Microprocessors and Microsystems 63 (2018) 237–248

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

BRAM-based function reuse for multi-core architectures in FPGAs

Pedro H. Exenberger Becker*, Anderson L. Sartor, Marcelo Brandalero, Antonio C. Schneider Beck

Instute of Informatics, Universidade Federal do Rio Grande do Sul 91501-970, Brazil

ARTICLE INFO

Article history:
Received 19 March 2018
Revised 19 July 2018
Accepted 19 September 2018
Available online 28 September 2018

Keywords:
Function-reuse
Soft-processors
Multi-core architectures
FPGA

ABSTRACT

Modern processors contain several specific hardware modules and multiple cores to ensure performance for a wide range of applications. In this context, FPGAs are frequently used as the implementation platform, since they offer architecture customization and fast time-to-market. However, many of them may not have the needed resources to implement all the necessary features, because of costs or complexity of the system to be implemented. When some needed functionalities do not fit in the target, they must be mapped into the much slower software domain. In this work, we exploit the fact these designs usually underuse their available BRAMs and propose a low-cost hardware-based function reuse mechanism for FPGAs, recovering some of the performance lost from the software part of applications that could not be implemented in hardware logic, with minimal impact on LUT usage. This is achieved by saving the inputs and outputs of the most frequently executed functions in a BRAM-based reuse table, so the next function executions with the same arguments can be skipped. This mechanism supports both precise and approximate modes and is evaluated with a 4-issue VLIW processor implemented in HDL, also considering a multi-core environment. Precise reuse, in single and multi-core scenarios, is assessed by running applications that use a software library to emulate floating point operations. Approximate reuse is evaluated over a single-core image-processing application that tolerates a certain level of error. Our scheme achieves 1.39× geomean speedup in the precise single-core, while the multi-core case demonstrates application improvements from 1.25× to 1.9× when we start sharing the reuse table. In the approximate scenario, we achieve 1.52× speedup with less than 10% error.

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 possibility to reprogram FPGAs guarantees good time-to-market, fast architectural 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 [1]). These processors have gained space in solutions to specific purpose problems by using modules that can be configured at synthesis time.

At the same time, nowadays' systems require high performance for a wide range of applications, increasing the demand for logic resources. Modern Multiprocessor System on Chip (MPSoCs) [2] usually comprise multi-core general purpose processors and dedicated hardware like Floating-Point Units (FPUs), security and cryptography modules, and coders/decoders for multimedia [3]. However, FPGA designs require more area and energy compared to Application Specific Integrated Circuit (ASICs) [4]. Therefore, the resources available in an FPGA may become a limiting factor to implement all the needed features. When specialized hardware cannot fit inside the FPGA, some of its functionalities must be mapped into the software domain, which is significantly slower.

In some cases, the Block Random Access Memories (BRAMs) present in these FPGAs may be underutilized when implementing such complex logic driven designs. Table 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, in their default configuration [5–7]. 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

* Corresponding author.
E-mail addresses: phebecker@inf.ufrgs.br (P.H.E. Becker), alsartor@inf.ufrgs.br (A.L. Sartor), mbrandalero@inf.ufrgs.br (M. Brandalero), caco@inf.ufrgs.br (A.C. Schneider Beck).

https://doi.org/10.1016/j.micpro.2018.09.007
0141-9331/

Table 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%

the register file in multiple-issue processors, which need multiple read ports to properly feed all the available functional units [8]. Hence, BRAMs are usually used only to implement moderate-sized caches, common in the scope of soft-cores running in embedded environments. In cases when more BRAMs are needed, such as in multi-core architectures where a large amount of shared memory is a must, BRAM occupation raises. On the other hand, the addition of multiple application-specific modules and the use of more robust General Purpose Processors (GPPs) designs (e.g.: Multiple-issue) put the pressure back on LUTs requirements.

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 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.

Our reuse concept can also be used within a soft-core multiprocessor 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 multithread programs (or multiples instances of a program) run over the same code, they benefit from such arrangement, increasing the reuse possibilities. At the same time, the introduction of a shared RT uses even less FPGA logic resources 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 from precise to approximate reuse using the same hardware structure. This 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 our technique by implementing it in a complex 4-issue Very Long Instruction Word (VLIW) soft-core described in Hardware Description Language (HDL). First, we investigate six single-core applications that process a significant amount of FP operations in different scenarios, including one where implementing a hardware FPU would prevent the addition of any new dedicated hardware because of the limited amount of resources available. In this case, we apply the technique 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. We show that an average speedup of 1.39× is achieved when considering an RT that fits in five different test targets. The average can be as high as 1.87× for targets with larger BRAMs.

Then, we consider a multicore environment, where multiple instances of each of the benchmarks execute on different cores with distinct inputs. We consider two different scenarios w.r.t. the RT: one individual RT per core; and one shared RT, where all cores access it when reuse is needed. We demonstrate that cores sharing the RT can reach performance levels as the former case (a dedicated RT per core), but demanding less FPGA BRAM, which can be particularly important if the target device has only a few BRAM blocks. We also present a case where a shared RT even improves performance of a particular instance of the benchmark executing on a core with its own private RT, from 1.25× to 1.9×.

Finally, for the approximate case, we evaluate an image processing filter software that tolerates a certain error level. We apply our technique over a set of 30 images, achieving an average speedup of 1.52× with less than 10% error (standard tolerable error rate for approximate image processing [9]), and reaching up to 2.97× when output quality is slightly more relaxed (around 17% error rate).

Since our approach needs a simple mechanism to work, the usage of slice registers and slice LUTs increases by 16.56% and 3.43% respectively in the single-core system, compared to 140% and 48% for an FPU or 11% and 13% for a dedicated image filter. When considering a multi-core system with a shared RT, we show that the use of slice registers can be amortized in 17.66% and slice LUTs in 3.06% in a quad-core system, as a result of logic replication avoidance.

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

2. Related work

This section presents related work on reuse for both single-core (Section 2.1) and multi-core (Section 2.2) environments, followed by works that exploit approximate reuse (Section 2.3).

2.1. Reuse in single-Core environments

A variety of works has discussed computation reuse [10]. Implementations vary from software (where reuse is also known as *memoization* [11]) to hardware-based solutions and cover different granularities of code. In [12], dynamic instruction reuse is presented with execution-driven simulation. The goal is to avoid re-execution of instructions in an out-of-order processor. Instructionsâ;; source registers are the inputs, and its results are the output. The scheme is enhanced with control of dependency links among instructions, providing reuse of a set of dependent instructions. Authors in [13] 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 speedup 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 in [14] and simulated using SimpleScalar [15]. 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) in [16]. In this case, less reusability is found compared to instruction reuse, but more speedup is obtained since larger chunks of code are involved.

The authors in [17] introduced the concept of dynamic function result reuse. In this case, only *pure* functions (global variables free,