. This table is filled at *design* time by the designer.
- *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.

Figure 4.2 summarizes, through a finite state machine, the functioning of the RU along with a program execution. With the *Functions Table* filled (by the system's designer, as we explain soon), on every *CALL* instruction targeting such *reusable functions*, the mechanism verifies for a previous execution of it in the RT (and thus, for the result of such previous execution), updating the processor context if it finds it. When the current function result is not available in the RT, the scheme must wait for the function to end, so the outputs are available, and then store them in the RT for further lookups. Thereby, the RT is dynamically filled at run-time. As occurs in related works (see Chapter 3) the function reuse is a technique to be applied on *pure* functions, and therefore it must be explicit to the RU whenever a function is *pure* or not (so it can be reused). Actually, our implementation considers that the RU should work with a subset of the program's *pure* functions, named *reusable functions*, marked (stored at the *Functions Table*) to be reused by the system's designer in *design* time. Therefore, it is up to the designer to find *pure* functions and decide if they should or should not be reused.

Figure 4.2: The Functioning of the Function Reuse Unit.



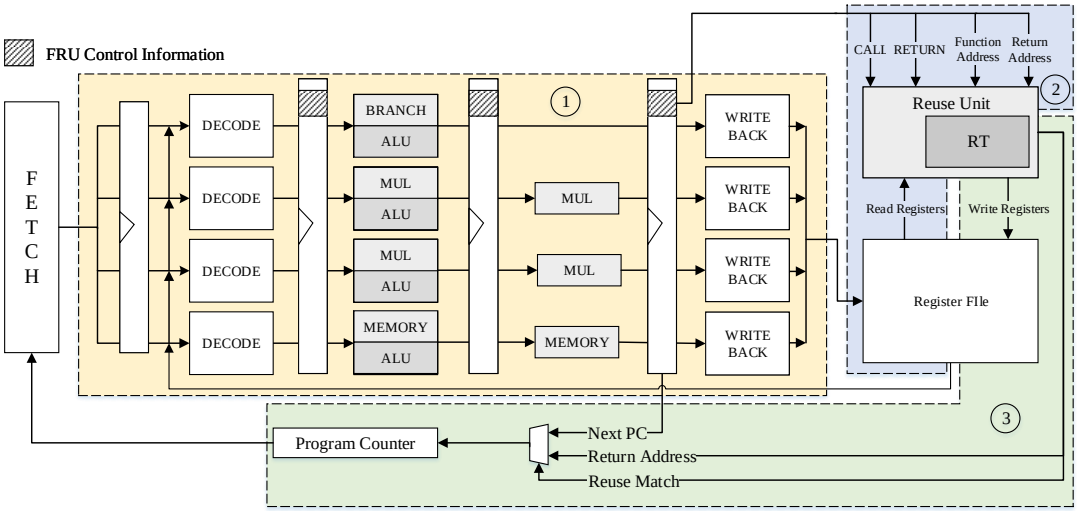
Source: The Author

Next, detailed explanation of micro-architectural aspects is presented.

4.2.1 Reuse Mechanism

Figure 4.3 details the ρ-VEX organization integrated with the RU. Three phases are highlighted, and correspond to (1, yellow) how the RU collects reuse information, (2, blue) verifies and stores reuse information, (3, green) and applies reuse (if possible). Each phase works as follows:

Figure 4.3: Organization of a 4-issue ρ-VEX with a Reuse Unit.



Source: The Author

Phase 1: When the pipeline decodes a *call*, the function and return addresses proceed through the pipeline until reaching the RU, which checks in the *Functions Table* if the function was defined as *reusable*, and also the number of parameters of that function, preparing for phase (2). If the function is *not reusable*, the processor continues their regular operation.

Phase 2: In this phase, the current function's input parameters are collected by accessing the register file. With it, the RU generates a hash key by *XORing* every 16-bit of data in the current input context and function address, similarly to the approach in (SURESH et al., 2015). The resulting key's LSBs are used as the RT index to fetch a table entry, which contains the fields shown in Figure 4.1, accordingly with the RT size. In case the fetched entry is valid, the entry's function address and input parameters are compared with those of the current call. If the comparisons match, there is an RT hit and phase (3) starts.

Otherwise, a reuse miss happens if the valid bit of the current entry is not set or if the function address and inputs do not match. In these cases, the RU waits for the function to execute regularly (until the *return* instruction signaling). Then, with the input context and outputs captured from the register file, stores a new entry (if the valid bit was not set) or replace an entry in the RT (in case of data mismatch). Therefore, the RT is dynamically filled as the program executes.

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, 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. Since reusability can be checked before the pipeline commits any instruction, no rollback mechanism is required.

In order to allow the RU to collect the inputs or the outputs of a function (available in registers *R3* to *R10*) while the processor is fetching instructions (and its operands), we added extra reading ports in the register file. Thereby, the RU and the processor can operate in parallel, and reuse misses cause no performance penalties. 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 the register file and its ports are synthesized with FPGA's registers and LUTs, adding ports increase their usage, but only marginally as our results will show. As mentioned in phase (3), 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 these write ports would be idle due to the pipeline flush and uses them to perform this operation, which results in no additional write ports. In this arrangement, the RU takes only *four* cycles to apply the reuse: one cycle to collect the inputs, two cycles to access the RT and check for reusability, and one cycle to write back the function results. For more aggressive performance enhancement, a forwarding mechanism could anticipate data to compare input context. However, four cycles are already much less than the dozens/hundreds of cycles taken by functions commonly reused in related work, and avoiding the forwarding mechanism saves logic resources. Therefore, we did not implemented a forwarding-aware RU.

4.3 Function Reuse in Multi-Core Environments

On a system with multiple cores, the straightforward approach for applying function reuse would be to attach a dedicated RU (and RT) for each core. However, as each application would have a dedicated RT running independently from the other cores, this would be like if we have multiple single-core reuse environments, which brings no novelty from the scientific analysis perspective. We, otherwise, propose to share the RT among cores, in order to possibly increase performance by benefiting from inter-core reuse, while amortizing resource usage since a single centralized RU would manage reuse instead of multiple, one-per-core, RUs. An overview of the idea is shown in Figure 4.4.

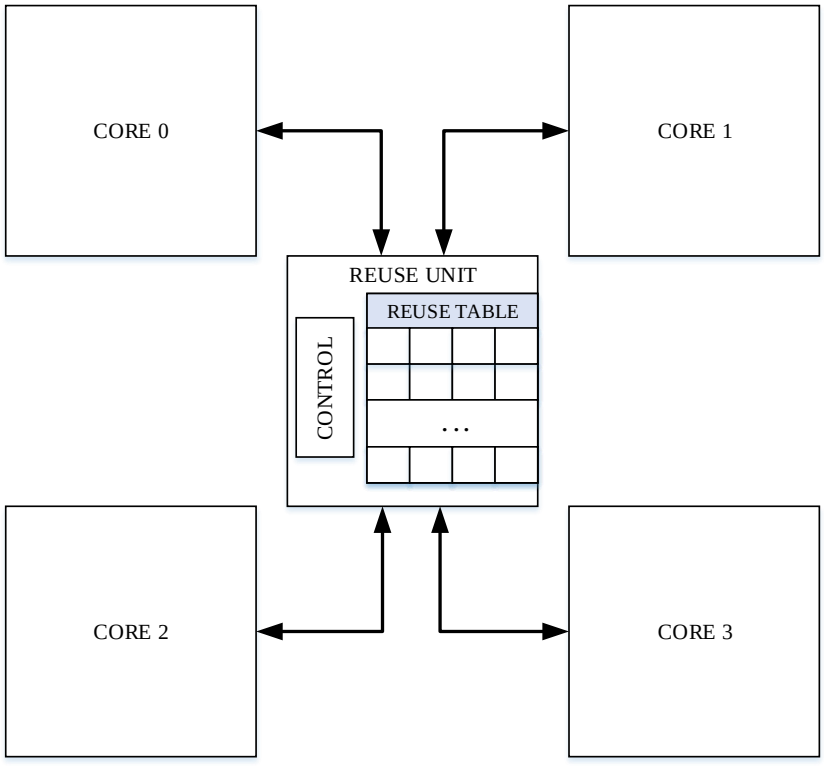
Our multi-core function reuse mechanism considers the following:

- Each core in the system is able to pass reuse-related information (as the functions and returns addresses) trough their pipelines (as occurs in the single-core case, see Figure 4.3), sending them to the centralized RU.
- The functioning of the RU is same as in single-core: it receives a function with its input context information, and fetch from (or update) the RT. Differences occur only in the interface with the system since now the RU needs to communicate with every system core. Therefore, all Inputs/Outputs (IOs) of the RU in Figure 4.3 should be multiplexed: information from pipeline as *call*, *return*, function address, return address; interface with register files; interface with fetch units. Thereby, no additional IOs ports in the RU are needed. The *decoding* of a CALL instruction in the pipelines can be used to control the access in the RU, multiplexing its IO to the

correspondent core in advance of the arrival of reuse information.

- Inter-core reuse is only possible if the functions to be reused have the same addresses in all programs instances. This is easily achieved by appropriate ordering the functions while writing programs, or when including libraries to compile a program. A specialized mechanism (such as a translate table) could be used to guarantee that same functions in different programs (with different addresses) are seen as the same by the RU, but we do not cover these mechanisms.
- Finally, when a core requests a reuse verification to the shared RU, the unit is locked for the 4 cycles it operates (for *reusable functions*), dropping any other requests during this time. If multiple requests occur in the same cycle, access grant is given to the smallest processor id that holds a *reusable function*. Cores that had dropped requests continue its regular execution. Note that each conflicting request could lead to a miss or hit in the RT, which means that dropping a request that would lead to a miss does not change the execution of the application. As future work, a queue will be implemented to handle all reuse requests.

Figure 4.4: A Multi-Core Environment with a Shared Reuse Unit.



Source: The Author

4.3.1 Multi-Core Function Reuse Simulator

To analyze the idea, we implement a simulator to handle function reuse in a multi-core environment, based on function traces. For such, we added a log generator to the RU. Thereby, we can execute programs in ρ-VEX to generate a trace (we will call it *reuse trace*) of the *reusable functions* called. A brief example of a *reuse trace* is depicted in Figure 4.5, where each line contains the timestamp (execution cycle) of the call, the target function, the inputs, and the time (cycles) the function took to execute. As one can see, the trace is sorted in ascending order according to the timestamp. Although the *reuse traces* are generated by separated single-core executions, providing them to our simulator is all the necessary to analyze performance in the multi-core environment. Thus, we can use software analysis (instead of slower hardware simulation), to compute precisely the performance gains of sharing the RT. This is important because, as we shall present in the Results Chapter, the number of scenarios to be combined in multi-core testing increases quickly.

Here we introduce FunctiOn Reuse MulticOre SimulAtor (FORMOSA): the simulator we implemented to manage multiple *reuse traces* to act as a centralized RU, handling function reuse requests from multiple programs, and updating the centralized RT accordingly. For this, the simulator considers *reuse trace* as a faithful abstraction of a program execution. Although we have generated traces from our modified ρ-VEX, FORMOSA could be used to analyze function reuse in any generic environment, as long as the log files follow the pattern showed in Figure 4.5.

The high-level operation of the FORMOSA is presented in Algorithm 1, and is explained briefly next. To use the simulator, the log files ($logs$) and the table size (tbl_size) must be given. FORMOSA executes until it consumes all lines from all $logs$. At each iteration, it consumes one line from each log and *chooses* the earliest request line (l. 2) by looking at the $tmstmp$ field, saving it in the $chosen\_line$ variable (l. 3). After, it checks whenever a reuse request will find the RU locked. It is done by simply check-

Figure 4.5: Example of a *Reuse Trace* from the Reuse Unit.

```
<tmstmp> <Func_ID> <Input1> <Input2> <Input3> <Input4> <Cycles>
  420    00011D30  40000000 00000000 40240000 00000000    345
  554    000104E0  3FC99999 9999999A 3FC99999 9999999A    123
  636    0000EF80  3FC99999 9999999A 3FC99999 9999999A    78
  773    000103C0  40000000 00000000 3FA47AE1 47AE147C    129
```

Source: The Author

Algorithm 1 The FORMOSA's Algorithm.

Require: log files ($logs$), table size (tbl_size).

$RU\_CYCLES \leftarrow 4$

1: **while** there are lines to be analyzed in logs **do**
2: $chosen\_log \leftarrow$ findSmallestTimestamp($logs$)
3: $chosen\_line \leftarrow$ readLineFromLog($chosen\_log$)
4: **foreach** log **in** $logs$ **where** $log \neq chosen\_log$ **do**
5: $log\_line \leftarrow$ readLineFromLog(log)
6: **if** $log\_line.tmstmp < chosen\_line.tmstmp + RU\_CYCLES$ **then**
7: $FoundRTLocked[log\_idx] \mathrel{+}= 1$
8: **else**
9: rollbackLogFile(log)
10: **end if**
11: **end for**
12: $tbl\_pos \leftarrow$ hash ($tbl\_size, chosen\_line.Func\_ID, chosen\_line.Inputs$)
13: $RT\_entry \leftarrow$ RT(tbl_pos)
14: **if** $chosen\_line.Func\_ID = RT\_entry.Func\_ID$ **and** $chosen\_line.Inputs = RT\_entry.Inputs$ **then**
15: $SavedCycles[chosen\_log\_idx] \mathrel{+}= chosen\_line.Cycles - RU\_CYCLES$
16: updateLogTimestamps($chosen\_log$,$SavedCycles$)
17: **else**
18: updateRT($tbl\_pos, chosen\_line.Func\_ID, chosen\_line.Inputs$)
19: **end if**
20: **end while**

ing if the remaining lines of the current iteration contain a timestamp that is in the range from $chosen\_line.tmstmp$ up to 4 more cycles (l. 6), which are the cycles that the RU is locked. For all log lines that found the RT locked, the reuse verification (which could or could not lead to reuse) is missed and counted so that one can analyze accessing conflicts later (l. 7). Logs where timestamps were ahead of $chosen\_line.tmstmp + 4$ have their handlers rolled-back (l. 9), since they can be the next *earliest timestamp* and must be reevaluated in the following iteration.

Finally, the RU can use the $chosen\_line$ to attempt reuse in the same way it occurs in the single-thread scenario. It generates the hash key to get the position (tbl_pos) to lookup in the RT (l. 12). Then, the RT is consulted, returning a RT_entry (l. 13). Function addresses ($FUNC_ID$s in the algorithm) and inputs are compared to check for a reuse match. When reuse is successful (l. 15), FORMOSA accumulates the number of cycles that were skipped (given by the number of cycles spent when the function was computed, less the four cycles to apply reuse). This is used later to observe performance gains over baselines. Also, the saved cycles are diminished from upcoming timestamps of the current $chosen\_log$ (l. 16), since in a reuse hit would cause upcoming calls to occur

earlier. If reuse fails, the RT is updated with $chosen\_line$ data in the tbl_pos (l. 18).

Note that FORMOSA does not require the output values in the *reuse traces*, since it does not execute any program (and therefore does not update any processor context). Thus, we save simulation memory by ignoring output values. The FORMOSA final results are the total saved cycles of each application (in its own core). Therefore, one has to know the number of cycles of the baseline, which is trivial, to infer metrics such as speedup. Finally, if the *reusable functions* are guaranteed to be *pure*, function reuse is also guaranteed to be safe and correct.

To validate the simulator, we have tested it with small sized traces, checking results manually. Especially, conflicts in border values of time (for locking the RU) were verified. Also, the simulator has the same results as the VHDL implementation (from the previous chapter) for the cases where a *single* trace is provided.

4.4 Enhancing Function Reuse Possibilities with Approximate Computing

Although previous work (and ours, as we discuss later) shows that reuse can achieve speedup, some classes of applications hardly benefit from function reuse. This occurs because the input space can be very extensive, reducing chances to match previous calculations. At the same time, many of them can tolerate some error in their outputs, being convenient for approximate computing. Classical examples are image filters, video encoders, and neural networks (HAN; ORSHANSKY, 2013; ESMAEILZADEH et al., 2012).

In this section, we present how our function reuse mechanism could embrace the approximate concept, with *approximate reuse*, leveraging reusability in the aforementioned scenarios. Also, we show how it could be used alongside with the *precise reuse*, with marginal changes in the former hardware. This is achieved by dropping some LSBs from the inputs, so similar/close values are hashed to the same positions in the hash function. Thus, we enhance reuse possibilities, increasing the number of scenarios where it can be useful. Although we discuss the minor changes to implement it in hardware, we did not implement it on VHDL, but instead, investigated approximate function reuse in a single-core scenario by simulating and estimating the behavior of applications, as discussed in the Results chapter.

Next, we highlight the necessary additions to the reuse mechanism. Since modifications for *approximate reuse* do not change the overall multi-core reuse arrangement,

we simplify the explanation considering a single-core environment.

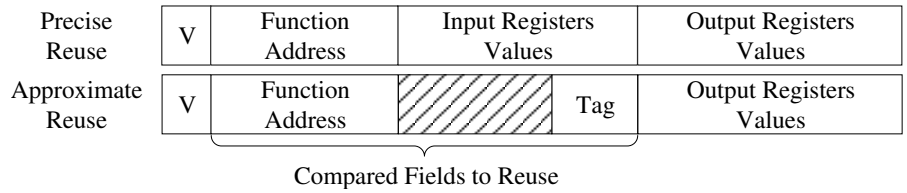
4.4.1 Modifications to Support Approximate Reuse

Below we detail the modifications required so a hardware RU supports *approximate reuse*:

- The *Functions Table* (which formerly stored the *reusable function's* addresses and quantity of inputs) must indicate, for each function, if it is to be *approximately* or *precisely* reused. If *approximately*, it must indicate how many LSBs should be dropped from the inputs for generating the hash key and tuning the performance/output quality trade-off. This is, as in the former case, up to the designer to set up at *design* time.
- When reusing *approximately*, the RU has to drop the inputs' LSBs before generating the hash key, to group close values. This is easily done in hardware by *ignoring* wires. The LSBs of the hash key itself still index the RT, while the full hash key is used as a *tag* in approximate mode.
- The RT entries can now contain data in two different structures (see Figure 4.6), accordingly with the reuse mode (precise or approximate). In precise mode, all inputs are stored. In approximate mode, only the *tag* is stored. The tag is stored in a sub-part of the former entries, and thus, no change in the RT entries width have to be made.
- When attempting precise reuse, input values are compared. Otherwise, when attempting approximate reuse, the *tag* is used. Therefore, comparing a tag uses a subset of the hardware that compares precise inputs.
- A few multiplexers must be included to control, based on the current reuse mode, when to compare the inputs or the tag; when to ignore LSBs of the inputs to calculate the hash key; and when to store the inputs or the tag in an RT entry.

As we described, it is possible to enhance function reuse with approximate reuse with marginal addition in control logic and *Functions Table*. Thereby, it is possible to have a generic RU that can switch between precise and approximate reuse easily.

Figure 4.6: A Reuse Table entry that supports precise and approximate modes.



Source: The Author

4.4.2 Modifying Software to Analyze Approximate Function Reuse

To measure the potential of approximate function reuse to improve performance, we have changed the original program/application we want to analyze for one which counts functions calls, how many times it repeats, and the number of table hits. This is done considering the same replacement policy and hash function that we propose to exist in hardware (including the dropping of LSBs). In time, the benchmark modification is based on the work of Brandalero et al. (2017). This modified version is executed on a regular Linux machine. This was needed because approximate benchmarks (as the one we use in Chapter 5) commonly consider image or video processing, requiring system calls to access files, and specific libraries not supported by the ρ-VEX platform. At the same time, we have implemented a pure C code of the respective kernel under analysis, hard-coding the inputs (like an image) in the source so we have a version that can run in ρ-VEX.

With the pure C version of the program, we have an estimation of the number of cycles that the benchmark and the function we want to approximately reuse demand on ρ-VEX. Likewise, with the modified benchmark application running on a regular Linux machine, we collect the number of reuse hits according to with the number of LSBs we chose to drop. With the number of cycles of the benchmark, the function we want to reuse, and the reuse hits, we can analyze performance improvement.

5 RESULTS

In this chapter, we explain the evaluation methods and present the results of each aforementioned function reuse technique. For performance results, which we introduce first, we follow the same order of previous chapters: first, we debate (*precise*) function reuse in single-core environments, then in multi-core environments and finally the potential of using approximate function reuse. After, we summarize area/resources usage analysis over the covered techniques.

5.1 Function Reuse in Single-Core Environments

5.1.1 Experimental Methodology

We performed cycle-accurate simulations of the processor execution using Mentor Graphics Modelsim 10. We evaluated the speedup of six benchmarks, five from the WCET benchmark suite (GUSTAFSSON et al., 2010): *lms, ludcmp, minver, qurt, st*; and one from the Powerstone suite (SCOTT et al., 1998): *fir*. To measure performance, we compared the execution cycles of the benchmarks on ρ-VEX with and without the RU, experimenting with RT sizes varying from one to 32K lines. We could use cycles as reliable performance metric since we do not change the critical path, and thus maintained the clock frequency. The final state (registers and memory values) of the enhanced processor was compared to the final state of the baseline, to ensure programs still executed correctly after our modifications. The benchmarks were compiled with LLVM (LATTNER; ADVE, 2004) with the back end modified to support the VEX ISA (JOST; NAZAR; CARRO, 2016), and *-O3* flag.

5.1.2 Case Study - Floating-Point

As a case study, we optimized the standard soft-float library (HAUSER, 2002) - which emulates FP operations using integer hardware - applying reuse over some of its functions. The soft-float is a library traditionally used to support FP operations in systems where no FP ALUs are available, or when FP is not supported by the ISA (the case for VEX ISA of ρ-VEX). It conforms to the IEEE Standard for Floating-Point Arithmetic,

and is available, for example, in the LLVM (used here) and in the gcc compiler. We considered the four basic operations (add, sub, mult, div) in double FP precision of the soft-float library as the *reusable functions*. The consideration is based on the profiling of applications, which showed that those functions are commonly called. As the soft-float library was statically linked at compile time, we could consult the assembly of the code to get the addresses of the functions, and the amount of input and output registers used, assigning it to the *Functions Table* on our VHDL description.

5.1.3 Reusability

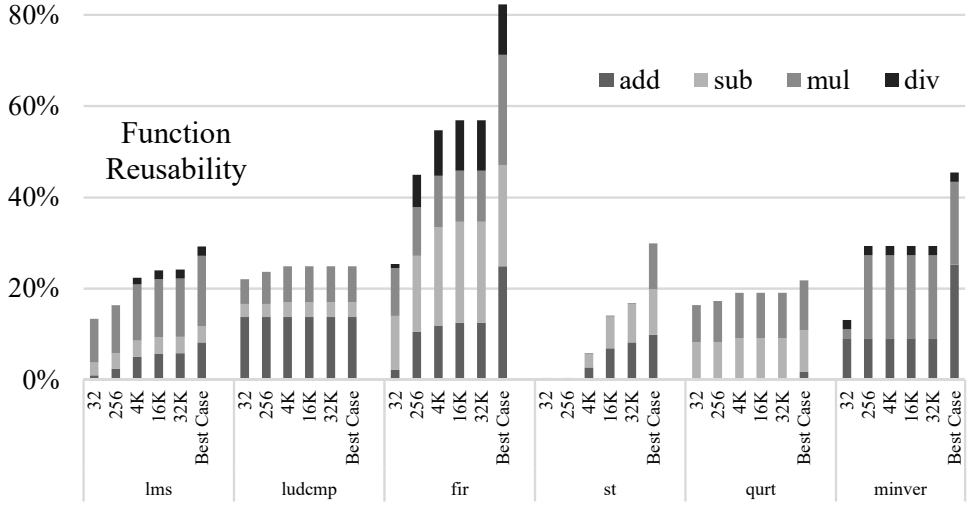
Figure 5.1 presents the reusability of the case study functions, which will influence the performance gains. Grouped by benchmarks, each column depicts the stacked RT hit rate for *add*, *sub*, *mul*, and *div* functions, according to the number of RT lines (*x-axis*). Naturally, reusability increases with the RT size, since more reuse information is available, so a match attempt succeeds. A *Best Case* scenario was also included, presenting results for a hypothetical RT that behaves like an infinite and perfect collection that stores all entries without replacements. This best case indicates how good are the table sizes and the chosen hash function. 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 generally take longer than *add* or *sub*. Although reusability does not guarantee performance gains (for instance, if a function is called only a few times or if it is extremely fast), it gives the intuition of the space exploration for when we further look upon performance speedups.

For these benchmarks, reusability can vary from almost none (*st* with 32 lines) to more than 80% (*fir* in the *best case*), while in overall, the functions reusability can sum up to more than 20% in all benchmarks, when using a sufficiently large table. In addition, *mul* was the most reused function, while *div* was the least one.

5.1.4 Performance

Fig. 5.2 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).

Figure 5.1: Reusability of case study functions

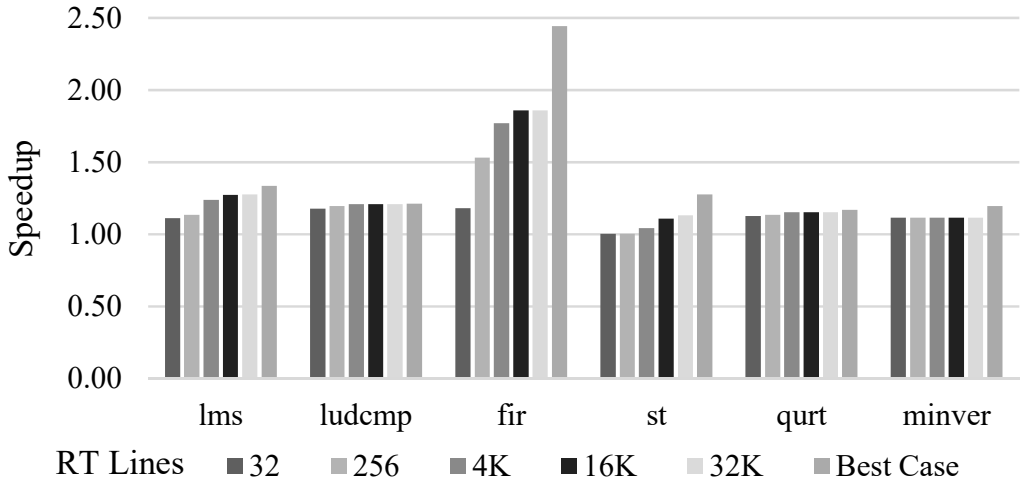


Source: The Author

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 places the improvements in 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 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).

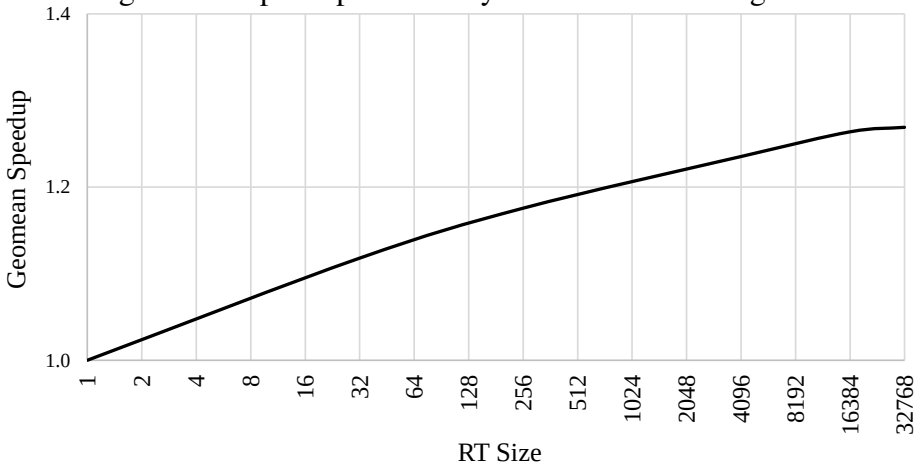
Finally, we can observe the speedup scalability as the RT grows. Figure 5.3

Figure 5.2: Speedup for different Reuse Table sizes compared to the baseline.



Source: The Author

Figure 5.3: Speedup Scalability as the Reuse Table grows.



Source: The Author

presents a comparison among the RT depth (in $\log_2$ horizontal axis) and the correspondent geometric mean speedup achieved (in the vertical axis). It elucidates that, in some cases, we can achieve considerable gains in performance with low resource overhead, as will be shown later. For example, a 32-line RT can provide a speedup of 1.12x over the baseline, while it can reach 1.18x of improvements with 256 lines. A 4K-line RT, which fits in all five tested FPGAs (which will be discussed later) can reach 1.23x speedup. Therefore, even resource-limited FPGAs can benefit from applying the proposed mechanism. When it comes to high-end FPGAs, 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 (e.g., *fir*).

5.2 Function Reuse in Multi-Core Environments

5.2.1 Experimental Methodology

The ρ-VEX processor, in which our work is strongly based on, has several limitations for running real-life benchmarks: e.g., it runs with no operating systems and therefore cannot call any traditional system function like for file and memory handling. More importantly, it cannot run parallel programs, which would be appreciable to this work. Although we could generate *reuse traces* by modifying other simulators that support parallel applications (as long as it respects the *reuse trace* format presented in Figure 4.5), we would be changing the simulation platform, and thereby affecting the results analysis, which is specially inadequate to compare the multi-core results with single-core

results, presented previously. Moreover, we want to keep the ρ-VEX as the platform since it provides reliable results in area.

We have constrained our multi-core test setup to multiple programs running the same kernel, but with different inputs - as it is not possible to run pure parallel applications in ρ-VEX- maintaining our benchmarks set (*fir, lms, ludcmp, minver, qurt, st*) and the *reusable functions* (*add, sub, mul, div*, in double FP). The baseline continues to be the system without function reuse. For each benchmark B, *four* versions, with different inputs i were created: B_i1, B_i2, B_i3, B_i4. We generated the *reuse trace* for each of these versions, which were combined among each other to feed FORMOSA with all possible versions combinations of benchmarks (see Table 5.1), and get speedup results.

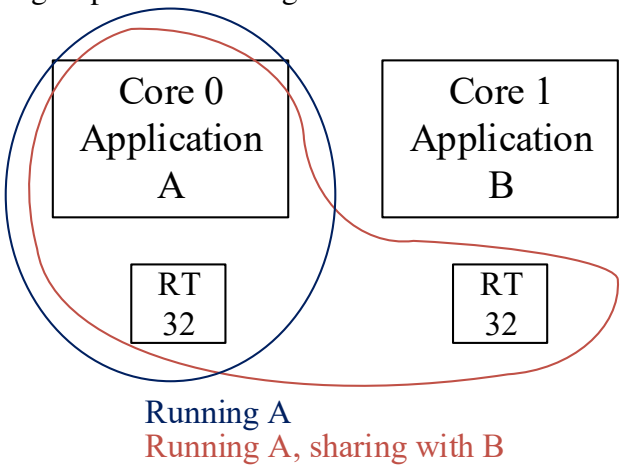
To examine the benefits of using a shared RT we always compare scenarios where the total BRAM per configuration is maintained, in other words, we preserve the RT size

Table 5.1: Investigated Simulation Scenarios for a Benchmark B

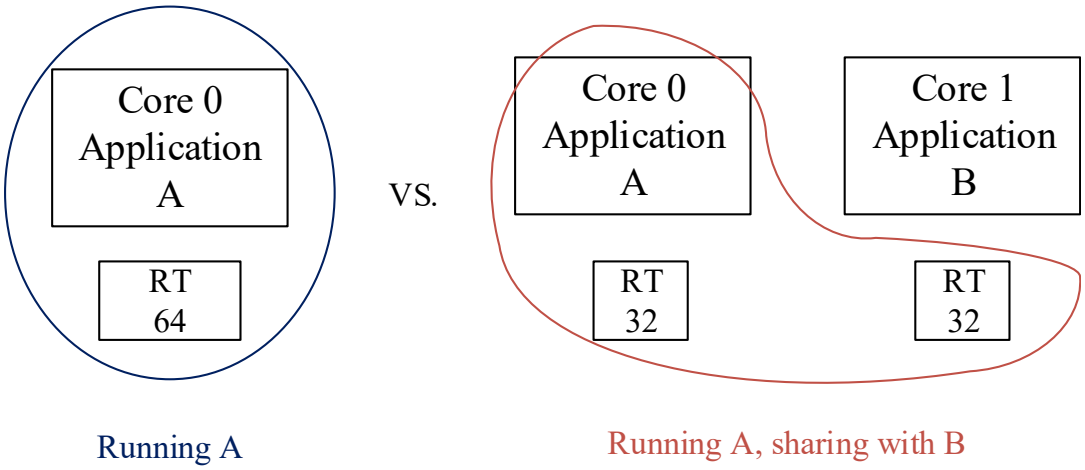
Scenario	**Description**	**RT Size as Seen by the Applications**
B_i1	B_i1 running with a dedicated RT	B_i1 sees an RT with size S.
B_i2	B_i2 running with a dedicated RT	B_i2 sees an RT with size S.
B_i3	B_i3 running with a dedicated RT	B_i3 sees an RT with size S.
B_i4	B_i4 running with a dedicated RT	B_i4 sees an RT with size S.
B_i1, B_i2	B_i1 and B_i2 sharing the RT	B_i1 and B_i2 see a shared RT with size $2 * S$.
B_i1, B_i3	B_i1 and B_i3 sharing the RT	B_i1 and B_i3 see a shared RT with size $2 * S$.
B_i1, B_i4	B_i1 and B_i4 sharing the RT	B_i1 and B_i4 see a shared RT with size $2 * S$.
B_i2, B_i3	B_i2 and B_i3 sharing the RT	B_i2 and B_i3 see a shared RT with size $2 * S$.
B_i2, B_i4	B_i2 and B_i4 sharing the RT	B_i2 and B_i4 see a shared RT with size $2 * S$.
B_i3, B_i4	B_i3 and B_i4 sharing the RT	B_i3 and B_i4 see a shared RT with size $2 * S$.
B_i1, B_i2, B_i3, B_i4	B_i1, B_i2, B_i3 and B_i4 sharing the RT	B_i1, B_i2, B_i3 and B_i4 see a shared RT with size $4 * S$.

per core, as we show in Figure 5.4. When we share the RT, an application will see a larger table, which does not mean that we have used more BRAM. Instead, the application can simply access a table from another core which formerly was not possible. If we share the RT in a dual-core, applications will see an RT two times larger. If we share the RT in a quad-core, applications will see an RT four times larger. As we can have single, dual, or quad-core sharing scenarios, results are always presented in relation to the table size per core, which is maintained regardless the number of total cores.

Figure 5.4: Measuring Impacts of Sharing the Reuse Table from a Multi-core Scenario.



(a) Maintaining the size per core, 32 lines in this case, explicits the benefits of sharing the RT. This is the approach we use in our performance analysis.



(b) Maintaining the total size, 64 lines in this case, is not a fair comparison since it compares two different configurations: sing-core and dual-core. We do not want to pay the cost of more cores, but instead, demonstrate how we could benefit from sharing the RT in a former multi-core scenario.

Source: The Author

5.2.2 Inputs Definition

We have redefined the inputs for all six benchmarks, creating four versions for each of them. For kernels where inputs are large arrays (*fir, lms, ludcmp, st*), we generated random values, using them as the inputs. Therefore, results are pessimistic, since in real-life data is more likely to be reusable (similar values, smaller input space). For benchmarks with few inputs (*minver, qurt*), they were defined based on examples from literature. For the *minver*, which calculates an inverse matrix, we have used matrices examples from (LAY, 1999). For the *qurt*, which calculates quadratic equations, we have used examples from a precalculus book (STITZ; ZEAGER, 2013).

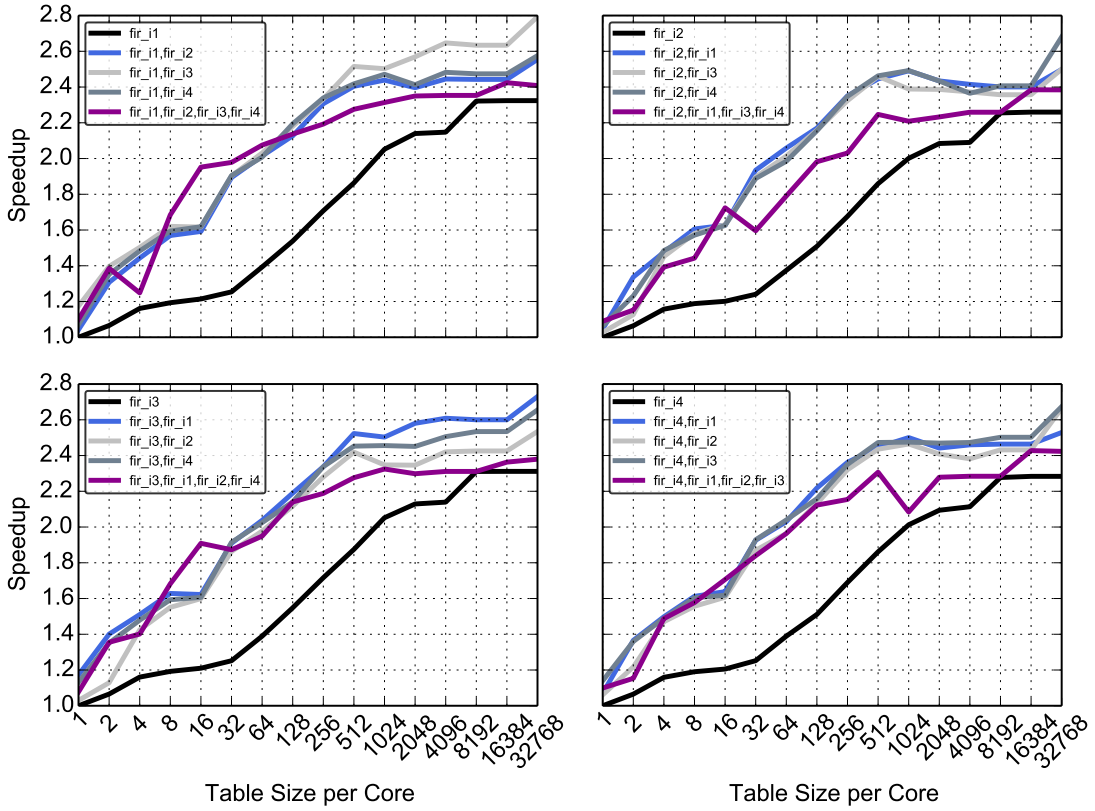
It is not a trivial task to define significant benchmarks inputs. In fact, much research is strictly targeted to build meaningful benchmarks and input sets. Our attempt here was to avoid running multiple applications with the same inputs, to do not introduce distortions to our results. Nevertheless, as we said before, we expect that our inputs definition is pessimistic. Naturally, as inputs were redefined, even single-core results upcoming, will differ from the previous section Function Reuse in Single-Core Environments.

5.2.3 Performance

We evaluate performance of a shared RT for each benchmark, pointing out strengths and weaknesses of the technique. Figure 5.5 presents results for the *fir* benchmark. Four plots present the speedup from the point of view of each benchmark version (i.e., the benchmark running with a given input). For example, in the upper-left plot, we show the behavior of the *fir* application with the input 1 (named *fir_i1*) and how its speedup depends on the applications it shares the RT with. Still in the upper-left plot, for every table size, we plot the speedup of *fir_i1* running alone with its dedicated RT (the black line); the speedup of *fir_i1* sharing the table with one other benchmark (e.g., *fir_i1*,*fir_i2* in royal blue is the speedup of *fir_i1* while sharing the RT with *fir_i2*); and the speedup of *fir_i1* when it runs sharing the RT with all other versions (*fir_i1*,*fir_i2*,*fir_i3*,*fir_i4* in dark magenta). Similarly, the upper-right plot shows results from the point of view of *fir_i2*, the lower-left of *fir_i3*, and lower-right of *fir_i4*.

As one can see, for the *fir* benchmark, using a shared RT is very beneficial. For every version of it, the shared RT scenarios perform better than having a dedicated RT, which is achieved maintaining the total BRAM within each configuration (recall Figure

Figure 5.5: Speedup for the fir benchmark.

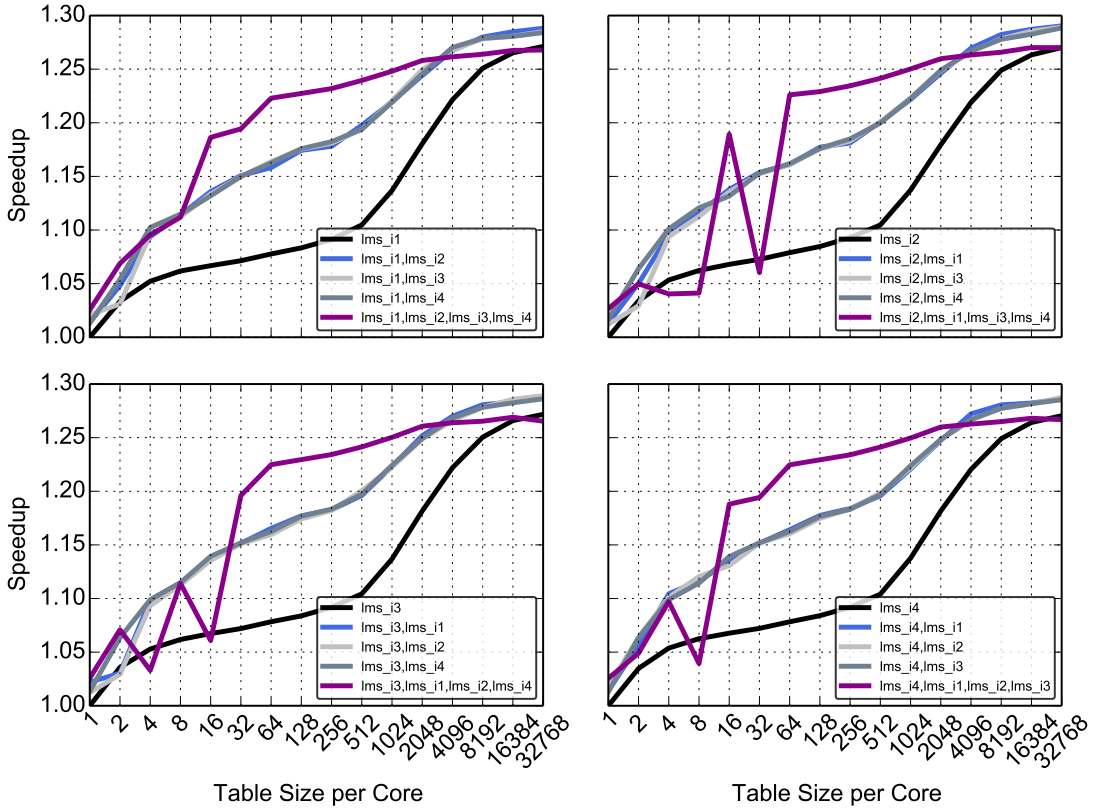


Source: The Author

5.4, item a). For small tables sizes, this can be explained since the applications see a larger table, which can lead to better data disposition. Furthermore, *fir* provides good inter-core reuse since it is based upon a filter, which is common in all versions, and thus, many redundant executions can be reused. For *fir_i3* (lower-left plot) with 32 lines per core, for example, the scenarios where the RT is shared reach about 1.9x of speedup, while a dedicated RT would reach a lower value of 1.25x. Finally, note how sharing the RT between two applications instead of four is better for larger tables. This phenomenon occurs as reuse rates of dual and quad-core simulations *plateau*. This means that even though we increase the RT, the hash mapping does not benefit from it to spread data in the RT. Otherwise, the same positions are being accessed and more applications accessing the table leads to more unwanted replacements, which mitigate performance gains.

Figure 5.6 depicts speedup for the *lms* benchmark. Again, sharing the RT is beneficial. This time, we highlight how using a *common* RT can be used to achieve equivalent performance of a *dedicated* RT while reducing total BRAM used. For *lms_i2* (upper-right plot), for example, sharing the RT with all other *lms* versions speeds up the application by 1.22x with 64 lines per core (256 lines total, in the quad-core configuration). With a

Figure 5.6: Speedup for the lms benchmark.



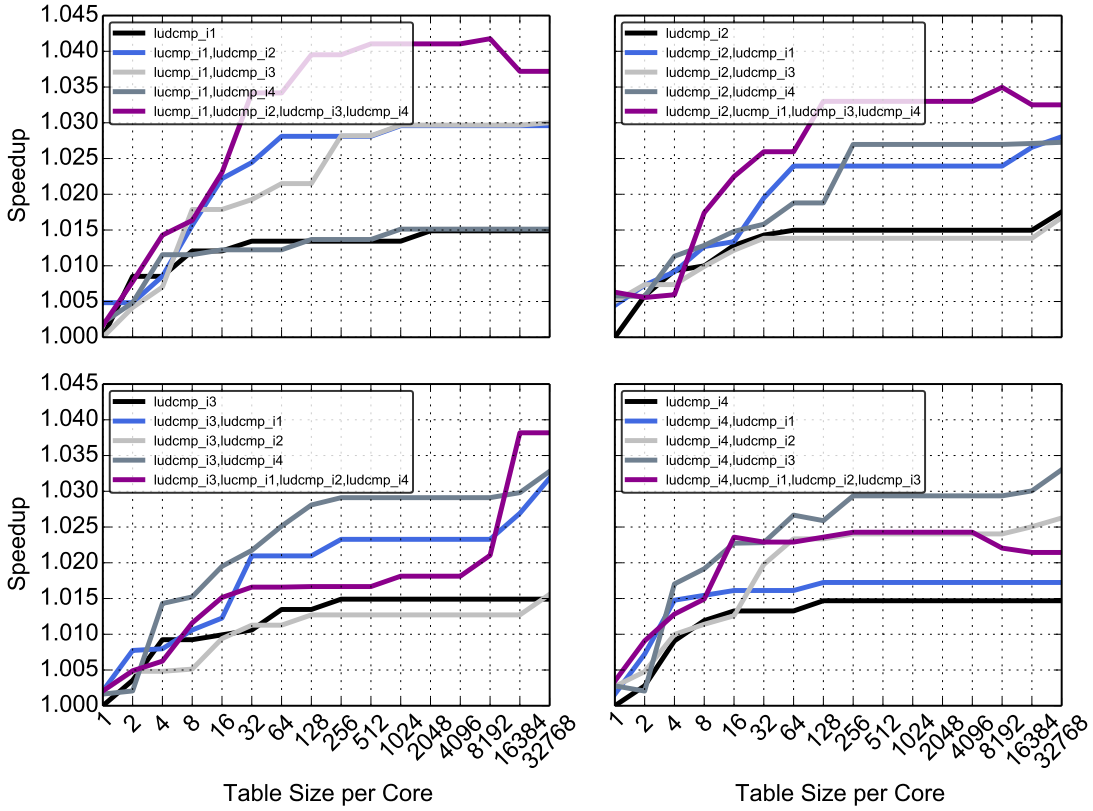
Source: The Author

dedicated RT (black line in the plot), *lms_i2* would exceed this mark only with 8K lines.

For small tables sizes (from 2 to 32 lines per core), speedup in the quad-core setup looks unstable in three out of four versions perspectives (*lms_i2*, *lms_i3,* and *lms_i4*). Such instability can occur due to the hash function, whose mapping depends on the RT size, which can be affected in the aforementioned range. This is amortized when larger tables are considered since better occupation (and thus, scatter-mapping) of the RT is possible. Nevertheless, like in *fir* (see Figure 5.5), multiple applications competing for the table reduces inter-core reuse benefits as reusability (and speedup) *plateau* for larger tables.

For the *ludcmp* benchmark, shown in Figure 5.7, speedup is very limited, reaching about 4% in the best case (speedup of *ludcmp_i1*, when sharing the RT in the four core environment). This is a consequence of the inputs, which have low locality. For this benchmark, inter-core reuse occurs in around 1% of the RU reuse attempts, which improves poorly the already deficient scenarios. Despite the charts looking very different from each other, the overall speedup is so small that lines are almost noise, a consequence of small differences in the absolute reuse hits. Actually, all charts look similar, with

Figure 5.7: Speedup for the ludcmp benchmark.



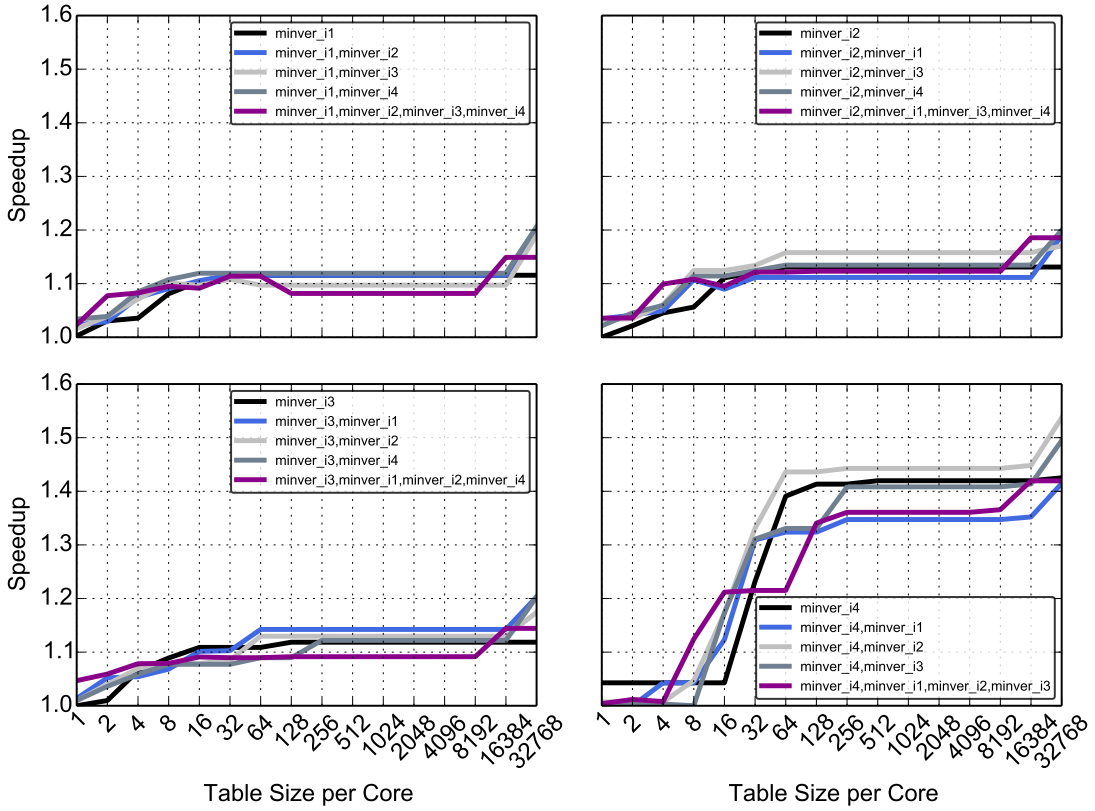
Source: The Author

speedups near to 0%.

In Figure 5.8, speedup for the *minver* benchmark is presented. For the versions *fir_i1* (upper-left), *fir_i2 (upper-right)*, and *fir_i3* (lower-left), running with a shared RT in the dual-core setup present a small improvement over the dedicated single-core cases, while quad-core setups present a small worsening in the performance.

Again, improvement comes from inter-core reuse rates of about 7% over all tables sizes, while inter-core conflicts (i.e., when one application tries to use the RU, but it is already occupied by another application running in a core with lower *id*) reaches only about of 1%, again for all tables sizes. This constant behavior occurs since *minver* is the shortest benchmark, with a small number of functions called, and where increasing the table size has minimal impact on the results. The *minver_i4* version is a bit different: reuse rates (and speedup) are higher because of the inputs, which have better reuse locality. In this case, as in previous analysis, when reuse rates are trending up (table size per core from 2 to 32 lines), sharing the RT is effective, improving performance over a dedicated one (the black line in the plots). When reuse inter-core saturates (table size per core above 64 lines) a dedicated RT regain importance.

Figure 5.8: Speedup for the minver benchmark.

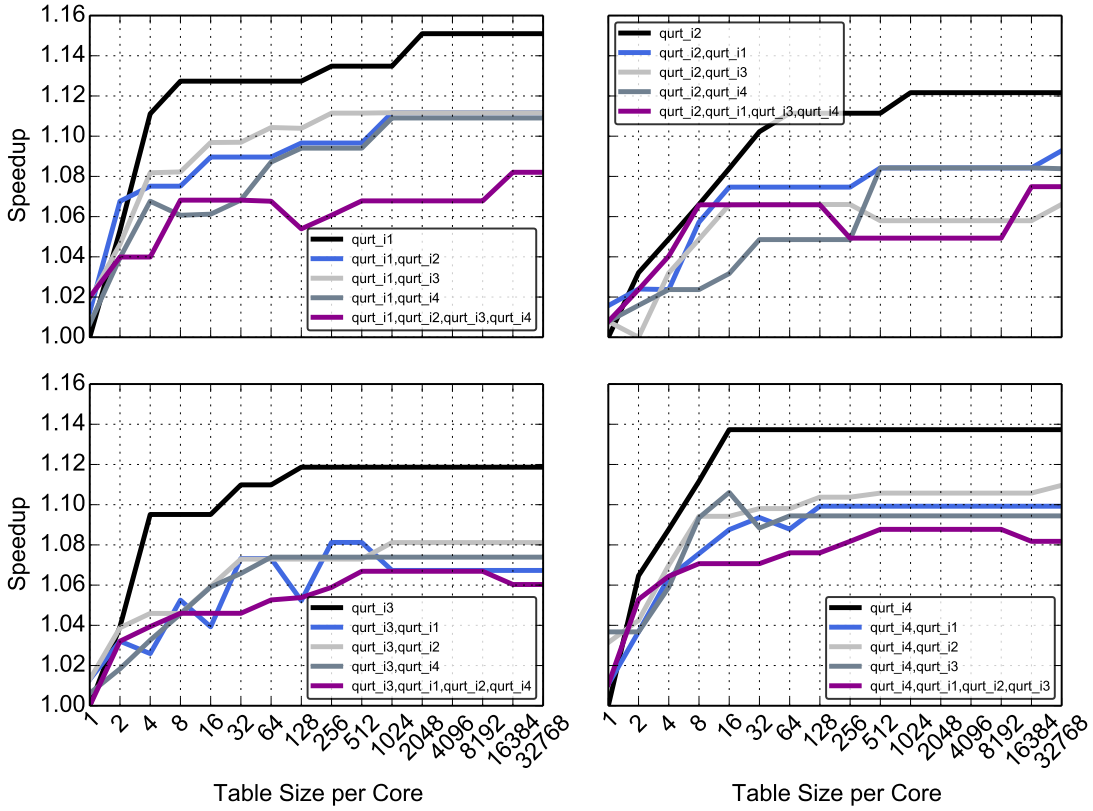


Source: The Author

For the *qurt* benchmark, presented in Figure 5.9, inter-core reuse is nonexistent. This is reflected in the charts, where dual or quad-core RT-sharing performs worse than the dedicated RT for almost all table sizes and benchmark versions. Therefore, even though the total RT size seen by applications is larger, programs are updating the RT in a non-collaborative way, which is a weakness of our technique. By this behavior, for 2K lines per core, speedup can drop from 1.15x (in the *qurt_i1* version, in a dedicated RT scenario) to 1.07x (roughly half) when sharing the table among all other versions (dark magenta).

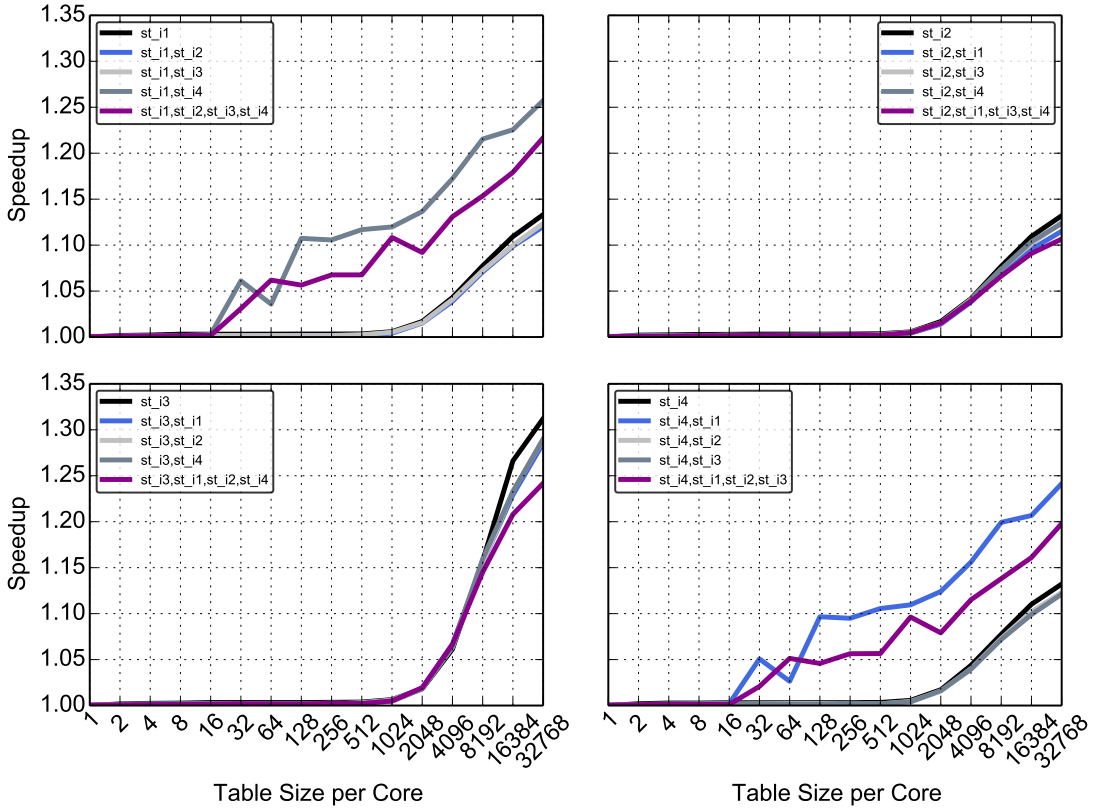
Finally, for the *st* benchmark, we can observe a mutual and strong collaboration between *st_i1* and *st_i4* versions, as depicted in Figure 5.10. Note how the technique leverages inter-core reuse in both versions, mutually. By the point of view of *st_i1* in the upper-left chart, the dark-grey line (*st_i1,st_i4*) prosper way earlier than the dedicated-RT case. The same occurs from the point of view of *st_i4*, in the blue line (*st_i4, st_i1*) of its respective chart (lower-right). For *st_i2* and *st_i3*, inter-core reuse is negligible and provides little changes in the performance.

Figure 5.9: Speedup for the qurt benchmark.



Source: The Author

Figure 5.10: Speedup for the st benchmark.



Source: The Author

In short, we showed that sharing the RU provides speedup for three out of six multiple-core benchmarks (*fir*, *lms*, and *st*). For small tables, speedup comes from apparent larger RT. When more RT lines are available, speedup comes from both inter-core reuse, and better data disposition. Moreover, we showed that as reuse possibilities reach its limit for large tables, sharing is no longer effective and achieves similar results that using a dedicated RT per core.

For one case (*ludcmp*), speedup occurs but is negligible, and for the two remaining (*minver* and *qurt*) performance overhead occurs. The latter is a consequence of programs executing in a non-collaborative way.

5.3 Using Approximate Computing to Function Reuse

5.3.1 Experimental Methodology

The *sobel* image-processing filter (case-study for approximate function reuse) from the AxBench suite (YAZDANBAKHSH et al., 2016) was used to evaluate approximate function-reuse using 30 distinct images, in an approximation scenario where 4 LSBs are dropped from the input. 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 function which operates the convolutional kernel was the one considered as *reusable*.

As discussed in the section 4.4.2 Modifying Software to Analyze Approximate Function Reuse, we create two versions for the benchmark we want to test under approximate reuse. With them, we can collect how many cycles the benchmark and each call for the convolutional *sobel* function spent, and the number of reuse hits. With this, we know how many functions we can skip and how much time they take to execute. Multiplying those values gives us the number of cycles saved, and thus, the speedup achieved. We proceed with this approach over the 30 images from the AxBench suite since reuse hits vary from image to image.

The reasons for using the *sobel* benchmark, instead of, for example, applying approximation in FP operations, was first to show that the technique can be applied to other applications, and second because there are well-established quality metrics for image processing. In our study, the error metric used to assess the sobel benchmark's output quality was the root-mean-squared (RMS) pixel difference between the original and approxi-

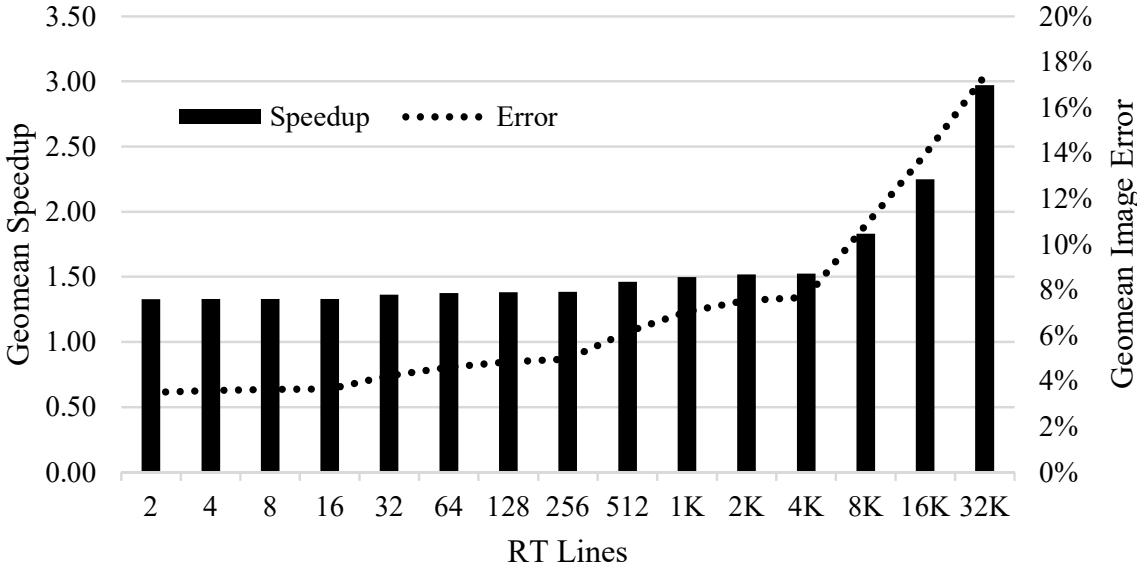
mated computations, normalized to the range 0-100%, already defined in the AxBench suite (YAZDANBAKHSH et al., 2016), which made the quality measurement easier and more trustworthy.

5.3.2 Performance

Fig. 5.11 shows geomean speedup and error rate considering 30 distinct images in the *sobel* benchmark. 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 (YAZDANBAKHSH et al., 2016)) for every table size up to 4K 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).

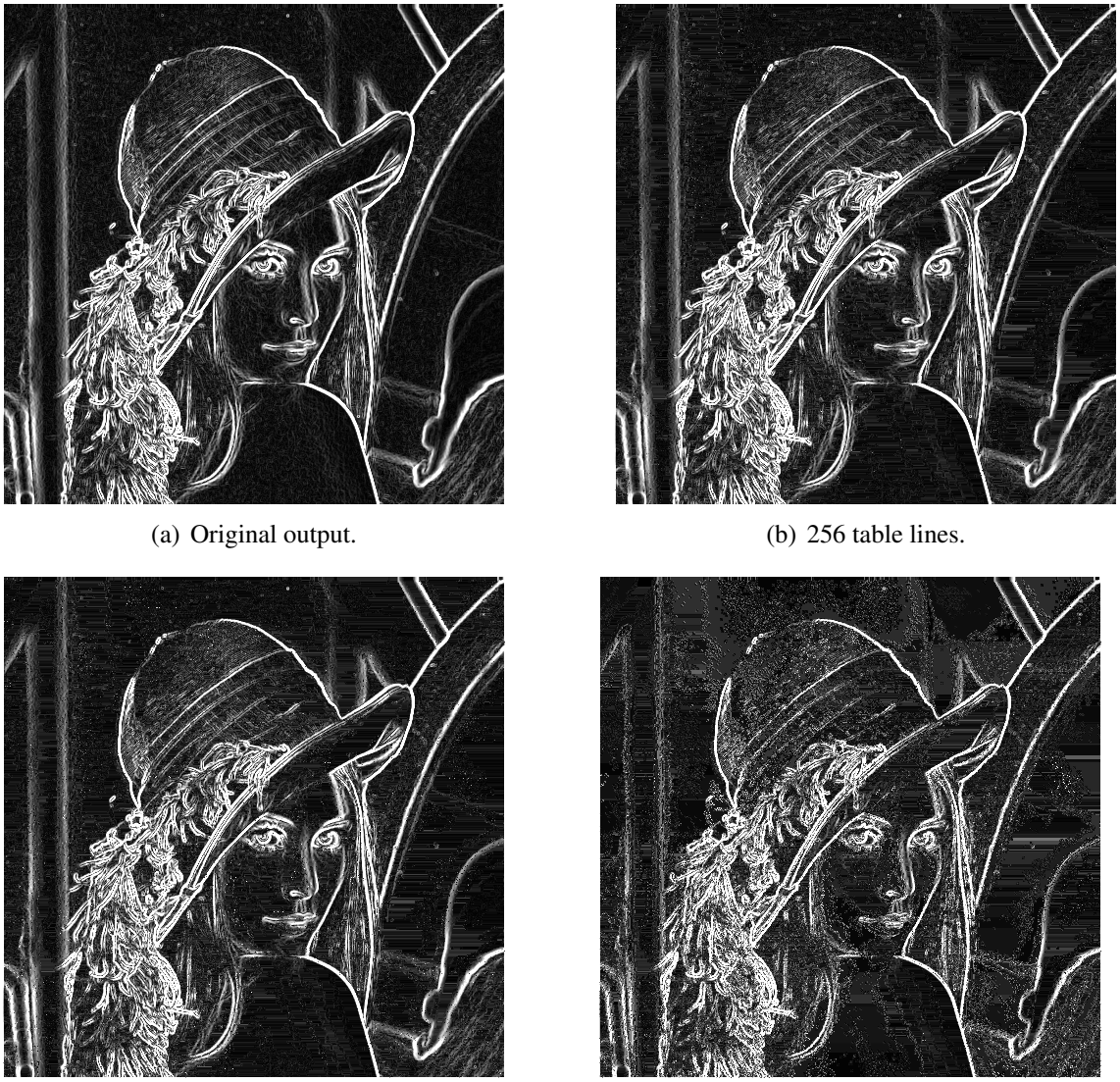
An example of possible outputs for an image is presented in Figure 5.12. It shows, for different table sizes, the decay in quality as the RT grows. As the kernel sweeps the image by fixing a line and moving on its columns from left to right, one can see traces of approximation in the same direction all over the picture. Again, larger tables increase

Figure 5.11: Speedup for approximate sobel image filter.



Source: The Author

Figure 5.12: The approximate *sobel* filter dropping 4 Least Significant Bits.



(a) Original output. (b) 256 table lines.

(c) 4K table lines. (d) 32K table lines.

Source: The Authors

reuse and error rates. This can be mitigated by changing the amount of LSBs dropped from input pixels when generating the *tag*, creating a better approximation at the cost of performance.

5.4 Resource Usage

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. This collected information illustrates *precise* reuse in the single-

core environment, the only technique actually implemented in VHDL. *Approximate* reuse, as we stated in Chapter 4.4, brings minimal area overheads over the results here presented, since most of it uses the same hardware infrastructure of *precise* reuse. Therefore, the system with *approximate* reuse is very similar in terms of area occupation when compared to the one with *precise* reuse only. We discuss resource impacts of the single-core first, and after we state benefits that the multi-core reuse can provide.

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 (LUNGDREN, 2014), and ρ-VEX with a hardware sobel filter. They all were synthesized following the methodology explained above, except the hardware implementation of sobel, which data was taken from (CHAPLE; DARUWALA, 2014), covering Virtex 5 FPGAs only. Cases as ρ-VEX with a double precision FPU and ρ-VEX with a hardware sobel filter were estimated by the simple sum of separate synthesis, since they are case studies to demonstrate how using a single and, as far as possible, *generic* reuse hardware can be cheaper than dedicated accelerators, especially for LUTs and registers. To an even more accurate measurement, these accelerators should be actually coupled to the processor itself.

Table 5.2 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) the results consider the size needed for the implementation of both modes (i.e., it considers the size for precise reuse), so it is possible to switch between them.

Table 5.2: 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%
Virtex 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%

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 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 32K 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 regarding 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 mechanism encompasses more system features, enabling performance gains when any new robust hardware module does not fit.

Therefore, our approach 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 an FPU were implemented in hardware.

For these multi-core scenarios, we have shown (Section 5.2.3) that sharing the RT can increase speedup of applications, even if using proportionally less total BRAM. Additionally, we would be avoiding the replication of hardware to control the RU for every core in the system, since a single centralized unit would be only one needed. This, however, would not change the fact that each core must modify its pipeline to comport reuse information (as described in Chapter 4). To get accurate results on area, we aim to implement the shared RU in VHDL as future work, since after our performance simulations, the technique has proven worthy.

6 CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

This work presented function reuse towards VLIW soft-core processors, presenting an alternative to costly hardware accelerators, by using possibly idle BRAMs. We showed how the technique can be implemented in the real-life ρ-VEX processor. We demonstrate that reusing functions can increase performance in processors while being cheap in terms of resources. More importantly, we extended the idea to a multi-core environment, analyzing impacts in performance and filling an important gap of research on *computational reuse* as a whole. This investigation indicated that it is possible to increase performance with RTs even smaller than in the former case. Moreover, we detailed and examined how the idea of approximate computing can be combined with function reuse, showing that, by the cost of quality, it is possible to increase reusability rates together with performance, which is promising to error-tolerant applications. Finally, we understand that our work has presented a vast set of results over the function reuse technique, and can be a good source of knowledge for future research on the topic.

6.2 Work Limitations

Our work covers three different techniques in the function reuse spectrum. However, we do it without exhaustive exploration of all possible trade-offs. First, we have considered a direct-mapped RT accessed by a hash function with an update-always policy. Also, the hash function itself was set as in a similar work but can be tuned to try to increase reuse rates. In our multi-core approach, while the RU is locked all other reuse requests are dropped. Otherwise, we could analyze performance and implementation impacts if we added a queue to attend to all requests. Also, we only considered multiple programs running together, and did not investigate multi-threaded programs. Finally, we present results for a few case studies, which could be extended to provide a stronger argumentation.

6.3 Future Work

As future work, we aim to overcome some of our research limitations. One of them being the RT policy. We understand that including a better replacement policy in the RT (as replacing the least recently used entry, for example) would be promising for our technique. This could be accompanied with associativity in the RT. Also, as FORMOSA is a software-level simulator, it is feasible to change it to observe impacts of a reuse request queue. If results improve considerably, we can then think of implications in hardware. In any case (with or without a reuse request queue) we intend to implement a multi-core with a shared RU in VHDL to get accurate resource usage results. Finally, an essential step to the future work is to embrace more applications, evidencing the possibilities for applying a function reuse scheme like ours.

REFERENCES

ALVAREZ, C.; CORBAL, J.; VALERO, M. Fuzzy Memoization for Floating-Point Multimedia Applications. **IEEE Transactions on Computers**, IEEE Computer Society, v. 54, n. 7, p. 922–927, jul 2005. ISSN 0018-9340.

AUSTIN, T.; LARSON, E.; ERNST, D. SimpleScalar: An infrastructure for computer system modeling. **Computer**, IEEE, v. 35, n. 2, p. 59–67, 2002.

BOPPANA, V. et al. UltraScale+ MPSoC and FPGA families. In: IEEE. **Hot Chips 27 Symposium (HCS), 2015 IEEE**. [S.l.], 2015. p. 1–37.

BRANDALERO, M. et al. Accelerating error-tolerant applications with approximate function reuse. **Science of Computer Programming**, 2017. ISSN 0167-6423. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0167642317300965>.

BUELL, D. et al. Guest Editors' Introduction: High-Performance Reconfigurable Computing. **Computer**, IEEE Computer Society, v. 40, n. 3, p. 23–27, mar 2007. ISSN 0018-9162. Available from Internet: <http://ieeexplore.ieee.org/document/4133992/>.

CHAPLE, G.; DARUWALA, R. D. Design of Sobel operator based image edge detection algorithm on FPGA. In: IEEE. **Communications and Signal Processing (ICCSP), 2014 International Conference on**. [S.l.], 2014. p. 788–792.

CITRON, D.; FEITELSON, D.; RUDOLPH, L. Accelerating multi-media processing by implementing memoing in multiplication and division units. In: ACM. **ACM SIGPLAN Notices**. [S.l.], 1998. v. 33, n. 11, p. 252–261.

DAS, R. et al. Performance and power optimization through data compression in Network-on-Chip architectures. In: **2008 IEEE 14th International Symposium on High Performance Computer Architecture**. IEEE, 2008. p. 215–225. ISBN 978-1-4244-2070-4. ISSN 1530-0897. Available from Internet: <http://ieeexplore.ieee.org/document/4658641/>.

ESMAEILZADEH, H. et al. Neural Acceleration for General-Purpose Approximate Programs. In: **Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture**. Washington, DC, USA: IEEE Computer Society, 2012. (MICRO-45), p. 449–460. ISBN 978-0-7695-4924-8. Available from Internet: <http://dx.doi.org/10.1109/MICRO.2012.48>.

FISHER, J. A. **Very long instruction word architectures and the ELI-512**. [S.l.]: ACM, 1983.

FLETCHER, B. H. FPGA embedded processors. In: **Embedded Systems Conference**. [S.l.: s.n.], 2005. p. 18.

GEER, D. Chip makers turn to multicore processors. **Computer**, v. 38, n. 5, p. 11–13, 2005. ISSN 0018-9162. Available from Internet: <http://ieeexplore.ieee.org/document/1430623/>.

GONZÁLEZ, A.; TUBELLA, J.; MOLINA, C. Trace-level reuse. In: IEEE. **Parallel Processing, 1999. Proceedings. 1999 International Conference on**. [S.l.], 1999. p. 30–37.

GUSTAFSSON, J. et al. The Mälardalen WCET Benchmarks: Past, Present And Future. In: LISPER, B. (Ed.). **10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)**. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010. (OpenAccess Series in Informatics (OASIcs), v. 15), p. 136–146. ISBN 978-3-939897-21-7. ISSN 2190-6807. The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7. Available from Internet: <http://drops.dagstuhl.de/opus/volltexte/2010/2833>.

HALL, M.; MCNAMEE, J. P. Improving software performance with automatic memoization. **Johns Hopkins APL Technical Digest**, JOHN HOPKINS UNIV APPLIED PHYSICS LABORATORY, v. 18, n. 2, p. 255, 1997.

HAN, J.; ORSHANSKY, M. Approximate computing: An emerging paradigm for energy-efficient design. In: **2013 18th IEEE European Test Symposium (ETS)**. [S.l.: s.n.], 2013. p. 1–6. ISSN 1530-1877.

HAUSER, J. **SoftFloat**. 2002. Available from Internet: <http://www.jhauser.us/arithmetic/SoftFloat.html>.

Hewlett-Packard Laboratories. **VEX Toolchain**. 2009. Available from Internet: <http://www.hpl.hp.com/downloads/vex/>.

HUANG, J.; LILJA, D. J. Exploiting basic block value locality with block reuse. In: IEEE. **High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On**. [S.l.], 1999. p. 106–114.

JOST, T. T.; NAZAR, G. L.; CARRO, L. Scalable memory architecture for soft-core processors. In: **2016 IEEE 34th International Conference on Computer Design (ICCD)**. [S.l.: s.n.], 2016. p. 396–399.

JÓŹWIAK, L. Advanced mobile and wearable systems. **Microprocessors and Microsystems**, Elsevier, v. 50, p. 202–221, 2017.

KALE, V. Using the MicroBlaze Processor Core to Accelerate Embedded System Development Using the MicroBlaze Processor to Accelerate Cost-Sensitive Embedded System Development. **WP469**, n. 1, 2016. Available from Internet: <www.xilinx.com>.

KAVI, K. M.; CHEN, P. Dynamic function result reuse. In: **Proceedings of the 11th International Conference on Advanced Computing (ADCOM-2003)**. [S.l.: s.n.], 2003. p. 17–20.

KERAMIDAS, G.; KOKKALA, C.; STAMOULIS, I. Clumsy Value Cache: An Approximate Memoization Technique for Mobile GPU Fragment Shaders. In: **1st Workshop On Approximate Computing (WAPCO 2015)**. Amsterdam: [s.n.], 2015. p. 6.

KUON, I.; ROSE, J. Measuring the gap between FPGAs and ASICs. **IEEE transactions on computer-aided design of integrated circuits and systems**, IEEE, v. 26, n. 2, p. 203–215, 2007.

LATTNER, C.; ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In: IEEE COMPUTER SOCIETY. **Proceedings of the**

international symposium on Code generation and optimization: feedback-directed and runtime optimization. [S.l.], 2004. p. 75.

LAY, D. C. **Linear algebra and its applications**. [S.l.]: Addison Wesley, Boston, 1999.

LONG, G. et al. Minimal multi-threading: Finding and removing redundant instructions in multi-threaded processors. In: IEEE COMPUTER SOCIETY. **Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.], 2010. p. 337–348.

LUDLOW, D. **What's the difference between Core i3, i5 and i7 processors?** 2014. Available from Internet: <http://www.expertreviews.co.uk/pcs/cpus/1400962/whats-the-difference-between-core-i3-i5-and-i7-processors>.

LUNGDREN, D. **FPU Double VHDL**. 2014. Available from Internet: <http://opencores.org/project,fpu{_}dou>.

MCKEOWN, M.; BALKIND, J.; WENTZLAFF, D. Execution Drafting: Energy Efficiency Through Computation Deduplication. In: **Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture**. Washington, DC, USA: IEEE Computer Society, 2014. (MICRO-47), p. 432–444. ISBN 978-1-4799-6998-2. Available from Internet: <http://dx.doi.org/10.1109/MICRO.2014.43>.

MOLINA, C.; GONZDALEZ, A.; TUBELLA, J. Trace-level speculative multithreaded architecture. In: **Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors**. [S.l.: s.n.], 2002. p. 402–407. ISSN 1063-6404.

NIOS, I. I. **Processor Reference Handbook**. [S.l.]: Altera, 2009.

OLUKOTUN, K.; HAMMOND, L. The future of microprocessors. **Queue**, ACM, v. 3, n. 7, p. 26–29, 2005.

PARULKAR, I. et al. OpenSPARC: An open platform for hardware reliability experimentation. In: **Fourth Workshop on Silicon Errors in Logic-System Effects (SELSE)**. [S.l.: s.n.], 2008.

QUALCOMM. **Snapdragon 835 Mobile Platform**. 2017. Available from Internet: <https://www.qualcomm.com/products/snapdragon/processors/835>.

Research Grand View. **FPGA (Field-Programmable Gate Array) Market Analysis By Application (Automotive, Consumer Electronics, Data Processing, Industrial, Military And Aerospace, Telecom) And Segment Forecasts, 2014-2024.** [S.l.]: Grand View Research, 2016. 130 p. ISBN 9781680381337.

SASTRY, S.; BODIK, R.; SMITH, J. Characterizing coarse-grained reuse of computation. In: **3rd ACM Workshop on Feedback Directed and Dynamic Optimization**. [S.l.: s.n.], 2000. v. 273, p. 274.

SCOTT, J. et al. Designing the Low-Power M• CORE TM Architecture. In: **Power driven microarchitecture workshop**. [S.l.: s.n.], 1998. p. 145–150.

SINHA, S.; ZHANG, W. Low-Power FPGA Design Using Memoization-Based Approximate Computing. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 24, n. 8, p. 2665–2678, aug 2016. ISSN 1063-8210.

SKLYAROV, V. et al. **Synthesis and Optimization of FPGA-Based Systems**. Cham: Springer International Publishing, 2014. 432 p. (Lecture Notes in Electrical Engineering, v. 294). ISBN 978-3-319-04707-2. Available from Internet: <http://link.springer.com/10.1007/978-3-319-04708-9>.

SODANI, A.; SOHI, G. S. **Dynamic instruction reuse**. [S.l.]: ACM, 1997. 194–205 p. ISSN 0163-5964. ISBN 0897919017.

STITZ, C.; ZEAGER, J. **Precalculus**. [S.l.]: Stitz Zeager Open Source Mathematics, 2013.

STOKES, J. **Crusoe explored | Ars Technica**. 2000. Available from Internet: <https://arstechnica.com/features/2000/01/crusoe/3/>.

SURESH, A. et al. Intercepting functions for memoization: a case study using transcendental functions. **ACM Transactions on Architecture and Code Optimization (TACO)**, ACM, v. 12, n. 2, p. 18, 2015.

TENDLER, J. M. et al. Power4 system microarchitecture. **IBM Journal of Research and Development**, IBM, v. 46, n. 1, p. 5–25, 2002.

TONG, J. G.; ANDERSON, I. D. L.; KHALID, M. A. S. Soft-Core Processors for Embedded Systems. In: **2006 International Conference on Microelectronics**. IEEE, 2006. p. 170–173. ISBN 1-4244-0764-8. Available from Internet: <http://ieeexplore.ieee.org/document/4243676/>.

Van Berkel, C. H. Multi-core for mobile phones. In: EUROPEAN DESIGN AND AUTOMATION ASSOCIATION. **Proceedings of the Conference on Design, Automation and Test in Europe**. [S.l.], 2009. p. 1260–1265.

WOLF, W.; JERRAYA, A. A.; MARTIN, G. Multiprocessor system-on-chip (MPSoC) technology. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 27, n. 10, p. 1701–1713, 2008.

WONG, S.; Van As, T.; BROWN, G. ρ-VEX: A reconfigurable and extensible softcore VLIW processor. In: IEEE. **ICECE Technology, 2008. FPT 2008. International Conference on**. [S.l.], 2008. p. 369–372.

XILINX; INC. **7 Series FPGAs Memory Resources User Guide (UG473)**. 2016. Available from Internet: <https://www.xilinx.com/support/documentation/user{_}guides/ug473{_}7Series{_}Memory{_}R>.

Xilinx Inc.; XILINX; INC. **7 Series FPGAs Data Sheet: Overview (DS180)**. 2017. Available from Internet: <https://www.xilinx.com/support/documentation/data{_}sheets/ds180{_}7Series{_}Ove>.

YAZDANBAKHSH, A. et al. AxBench: A Benchmark Suite for Approximate Computing. **IEEE Design and Test**, n. special issue on Computing in the Dark Silicon Era 2016, 2016.

**

APPENDIX A — PROJECT DESCRIPTION (TG1)

Function Reuse on a Multi-Core VLIW Soft-Core Processor

Pedro Henrique Exenberger Becker[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

Abstract. *Modern processors contain several specific hardware modules and multiple cores to ensure performance on a wide range of applications. However, for FPGA-based processors, those modules may not fit in the device. To alleviate this performance overhead, we exploit the fact that logic-driven designs usually underuse available FPGA BRAMs. We propose a low-cost hardware-based function reuse mechanism, which can optimize software execution, in a multi-core design. This is accomplished by saving the inputs and outputs of the most recurring functions in a BRAM-based reuse table, so they can be reused in the next function calls, skipping actual execution, and improving performance.*

1. Introduction

The implementation of processors in Field-Programmable Gate Arrays (FPGAs), known as soft-core processors, has known benefits such as architecture customization, hardware acceleration, and obsolescence mitigation [Fletcher 2005]. 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 users with performance gains in dedicated tasks.

At the same time, nowadays systems require high performance for a wide range of applications, which increases the demand for resources. The use of multi-core processors (e.g., ARM Cortex-A53 [Boppana et al. 2015]) together with dedicated hardware like Floating-Point Units (FPUs), security and cryptography modules, and coders/decoders for multimedia, are commonly adopted in any modern design, such as in Multiprocessor Systems on Chip (MPSoCs) [Wolf et al. 2008]. However, FPGA designs require more area and energy compared to Application Specific Integrated Circuits (ASICs) [Kuon and Rose 2007]. Therefore, in many cases, a number of resources available in an FPGA is a limiting factor. In case specialized hardware can't fit inside the FPGA when implementing a system with soft-processors benefits, some of its functionalities must be mapped into the software domain, which is significantly slower.

To alleviate this constraint and increase the design space, this work proposes a low-cost and generic hardware solution to speed up specific software parts, within a multi-core design without the need for implementing dedicated hardware components. Our study leverages the fact that, most of the time, Block Random Access Memories (BRAMs) are not used in the same proportion as Look-Up Tables (LUTs) and Registers in FPGA logic-driven designs (e.g., soft-cores), to adopt a function reuse scheme. The main idea presented here is to automatically store the input and output arguments of a set of functions that belong to a given library in a BRAM Reuse Table (RT). When the same function call repeats and the values of the arguments match, the output value can be directly fetched from the RT, avoiding re-calculation, and improving software performance. Since there is a software library instead of an ASIC to perform a particular

function, a significant number of LUT and registers are saved; and, by using our technique, the performance of such software library can be improved using BRAMs, which would otherwise be idle.

Additionally, given a soft-core multi-processor design, it is possible to share the RT among cores. Thus, programs that are simultaneously running on the soft-core can both update the RT as they calculate function results, and fetch results calculated by other processes from the RT. At the same, the introduction of a shared RT is much cheaper than if we would introduce a dedicated RT for each core. By using this approach, we expect to accelerate multiple cores while reducing the proportional impact of an RT introduction in the FPGA design.

The remaining of this work is organized as follows: Section 2 presents a Background over topics related to the research. Section 3 discusses Related Work. Next, section 4 states the Work Proposal, detailing the reuse scheme and the goals of the work. Section 5 presents the proposed Methodology. Section 6 argue about the Schedule for the upcoming steps. Finally, Section 7 debates Conclusions.

2. Background

In the upcoming sub-sections, we introduce some important concepts that support our study. They will be briefly defined together with its role in this work.

2.1. FPGA

At the beginning of the computation era, general-purpose processors used to be the only option that a systems designer had to accomplish complex design specifications rapidly. Thus, the designers' decision over the system hardware was limited to choosing the most appropriate processor and the grain on which the designer had to define functionalities was in the level of programs. After, in the year of 1980, the first ASICs were developed, giving the possibility for the designer to determine specific hardware implementation of the project. FPGAs emerged in this scenario as an intermediary solution between processors and ASICs, providing better control over the hardware implementation compared to processors, and more flexibility, testability, and time-to-market over ASIC. With its differentials, the FPGA market had reached the estimation of USD 6.36 billion by the year of 2015 with continuous growth expectation [Research Grand View 2016]. Thereby, this research is based on a technology that is already established and has a solid market.

FPGA gained market share with its uniqueness: it rests on the idea of a reconfigurable circuit. Figure 1 depicts a traditional FPGA architecture. It is an array of Configurable Logic Blocks (CLBs), routing channels, BRAMs and Digital Signal Processing (DSP) units. The reconfigurability is a result of the programmability of the CLBs and routing channels, as we present below.

A CLB is the fundamental component on the FPGA architecture. Inside each CLB, there are a set of small tables (LUTs). With these tables, the FPGA can implement logical functions over the CLB inputs (using the LUTs as a truth-table). Thus, combining many CLBs through the routing channels can lead to implementations of very complex logical functions. Moreover, the LUTs can be used as memory elements for small data amount. Since both CLB and routing channels are programmable, the circuit that an FPGA implements can be changed by reprogramming those components.

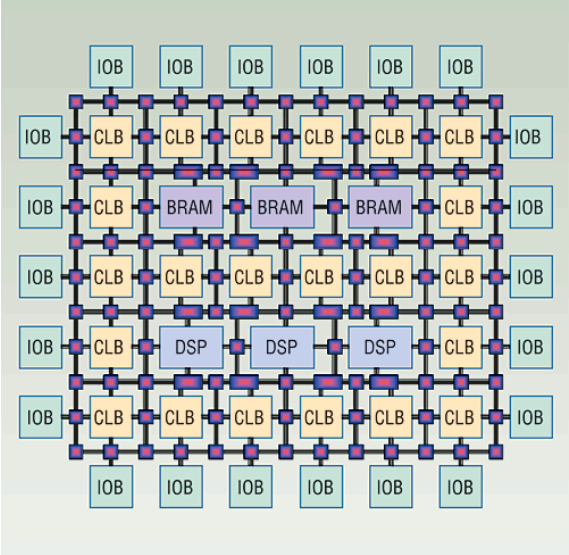


Figure 1. A typical FPGA Architecture [Buell et al. 2007]

Aside CLBs and routing channels, FPGA also has Input/Output Blocks (IOBs), to connect the FPGA with the outside world, and embedded blocks. Traditional architectures contain both embedded DSPs units, to speed up costly operations that are implemented by cascades of truth-tables (e.g., multiplications), and BRAMs, which will be detailed in the following subsection.

2.1.1. Block Random Access Memories (BRAM)

BRAMs are embedded memory blocks used for storing large sets of data more efficiently than by LUTs, and are widely available in modern FPGAs. For example, the Xilinx 7 series FPGAs contains from 5 to 1880 dual-port BRAMs, depending on the model, each storing 36Kb of data. These blocks can be divided into two independent 18Kb BRAMs. In both cases, dual-port is assured, and each port is completely independent of another, sharing only the stored data [Xilinx Inc. 2017].

These embedded blocks can also be configured in different associations (e.g., 32K 1-bit lines, 16K 2-bit lines, ..., 1K 32-bit lines, 512 64-bit lines), and can be interconnected to create wider and deeper memory structures [Sklyarov et al. 2014]. Finally, both read and write operations are synchronous, requiring an active clock edge.

Since logic-driven designs, as soft-core processors, generally underuse available BRAMs, we propose to better occupy those components by implementing the RT for enhancing applications' performance.

2.2. Soft-Core Processors

A soft-core processor is a hardware description language model of a specific processor that can be customized and synthesized for an ASIC or FPGA target [Tong et al. 2006]. In this study, however, we consider only FPGA-based soft-cores.

FPGA-based soft-cores became popular as they bring advantages such as (i) ar-

chitecture customization, since FPGA allows non-standard implementation, according to the design requirements; (ii) obsolescence mitigation, as the hardware description perpetuates while hardware technologies advance; (iii) cost reduction, considering that multiple components can be replaced with a single FPGA; and (iv) hardware acceleration, since specific algorithms can be implemented in hardware, for example, to achieve better performance [Fletcher 2005].

There is a variety of commercial and academic soft-core processors as a demonstration of its representativeness. Examples from the industry comprehend Xilinx *MicroBlaze* soft processor, an Intellectual Property (IP) for Xilinx FPGAs [Kale 2016], the Altera Nios II [Nios 2009], and the open sourced hardware description of Sun's UltraSparc T1 and T2, which were released by the OpenSparc Project [Parulkar et al. 2008]. From academia, we highlight the ρ-VEX Very Long Instruction Word (VLIW) processor [Wong et al. 2008] from TU Delft.

2.3. Modern Architectures

The required performance of computing devices has increased as technology advances. The transistor's scaling improved processors frequency, while the exploitation of Instruction-Level Parallelism (ILP) increased processors throughput. However, increased clock rates dissipate more power, which became a barrier. At the same time, the ILP exploitation by deep pipelining and out-of-order processors seems to have reached a plateau [Das et al. 2008].

To overcome the above challenges and to guarantee computers ascendant performance, various solutions were proposed. We detail three solutions in the upcoming sub-sections, which are strongly related to this work: Multi-Core Processors, VLIW Processors, and MPSoCs.

2.3.1. Multi-Core Processors

Optimizing a single core processor with pipeline, ILP exploitation and out-of-order execution became insufficient. Multi-core processors were proposed, observing that complex systems usually execute multiple tasks. Thus, in a multi-core environment, various tasks could be distributed among multiple processors, increasing the overall throughput of the system. Figure 2 presents a quad-core Intel i7 overview as example. Each core can execute independently of others and communicates by using the shared cache memory.

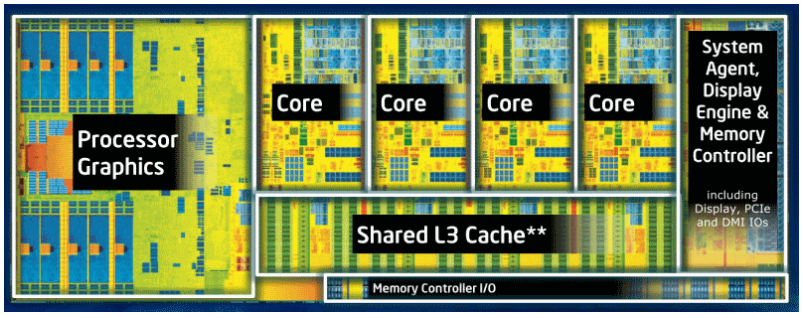


Figure 2. A four-core Intel i7 4770K [Ludlow 2014]

The first commercial multi-core processor was the IBM Power4 [Tendler et al. 2001] released in 2001. After, other companies such as Intel, AMD, and Sun also turned to multi-core production [Geer 2005]. Ever since, multi-core designs became almost a standard for General Purpose Processors (GPPs), from Intel Pentium D up to the latest Intel i7 Series. Not enough, multi-core processors are highly used in nowadays embedded systems as smartphones [Van Berkel 2009].

2.3.2. VLIW Processors

The extraction of ILP at execution time enabled processors performance gains but introduced more hardware (and complexity) given the logic required to find parallel instructions [Olukotun and Hammond 2005] dynamically. As alternative to on-the-fly ILP exploitation, the VLIW processors were proposed [Fisher 1983].

A VLIW processor contains multiple execution pipelines, the issue-slots, so that it can execute more than one instruction at a time. The advantage of using these processors is that the instruction parallelism is extracted by the compiler. Thus, a VLIW processor can benefit from ILP while maintaining a simple microarchitecture. The parallel instructions are disposed inside a *very long instruction word*, in a set of independent instructions that can be executed concurrently without any concern by the processor. The arrangement of the instruction even maps each instruction with the issue-slot on which it will be executed.

Because of its simple organization alongside its ILP exploitation, the VLIW processors are powerful yet less resource-consuming in comparison to a complex superscalar. These factors make the VLIW a good architecture option for resource constrained FPGA-based soft-core processors.

2.3.3. MPSoCs

Some computer tasks, like real-time video encoding, for example, execute a complex algorithm with very high throughput demand. When that occurs, using a GPP may not be the best option, either for performance or energy consumption, since a specialized hardware can achieve better performance with more efficiency.

Let us consider the case of smartphones and tablets as an example. They are required to execute a broad range of tasks with minimum battery consumption. For this reason, these devices have specific hardware modules in their main chip die to run high-performance applications efficiently [Jóźwiak 2017] and work together with the GPP processors. This chip arrangement is known as MPSoCs. Figure 3 depicts an overview of the Intel Atom Z3770, used in some tablet models, which contains dedicated hardware for processing audio, graphics, camera image, security, and several other specialized modules.

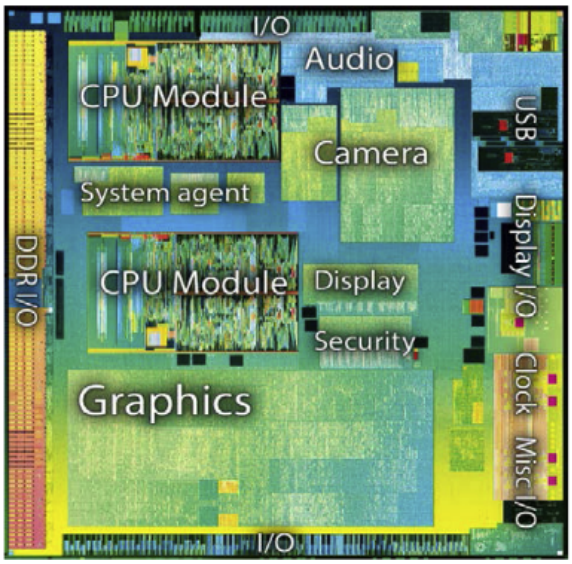


Figure 3. Intel Atom Z3770 [Jóźwiak 2017]

2.3.4. Reuse

Here we introduce the concept of reuse in the computing scope. A more detailed overview among techniques and approaches will be covered in the Section 3 (Related Work).

Reuse of computation is based on the observation that deterministic execution - where a set of inputs always leads to the same result - often repeats within programs. Reuse exploits this by saving input and result (output) values of those executions in a RT. The RT is searched when reentering a given snippet of execution, checking whether the current and the saved input values are the same. In case of a hit, the result (output value) from the RT is fetched faster than recomputing it.

For example, consider the hypothetical code bellow:

$$r.1 = r.6$$
$$r.2 = 1 + r.1$$
$$r.3 = 1 + r.2$$

Figure 4. Example of a Reusable Code Snippet

The inputs are the values which are read before been written; in the example, the register $r.6$ is the only input. The outputs are the written values, which will be read further, in the case $r.1, r.2, r.3$. Consider $r.6$ = 1 in a given execution. Therefore, after the code executes, $r.1 = 1, r.2 = 2, r.3 = 3$. Note that any eventual execution in which $r.6$ = 1 will imply in the same result in the outputs $r.1, r.2, r.3$. In this scenario, saving the input and output values for consulting in a new occurrence could skip execution of three instructions.

2.3.4.1. Function Reuse

A particular case for reuse is when the evaluation of reuse occurs in the grain of functions. In this case, the function parameters are the inputs, and the values returned by functions are the outputs.

The following example illustrates a simple scenario where function reuse could be used:

$$a = sin(\pi)$$
$$\vdots$$
$$b = sin(\pi)$$

Figure 5. Example of a Reusable Function Result

In this case, the second execution of sin could be skipped, given a repetition of the inputs (π in this case). Naturally, in this simple case, the explicit re-operation of $sin(\pi)$ could be avoided by better programming or even compiler optimization. Note, however, that in a more realistic situation the input parameters can, and often will, be variables, whose values are unknown at compile time and unpredictable by the programmer.

3. Related Work

A variety of works has discussed reuse of computation [Sastry et al. 2000]. Implementations vary from software (where reuse is also known as memoization [Hall and McNamee 1997]) to hardware-based solutions and cover different granularities of code. In [Sodani and Sohi 1997], dynamic instruction reuse is presented with execution-driven simulation. The goal is to avoid re-execution of instructions in an out-of-order processor. Source registers of the instructions are the inputs, and the instruction result is the output. The scheme is enhanced with control of dependency links among instructions, providing reuse of a set of dependent instructions. The authors in [Citron et al. 1998] proposed the reuse of Floating-Point (FP) instructions, only. They focused on multimedia applications, where they claim that reusability is more promising. On each function unit that takes more than a cycle to execute (like an FP divider or multiplier), a MEMO-TABLE is used to store the results. Average speed up between 8% and 22% is achieved. Despite a hardware scheme is discussed, the results are taken from an instruction level simulator.

Reuse of a set of instructions within a basic block is considered in [Huang and Lilja 1999] and simulated using SimpleScalar [Austin et al. 2002]. Any source operand (from registers or memory) that has not been previously written inside a basic block is considered part of the input. The last value written in any register or memory destination is considered as part of the output. Their work shows performance improvements of up to 14%. A similar system is proposed over trace level (a set of sequential basic blocks) in [González et al. 1999]. In this case, less reusability is found compared to instruction reuse, but more speedup is obtained.

The authors in [Kavi and Chen 2003] introduced the concept of dynamic function result reuse. In this case, only pure functions (global variables free, no I/O requirement, nor any change in the global state of the program) can be reused, in a form that the return value is only a consequence of the input parameters of the function. The authors verify the impact of (i) reuse buffer size; (ii) reuse buffer associativity; and (iii) amount of input parameters of functions. The study presented from 10% to 67% of reusability on a variety of applications, supporting the use of the function reuse concept. Finally, the authors in [Suresh et al. 2015] implemented function reuse in the interface between programs and operating system. Their mechanism intercept calls to the dynamically linked math library by preloading a memoized version of it. 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).

Our work is the first to consider reuse specifically targeted to FPGAs, exploiting its unique components and intrinsic characteristics, like the existence of BRAMs and its specific design constraints. By presenting function reuse in FPGA for soft-core designs, we can open new possibilities for design space exploration and new trade-offs for HW/SW co-design in such devices. For instance, low-price FPGA may regain space in project decision, since our approach allows for performance gains with a low overhead in LUTs at the price of BRAM occupation, which are many times underutilized.

Additionally, we propose to cover reuse with a shared RT in a multi-core environment, which was never proposed before. Such analysis can update the knowledge of reusability in modern designs, and also explain the behavior of reuse when the RT is populated simultaneously from different applications. From the resource consumption view, a shared RT demands proportionally fewer resources from the overall design than when it is dedicated to a single core.

We also present, to the best of our knowledge, the first hardware implementation of such technique. Through this, we have 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 abstract layers, leading to a solution independent of user space or operating systems, which also covers bare metal embedded systems, wherein many times operating systems are not available.

4. Work Proposal

This work proposes to implement a Function Reuse Unit (RU) compatible with the ρ-VEX VLIW soft-core processor. The reuse scheme will be inserted in a multi-core environment where multiple ρ-VEX soft-core processor share the reuse module. The intent is to speed up software solutions (e.g., software libraries for specific tasks) in cases when adding specific hardware to the FPGA target is not possible due to area constraints, compromising the design scalability. We propose to implement the RU by managing an RT that exploits idle BRAMs in the FPGA design, allocating FPGA resources smartly.

The RT will be populated during the execution of the applications. Modifications in the processor description will be made to detect function calls, to gather the input parameters of functions, to gather output values of functions, and to skip execution when reuse is possible. Information collected from the processor will be passed to the RU, which will update the RT data. The RU will perform over a set of marked functions.

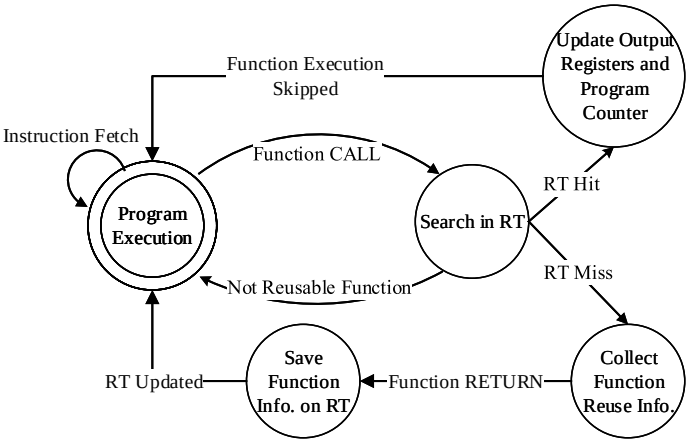


Figure 6. The functioning of the Reuse Unit

Therefore, it will be possible for the designer to choose the functions on which reuse will be applied, including functions that are used in more than one application (e.g., reuse the results of the function $sin()$ in multiple mathematical applications).

To exemplify, Figure 6 depicts a diagram of the functioning of the proposed RU for a single application. It represents that program executes normally until a is call instruction to a *reusable function* (i.e., the functions marked to be reused by design) is found. For the first call occurrence of a function and its inputs, no information will be available in the RT; thus an RT miss occurs. To include the function execution information in the RT, the RU waits for the function to end, so the outputs are available. With all information, a reuse entry (see Figure 7) is saved in the RT for further consultation. In the following function calls, the function identification and the input parameters are used to search in the RT. When an RT hit occurs, the matched entry contains the output values. These output values are used to update the processor context, skipping actual execution of the function. Note, as the scheme depicts, that the RU only consider functions marked as *reusable*. Also, since the table is finite, there will be replacement of entries as the applications execute.

Although the example has presented the RU functioning with a single application, this work intents to share the RU (and thus, the RT) among multiple cores. Figure 8 depicts this scenario. In this case, multiple applications will consult and populate the RT concurrently. This enables applications to benefit from both internal redundancies

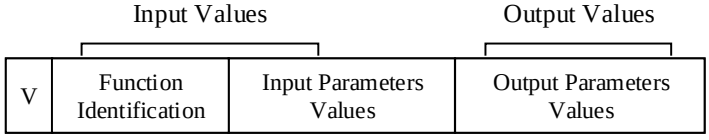


Figure 7. A Reuse Table Entry

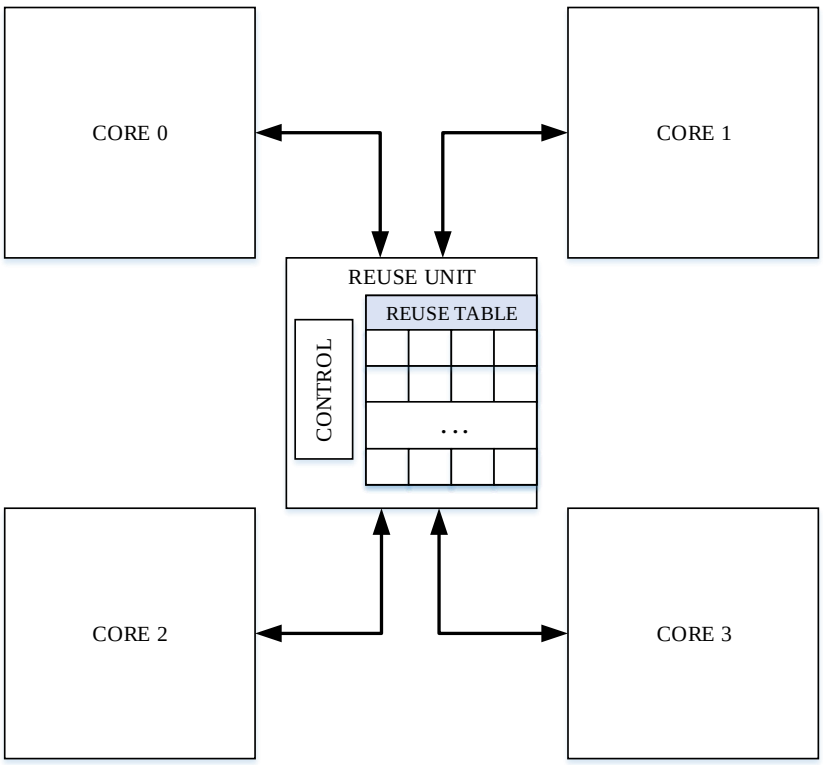


Figure 8. A Multicore Environment with a Shared Reuse Unit

(repetitions inside the same application) and external redundancy (computation that would be repeated in multiple applications but can be executed by one and reused by the others).

Finally, it is expected from this work to present results regarding performance speedup, as well as the FPGA components' usage. The results should be discussed, pondering pros and cons of our technique, and comparing it with related works.

5. Methodology

In the following sub-sections, we detail the methodology for each step of the research.

5.1. Function Reuse Module Implementation

As our work proposes a RU to be attached to a real processor hardware description, a RU will be implemented in VHDL, and Mentor Graphics Modelsim 10 will be used for simulations and tests. At first, the functioning of the unit will be tested in a single-core processor and only later in a multi-core environment.

5.2. Extraction of Results

Once implemented, the results of the reuse scheme must be extracted so one can analyze its potential. The results of performance will be evaluated by comparing the number of cycles to execute a set of applications with and without the RU. Also, since the RT is shared, its current data may vary depending on the applications running, varying the reusability as well. Therefore, different applications will potentially lead to different

Table 1. Schedule

	Jul.	Aug.	Sep.	Oct.	Nov.	Dec.
Reuse Unit Implementation in Single-core						
Reuse Unit Implementation in Multi-core						
Benchmarks Definition						
Extraction of Performance Results						
Extraction of FPGA Components' Usage						
Writing of the Final Term Paper						

performance speedups. The impact of the applications will be considered in the results discussion.

Moreover, the impact of attaching a RU to the design will be verified. Since the majority of the mechanism consists of a BRAM table, it is expected that our scheme uses less LUTs and registers than specific hardware accelerators would. To confirm this expectation, we will synthesize the system in Xilinx ISE/Vivado.

6. Schedule

Table 1 presents the schedule for the next steps of this work.

7. Conclusions

This work discussed the function reuse in the scope of soft-core processors implemented in FPGAs. It introduced background explanation and contextualized the work among related ones. A work proposal was stated to define the next steps of the research: to attach a RU in a multi-core environment of the ρ-VEX soft-core processor in an attempt to increase applications performance when specific hardware accelerators can't fit in the FPGA target. Especially, in multi-core environments, the addition of multiple cores may be a more generic solution for overall throughput than specific hardware, turning a RU into a generic and resource-rational approach since it uses idle BRAM for its construction. Additionally, a brief methodology was proposed, and also the schedule for the continuation of the work.

References

Austin, T., Larson, E., and Ernst, D. (2002). SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67.

Boppana, V., Ahmad, S., Ganusov, I., Kathail, V., Rajagopalan, V., and Wittig, R. (2015). UltraScale+ MPSoC and FPGA families. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*, pages 1–37. IEEE.

Buell, D., El-Ghazawi, T., Gaj, K., and Kindratenko, V. (2007). Guest Editors' Introduction: High-Performance Reconfigurable Computing. *Computer*, 40(3):23–27.

Citron, D., Feitelson, D., and Rudolph, L. (1998). Accelerating multi-media processing by implementing memoing in multiplication and division units. In *ACM SIGPLAN Notices*, volume 33, pages 252–261. ACM.

Das, R., Mishra, A. K., Nicopoulos, C., Park, D., Narayanan, V., Iyer, R., Yousif, M. S., and Das, C. R. (2008). Performance and power optimization through data compression in Network-on-Chip architectures. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 215–225. IEEE.

Fisher, J. A. (1983). *Very long instruction word architectures and the ELI-512*, volume 11. ACM.

Fletcher, B. H. (2005). FPGA embedded processors. In *Embedded Systems Conference*, page 18.

Geer, D. (2005). Chip makers turn to multicore processors. *Computer*, 38(5):11–13.

González, A., Tubella, J., and Molina, C. (1999). Trace-level reuse. In *Parallel Processing, 1999. Proceedings. 1999 International Conference on*, pages 30–37. IEEE.

Hall, M. and McNamee, J. P. (1997). Improving software performance with automatic memoization. *Johns Hopkins APL Technical Digest*, 18(2):255.

Huang, J. and Lilja, D. J. (1999). Exploiting basic block value locality with block reuse. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pages 106–114. IEEE.

Jóźwiak, L. (2017). Advanced mobile and wearable systems. *Microprocessors and Microsystems*, 50:202–221.

Kale, V. (2016). Using the MicroBlaze Processor Core to Accelerate Embedded System Development Using the MicroBlaze Processor to Accelerate Cost-Sensitive Embedded System Development. *WP469*, (1).

Kavi, K. M. and Chen, P. (2003). Dynamic function result reuse. In *Proceedings of the 11th International Conference on Advanced Computing (ADCOM-2003)*, pages 17–20.

Kuon, I. and Rose, J. (2007). Measuring the gap between FPGAs and ASICs. *IEEE transactions on computer-aided design of integrated circuits and systems*, 26(2):203–215.

Ludlow, D. (2014). What's the difference between Core i3, i5 and i7 processors?

Nios, I. I. (2009). Processor Reference Handbook.

Olukotun, K. and Hammond, L. (2005). The future of microprocessors. *Queue*, 3(7):26–29.

Parulkar, I., Wood, A., Hoe, J. C., Falsafi, B., Adve, S. V., Torrellas, J., and Mitra, S. (2008). OpenSPARC: An open platform for hardware reliability experimentation. In *Fourth Workshop on Silicon Errors in Logic-System Effects (SELSE)*.

Research Grand View (2016). *FPGA (Field-Programmable Gate Array) Market Analysis By Application (Automotive, Consumer Electronics, Data Processing, Industrial, Military And Aerospace, Telecom) And Segment Forecasts, 2014-2024*. Grand View Research.

Sastry, S., Bodik, R., and Smith, J. (2000). Characterizing coarse-grained reuse of computation. In *3rd ACM Workshop on Feedback Directed and Dynamic Optimization*, volume 273, page 274.

Sklyarov, V., Skliarova, I., Barkalov, A., and Titarenko, L. (2014). *Synthesis and Optimization of FPGA-Based Systems*, volume 294 of *Lecture Notes in Electrical Engineering*. Springer International Publishing, Cham.

Sodani, A. and Sohi, G. S. (1997). *Dynamic instruction reuse*, volume 25. ACM.

Suresh, A., Swamy, B. N., Rohou, E., and Seznec, A. (2015). Intercepting functions for memoization: a case study using transcendental functions. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(2):18.

Tendler, J. M., Dodson, S., Fields, S., Le, H., and Sinharoy, B. (2001). POWER4 System Microarchitecture.

Tong, J. G., Anderson, I. D. L., and Khalid, M. A. S. (2006). Soft-Core Processors for Embedded Systems. In *2006 International Conference on Microelectronics*, pages 170–173. IEEE.

Van Berkel, C. H. (2009). Multi-core for mobile phones. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1260–1265. European Design and Automation Association.

Wolf, W., Jerraya, A. A., and Martin, G. (2008). Multiprocessor system-on-chip (MPSoC) technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701–1713.

Wong, S., Van As, T., and Brown, G. (2008). ρ-VEX: A reconfigurable and extensible softcore VLIW processor. In *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 369–372. IEEE.

Xilinx Inc. (2017). 7 Series FPGAs Data Sheet: Overview (DS180).