Custom Instruction Integration Method within Reconfigurable SoC and FPGA Devices

Yassine Aoudni
CES Lab, ENIS National Engineering School of Sfax, Soukra street 3032 Sfax, Tunisia
LESTER Lab, research centre BP 92116 Lorient 56321, France
Email: yassine.aoudni@univ-ubs.fr

Guy Gogniat
LESTER Lab, research centre BP 92116 Lorient 56321, France
Email:guy.gogniat@univ-ubs.fr

Mohamed Abid
CES Lab, ENIS National Engineering School of Sfax, Soukra street 3032 Sfax, Tunisia
Email:mohamed.abid@enis.rnu.tn

Jean-Luc Philippe
LESTER Lab, research centre BP 92116 Lorient 56321, France
Email:jean-luc.philippe@univ-ubs.fr

Abstract— **General-purpose processors that are utilized as cores are often incapable of achieving the challenging cost, performance, and power demands of high-performance audio, video, and networking applications. To meet these demands, most systems employ a number of hardware accelerators to off-load the computationally demanding portions of the application. As an alternative to this strategy, we examine customizing the computation capabilities of a core processor for a particular application. Our goal is to generate a prototype of reconfigurable custom instruction SoC to answer application request using FPGA technology. To give more flexibility to system, we addressed customized core with coarse and finite granularity. In this paper, we provide an overview of a method to identify coarse and finite grain instruction set extensions in application code and integration process in reconfigurable SoC based on NIOSII processor core. 3D synthesis application was proposed as a case study for experimentation.**

I. Introduction

A common method for providing performance improvement for an embedded computer system is to create customized hardware solutions for particular tasks. For example, embedded computer system often has one or more application specific integrated circuits (ASICs) to perform computationally demanding tasks. ASICs are very effective at improving performance, typically yielding several orders of magnitude speedup along with reduced energy consumption. Unfortunately, there are also negative aspects to using ASICs. The primary problem is that ASICs only provide a dedicated hardwired architecture solution, meaning that only a limited number of applications will be able to fully explore the ASIC architecture. If an application changes, because of a fixed bug or a change in standards, the system will usually no longer be able to take advantage of the ASIC device architecture. So, reconfigurable feature was introduced in embedded system concept as a need to solve bugs and to support the evolution of standards. Another drawback is that even when a system can utilize an ASIC, it must be specially rewritten to do so. Rewriting system applications or few tasks of applications can be a large engineering burden. For this reason, the use of reusable components libraries is encouraged to accelerate the design process and to conserve the compatibility for the evolution in system on chip (SoC) design.

Adding custom hardware in processor core is another method for providing enhanced performance in SoC. In general, the critical portions of an application's data flow graph (DFG) can be accelerated by mapping them to a custom hardware. Usually, there are two granularity levels to add dedicated hardware to processor core system: instruction granularity level and function granularity level. The instruction granularity consists in linking custom hardware with the main registers of processor core and a custom instruction opcode is added to the processor instruction set. The number of custom instructions depends on the processor core capacity, for example ARM core provides 16 custom instruction extensions. The function granularity consists in adding the custom hardware as a slave or a master peripheral using bus communication. In this case one instruction extension can not drive the functionality between the processor and the customized peripheral. So in many cases, specific subroutines should be coded to control the custom hardware activity and the communication with the processor core. The number of added hardware functions depends on the bus bandwidth and the device size in the case of FPGA circuits. In the case of instruction granularity the processor is in hold mode and it is blocked in custom instruction execution, but in function granularity the mutual execution of processor core and custom peripheral is possible.

In this paper we are interested to custom instruction integration in reconfigurable system on chip. We propose a method to identify the parts of application code that should be executed as custom instructions. Then, we present the integration process of custom instructions within NIOSII processor core. 3D frame synthesis application is used as an experimentation case study. The paper is organized as follows. In Section 1 we discuss the related work in custom instruction design. Section 2 presents custom instruction identification method. Section 3 talks about the integration process within NIOSII system. Section 4 presents 3D frame synthesis application case study and results of custom instruction integration. Finally, we close with conclusions.

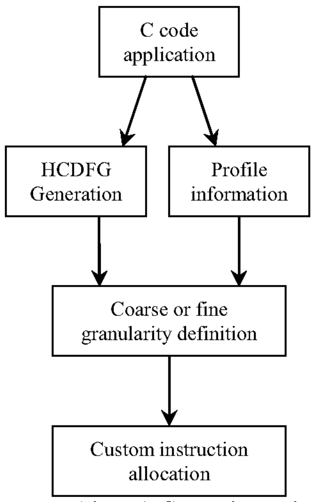
II. Related work

Partitioning an embedded application into hardware and software parts has been studied for years. The software parts are usually run on an embedded processor, while the hardware parts are implemented as co-processors. Such partitions are at the function or application level, which provides a coarse-grained speedup, compared to pure software implementations. Wolf surveyed the development of hardware- software co-design in the past decade and concluded that co-design is becoming a mainstream technology [3]. In [4], De Micheli et al. also surveyed the research developments in hardware-software co-design since its emergence in the early 90s. The recent development of behavioral synthesis (C-based) tools makes automatic hardware-software partitioning from high-level languages (e.g., C/C++) possible. Configurable and extensible processors are also gaining popularity. A lot of research has focused on automatically extending an instruction set with custom instructions to speed up embedded applications. Arnold et al. presented a semi-automated method to extend the domain-specific instruction set for embedded processors [5]. The possible instructions are restricted by the number of inputs/outputs. Atasu et al. introduced an algorithm to form custom instructions by selecting maximal speedup convex dataflow graphs [6]. Cheung et al. proposed techniques for custom instruction selection, and mapping complex application code to pre-designed custom instructions [7]. In [8], Sun et al. proposed an automatic method to generate custom instructions using operation patterns, and select the instructions under an area budget. Goodwin et al. proposed techniques to generate custom instructions by identifying VLIW-style operations, vectorized operations, and fused operations in [9]. These works make automatic custom instruction generation possible, providing fine grained hardware-software partitioning. Fie Sun et al. proposed a methodology to synthesize custom instructions and co-processors simultaneously based on Xtensa platform from Tensilica [17]

In our work, we focus on prototyping custom instructions within reconfigurable SoC using FPGA technology, thus several prototypes can be proposed as solutions given different performance modes in term of execution time, power consumption and resources usage. We propose a method to identify custom instructions in application code then we present the process steps for custom instruction integration within NIOSII system. A 3D synthesis application was adopted as an experimentation case study.

III. Custom instruction identification method

In the next section, we aim to explain the main idea of custom instruction identification method for coarse and fine granularity. As described in Figure 1, the identification method is based on HCDFG (Hierarchical Control Data Flow Graph) generation and profiling information. Indeed, starting from C code description of the application, we use a parser to scan the C code and generate the HCDFG of application. Thus, we have the possibility to identify the dependencies between branches in HCDFG and the number of arithmetic and logic operations used in the application (Figure 2).


C code application
HCDFG Generation
Profile information
Coarse or fine granularity definition
Custom instruction allocation
Figure 1: Custom instruction allocation

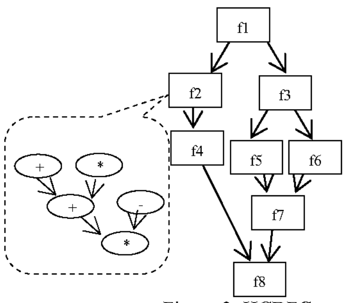


Figure 2: HCDFG example

On the other hand, we need profiling information to know the call number of each function within the application. Then, data given by HCDFG description and profiling are collected to identify the portion of the application that should be accelerated by hardware implementation. Several functions of the application can be selected for acceleration. In this step, we have two implementation versions of each selected function:

1- Using fine grain operations: the selected branch of HCDFG is described with basic arithmetic and logic operations. So, in order to accelerate this function, we propose to implement arithmetic and logic operations as a custom instruction within a processor core. In this case, all selected functions can be executed using the same customized operator.
2- Using coarse grain operations: if the selected function

```
{…
 int b=4;
 int a=12;
 int result;

//initial code
 result = a+b;

…
}
```

```
{…
 int b=4;
 int a=12;
 int result;

//using custom instruction opcode
 result = nm_addi(a,b);

…
}
```

exists in library as a hardware module, then the full function can be added as a custom instruction to the processor core. Or we can describe a specific hardware module for a selected function. In this case, each function is executed using the appropriate customized module and operators can not be shared between selected functions.

IV. Integrated custom instruction within NIOSII processor core

After the custom instruction identification, we aim to prototype the application with accelerated functions using custom instructions. So, we propose to use the NIOSII prototyping platform [18]. Indeed, NIOSII is a soft core that offers the possibility to integrate 5 custom instructions using three main registers: (dataa[0..31] , datab[0..31]) as inputs and (result[0..31]) as output.

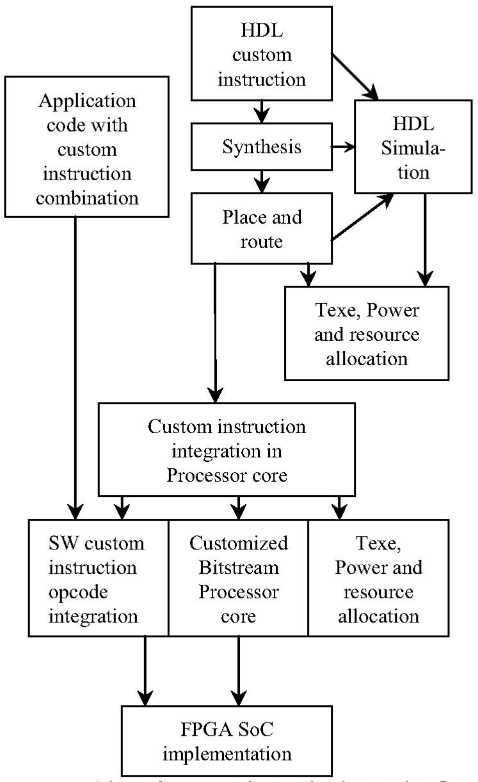

HDL custom instruction
Application code with custom instruction combination
Synthesis
HDL Simula-tion
Place and route
Texe, Power and resource allocation
Custom instruction integration in Processor core
SW custom instruction opcode integration
Customized Bitstream Processor core
Texe, Power and resource allocation
FPGA SoC implementation

Figure 3 : custom instruction integration flow

After the identification of custom instruction in application code and as described in Figure 3, three steps are needed to successfully integrate a custom instruction within NIOSII processor core. Firstly, a HDL description of custom instruction must be designed and verified using VHDL or Verilog and Quartus environment. This step will help the designer to test and verify the functionality of the custom instruction before integration in NIOSII core and to get performance information about the hardware module (resource allocation, execution time and power dissipation).

Figure 4 : custom instruction opcode insertion

The second step consists in updating the initial application code with the custom instruction opcode. For example, in Figure 4 "*nm_addi*" was used as a custom instruction opcode for addition operation of integer operands. In third step, the integration of custom instruction in NIOSII core consists in generating a specific coprocessor interface to adapt the communication between the custom hardware module and the ALU of NIOSII core. The coprocessor interface must respect:

- data size and nomination for inputs and outputs
- control signals for sequential and combinatory hardware module

After these three steps, SOPC Builder and Quartus environment can be used to integrate custom instruction opcode in NIOSII compiler and to generate the customized NIOSII bitstream. In addition, performance information can be collected to specify the custom prototype by resources usage, execution time and power dissipation.
In the next section, we propose some experimentation results to detail the custom instruction integration with NIOSII processor core.

V. 3D synthesis case study

We choose as case study the 3D graphic pipeline. The main function of the pipeline is to render, a two-dimensional image given a virtual camera, 3D objects, light sources, lighting models, and textures. The typical 3D graphics system can be divided in three stages in pipelined format as in Figure 5. In our case, we are interested to study the Geometric engine.

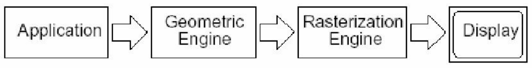


Figure 5 : 3D graphic pipeline

We proposed to implement the 3D application using a C language. The profiling process identifies in the global application code 6 functions: Scalaire, Vectoriel, Mult_matrice, Projection, Transformation and Znormal. In table 1, we present a summary of profiling results with three different objects.

We mention that the called number of each function depends on the object size. These six functions were scanned by the parser to generate the HCDFG of the application. As deduction form HCDFG, table 2 gives statistic information of basic arithmetic operations in each function. From table 2, it is clearly that hardware implementation of basic arithmetic operation will speed up the application. Also, we need to know if it is more benefit or not to use custom hardware of the six selected function in our case study.

Function name	Called number		
	Object1	Object2	Object3
Scalaire	1120	2260	3120
Vectoriel	2380	5280	7280
Mult_matrice	50	150	250

Projection	1210	2260	3660
Transformation	1210	2260	3660
Znormal	2380	5280	7280

Table 1 : profiler summary results

	Add	Sub	Mul	Div
Scalaire	3	0	3	0
Vectoriel	6	3	6	0
Mult_matrice	14	0	16	0
Projection	6	3	8	3
Transformation	8	4	8	2
Znormal	10	6	12	6
Total	47	16	53	11

Table 2: basic arithmetic operations statistics

Based on information given by table1 and table2, we have the possibility to customize 3D synthesis application in two granularity levels:

1- fine granularity: using basic arithmetic operations as a custom hardware
2- coarse granularity: using hardware module of each of the six identified functions.

Two versions of the 3D synthesis application code were implemented within NIOSII processor core and StratixII FPGA device. The performance results are given by table 3.

	Execution time (µs)	Power dissipation (mw)	Resources usage (StratixII FPGA)
Fine grain version	21189	562,87	33 %
Coarse grain version	12180	924,36mW	45%
Ratio	74%	-64%	-36%

Table 3: Performance results of 3D synthesis application

We remark that the coarse grain version provides a speedup to 3D synthesis application but the SoC will lose in low power dissipation and resources usage. As a conclusion, we give two custom configuration prototypes for 3D synthesis application with different functional mode for a StratixII FPGA implementation.

VI. Conclusion

In this paper, we proposed a method for custom instruction identification and integration in SoC design using reconfigurable processor core and FPGA technology. Our identification method is based on HCDFG description and profiling information. The integration process of custom instruction in processor core was done using coarse and fine granularity. For experimentation, we proposed to use NIOSII processor core and StratixII FPGA device to prototype the customized reconfigurable SoC. 3D synthesis application was chosen as case study to validate the identification method and the integration process.

References

[1] K. Wakabayashi, "C-based synthesis experiences with a behavior synthesizer, "Cyber"," in Proc. Design Automation & Test Europe Conf., Mar. 1999, pp. 390–393.

[2] Catapult C Synthesis. Mentor Graphics http://www.mentor.com

[3] W. Wolf, "A decade of hardware/software codesign," Computer, vol. 36, no. 4, pp. 38–43, Apr. 2003.

[4] G. De Micheli, R. Ernst, and W. Wolf, Readings in Hardware/Software Co-design. Morgan Kaufmann Publishers Inc., San Francisco, CA, 2001.

[5] M. Arnold and H. Corporaal, "Designing domain-specific processors," in Proc. Int. Symp. HW/SW Codesign, Apr. 2001, pp. 61–66.

[6] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in Proc. Design Automation Conf., June 2003, pp. 256–261.

[7] N. Cheung, S. Parameswaran, J. Henkel, and J. Chan, "MINCE: Matching instructions using combinational equivalence for extensible processors," in Proc. Design Automation & Test Europe Conf., Feb. 2004, pp. 1020–1027.

[8] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "A scalable application-specific processor synthesis methodology," in Proc. Int. Conf. Computer- Aided Design, Nov. 2003, pp. 283–290.

[9] D. Goodwin and D. Petkov, "Automatic generation of application specific processors," in Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems, Oct. 2003, pp. 137–147.

[10] M. Girkar and C. D. Polychronopoulos, "Automatic extraction of functional parallelism from ordinary programs," IEEE Trans. Parallel & Distrib. Systems, vol. 3, no. 2, pp. 166–178, Mar. 1992.

[11] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," IEEE Trans. Evolutionary Computation, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[12] XtensaTM microprocessor. Tensilica Inc. (http://www.tensilica.com).

[13] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in Proc. Int. Symp. Microarchitecture, Dec. 1997, pp. 330–337.

[14] Design Compiler. Synopsys Inc. (http://www.synopsys.com).

[15] CB-11 Cell Based IC Product Family. NEC Electronics, Inc. (http://www.necel.com).

[16] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: Task graphs for free," in Proc. Int. Symp. HW/SW Codesign, Mar. 1998, pp. 97–101.

[17] Fei Sun, Srivaths Ravi , Anand Raghunathan, and Niraj K. Jha, "Hybrid Custom Instruction and Co-processor Synthesis Methodology for Extensible Processors", Proceedings of the 19th International Conference on VLSI Design, VLSID 2006.

[18] NiosII development kit user guide, Altera web site, www.altera.com