A FRAMEWORK FOR ASIP AND MULTIPROCESSOR ARCHITECTURES INTEGRATION WITHIN RECONFIGURABLE SOC AND FPGA DEVICES

Yassine Aoudni† and Mohamed Abid

CES Lab, Electrical Department, National Engineering School of Sfax, Soukra Street 3032 Sfax Tunisia

and Guy Gogniat, Jean-Luc Philippe

LESTER Lab, Centre of Research BP 92116 Lorient 56321, France

الخلاصة:

لقد اكتسبت المعالجات المختصة في تطبيقات محددة (ASIPs) شعبية في مجال إنتاج الرقائق كما هو الشأن في الأوساط البحثية. فهي (ASIPs) توفر حلاً ناجعاً موفقاً بين الكفاءة والمرونة بالنسبة للأنظمة المدمجة (SoC). وبعامة فإن المعالج المختص في تطبيقات محددة يوفر القدرة على توسيع مجموعة التعليمات الأساسية لمعالج ذي أغراض عامة بمدّه بمجموعة من التعليمات بحسب الطلب التي تكون مدعومة بموارد محددة متوفرة على المعالج المختص. والموارد المادية التي تتضمن التعليمات الخاصة يمكن أن تكون من فئة الوحدات الوظيفيه القابلة لإعادة التهيئة وهي قيد التشغيل أو من تلك الرقائق سابقة التأليف. وفي هذه المقالة نقترح إطاراً لتطبيق النماذج داخل أجهزة FPGAs باستخدام ASIPs وتصميم متعدّد المعالجات. في مرحلة أولى، ونقدم لمحة عامة عن طريقة لتحديد التعليمات الخشنة والتعليمات المحددة في التمديدات لبرنامج التطبيقات وعملية إدماج الـASIP. في مرحلة ثانية، وقد قسمّنا البيانات المدخلة للحصول على المعطيات المتزامنة ذات التبعية بالنسبة للتطبيق المستهدف كي نتمكن من إنجاز العمليات على تصميم متعدّد المعالجات. وفي مرحلة أخيرة، نقدّم مقارنة بين مستويات الأداء التي قدمها التطبيق المستهدف باستعمال تصميم من فئة الـ ASIP و آخر متعدد المعالجات.

† To whom correspondence should be addressed:
Email: yassine.aoudni@univ-ubs.fr
Fax : (33/0) 2-97-87-45-27

Paper Received: 16 April 2007; Paper Rewritten: 29 October 2007; Paper Accepted: 28 November 2007

ABSTRACT

Application-Specific Instruction-set Processors (ASIPs) have gained popularity in production chips as well as in the research community. They offer a viable solution to the tradeoff between efficiency and flexibility for the embedded System-on-a-Chip (SoC). Generally, an ASIP has the capability to extend the base instruction set of a general-purpose processor with a set of customized instructions supported by the specific hardware resources provided on the ASIP. The hardware implementing the specific instructions can be either runtime reconfigurable functional units, or pre-synthesized circuits. In this paper, we propose a framework for prototype application within FPGAs devices using ASIPs and multiprocessors architecture. Firstly, we provide an overview of a method to identify coarse and finite grain instruction set extensions in application code and integration process of ASIP. Second, we split the data input application to get concurrence and the data dependency of the target application in order to process to multiprocessor execution. Finally, we compare the performances levels given by the execution of the application using ASIP architecture and multiprocessor architecture.

***Key words*: ASIP, FPGA, system on chip**

Yassine Aoudni, Mohamed Abid, Guy Gogniat, and Jean-Luc Philippe

A FRAMEWORK FOR ASIP AND MULTIPROCESSOR ARCHITECTURES INTEGRATION WITHIN RECONFIGURABLE SOC AND FPGA DEVICES

1. INTRODUCTION

Embedded systems have been widely used in various fields in today's world. However, designing a modern embedded system in nanometer technologies is more difficult than ever, and the problems continue to worsen with shrinking feature sizes. Due to the complexity and electrical design challenges posed by each new technology generation, the design productivity gap continues to grow larger despite the increasingly expensive CAD tools. This requires a move toward the use of programmable and reconfigurable solutions to allow more flexibility for accommodating specification changes and avoiding potential design errors. Application-Specific Instruction-set Processors (ASIPs) have gained popularity in production chips as well as in the research community. They offer a viable solution to the tradeoff between efficiency and flexibility for the embedded System-on-a-Chip (SoC). Generally, an ASIP has the capability to extend the base instruction set of a general-purpose processor with a set of customized instructions supported by the specific hardware resources provided on the ASIP. The hardware implementing the specific instructions can be either runtime reconfigurable functional units, or pre-synthesized circuits.

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 at which 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. An other solution consists of parallel execution mode with multiprocessor core. In this case, we need to analyze the concurrence and the data dependence between the application tasks.

In this paper we are interested to custom architecture integration in reconfigurable system on chip. We propose a method to identify the parts of application code that should customize. Then, we present the integration process of custom parts within FPGA devices based on general purpose processor core. Indeed, the proposed integration process offers the opportunity to generate custom architectures of execution mode with:

- ASIP architecture
- Multiprocessor architecture

In the experimentation part, we target MPEG2 decoder and 3D graphic to show the efficiency of the integration method and to compare the performances given by ASIP and multiprocessor architectures.

The paper is organized as follows. In Section 1 we discuss the related work in custom architecture design. Section 2 presents identification method of custom parts. Section 3 talks about the integration process within FPGA devices based on NIOSII processor core and the multiprocessor integration. Section 4 presents the experimentations results. Finally, we close with conclusions.

2. 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 architectures within reconfigurable SoC using FPGA technology, thus several prototypes of custom instruction and multiprocessor architectures 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 parts in application code then we present the process steps of integration for ASIP and Multiprocessor architectures.

3. IDENTIFICATION METHOD OF CUSTOM PARTS

In the next section, we aim to explain the main idea of identification method of custom part for a target application. 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 of identifying the dependencies between branches in HCDFG and the number of arithmetic and logic operations used in the application (see Figure 2).

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 three implementation versions of the application:

- Full software execution mode: all functions of the application were developed using C language. In this step we use the profiler in order to show the software execution time of each function, and then we can calculate the speed up of the hardware execution. We suppose that we have a C compiler processor core compiler.
- Using custom instruction: The selected function can be represented by at least one branch of HCDFG. Then each branch is implemented as a hardware component using basic arithmetic and logic operations. The communication between the custom hardware and the processor core was provided by the main registers of ALU (Arithmetic and Logic Unit). As a consequence, the hardware component will be instantiated into coprocessor interface top level entity in order to guaranty the compatibility between the custom hardware and the ALU.
- Using multiprocessor architecture: concurrence and data dependency between functions offers the opportunity to partition the application on multiprocessor architecture to benefit of parallel execution mode. The number of processor cores depends on the application parallelism degree and area constraint. We propose to use hardware operating system services like mailbox, semaphore, and message queue for communication between processor cores.

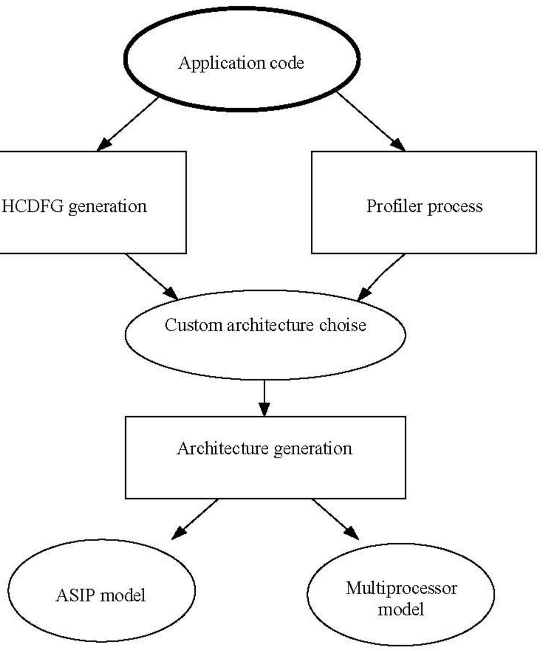

Figure 1. Identification method flow

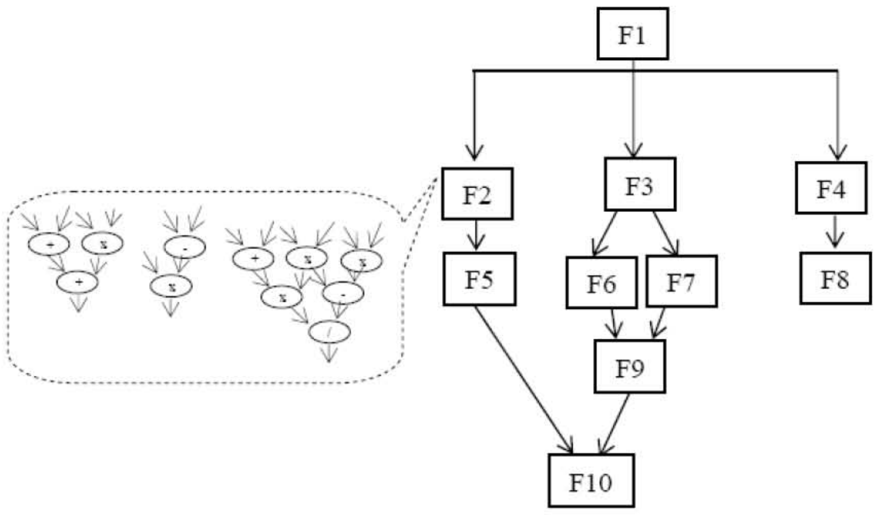

Figure 2. HCDFG example

4. INTEGRATION OF CUSTOM ARCHITECTURE WITHIN FPGA DEVICES

4.1. ASIP Integration Using NIOSII Processor

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.

As described in Figure 3, three steps are needed to successfully integrate a custom instruction within the 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 the NIOSII core and to get performance information about the hardware module (resource allocation, execution time and power dissipation).

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.

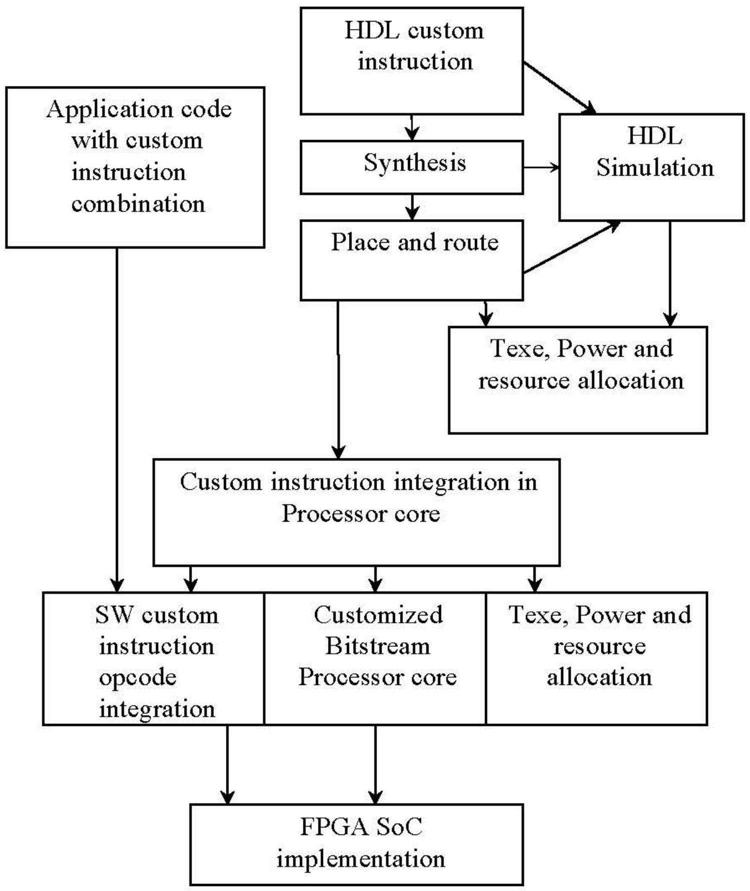
HDL custom instruction
Application code with custom instruction combination
Synthesis
HDL Simulation
Place and route
Texe, Power and resource allocation
Custom instruction integration in Processor core
SW custom instruction opcode integration
Customized Bitstream Processor core
Texe, Power and resource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

Figure 3. Custom instruction integration flow

```
{…
 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);
…
}
```

Figure 4. Custom instruction opcode insertion

4.2. Multiprocessor Integration

In our work we propose to use FPGA features (high capacities, embedded block RAMs) and multi master Avalon bus in order to define the multiprocessor architecture. In addition, in order to manage the communication and shared data between processors we propose to use hardware operating system services like mailbox and mutex. Figure 5 gives an example of multiprocessor architecture that uses a mutex hardware module to manage shared memories.

A mutex allows cooperating processors to agree that one of them should be allowed mutually exclusive access to a hardware resource in the system. This is useful for the purpose of protecting resources from data corruption that can occur if more than one processor attempts to use the resource at the same time. The mutex core acts as a shared resource, providing an atomic "test and set" operation in which a processor may test if the mutex is available and if so, acquire it in a single operation. When the processor is finished using the shared resource associated with the mutex, the processor releases the mutex. At this point, another processor may acquire the mutex and use the shared resource. Without the mutex, this kind of function would normally require two separate "test" and "set" instructions between which, another processor could also test for availability and succeed. This situation would leave two processors both thinking they successfully acquired mutually exclusive access to the shared resource when clearly they did not.

The mailbox component contains two mutexes: One to ensure unique write access to shared memory and one to ensure unique read access from shared memory. The mailbox core is used in conjunction with a separate memory in the system that is shared among multiple processors.

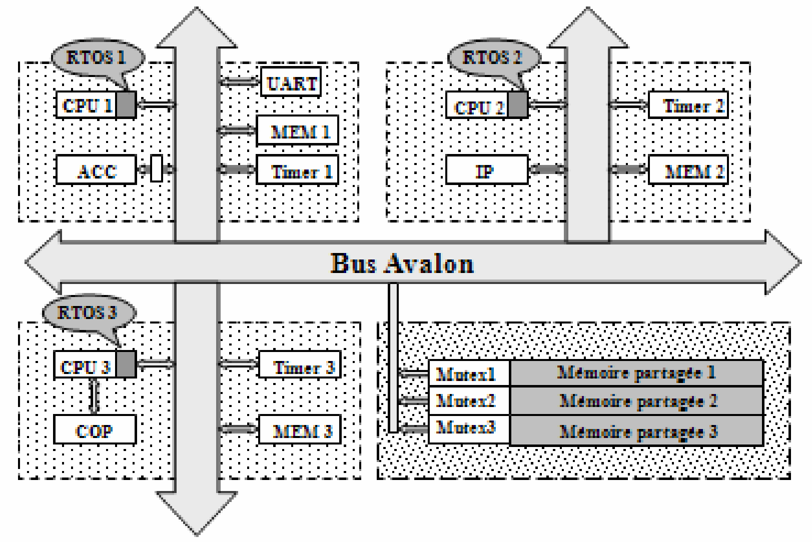

Figure 5. Example of multiprocessor architecture

In the next section, we propose some experimentation results to detail the custom instruction and multiprocessor integration with NIOSII processor core.

5. EXPERIMENTATION

5.1 ASIP Execution Mode

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.

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

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 clear that hardware implementation of basic arithmetic operation will speed up the application. Also, we need to know if it is of more benefit or not to use custom hardware of the six selected functions in our case study.

Table 1. Profiler Summary Results

Function name	Called number		
	Frame1	Frame2	Frame3
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 2. Basic Arithmetic Operations Statistics

	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

Based on information given by Tables 1 and 2, we have the possibility to customize 3D synthesis application in two granularity levels:

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

Table 3. Performance Results of 3D Synthesis Application on FPGAs Devices

	Execution time (µs)	Power dissipation (mw)	Resources usage
STRATIX II device			
ASIP Fine grain version	21189	562,87	33 %
ASIP Coarse grain version	12180	924,36mW	45%
Ratio	74.36%	-64.2%	-36.35%
CYCLONE II device			
ASIP Fine grain version	17250	163,5 mW	47 %
ASIP Coarse grain version	10375	225,7mW	73%
Ratio	66.26%	-38.04%	-55.33%

Two versions of the 3D synthesis application code were implemented within NIOSII processor core within STRATIX II and CYCLONE II FPGA devices. The performance results are given in Table 3.

We remark that the coarse grain version provides a speedup to the 3D synthesis application but the SoC will lose in low power dissipation and resources usage. Also, the CYCLONE II technology is more efficient than STRATIX II in power dissipation, but the impact of the coarse grain version is more important for STRATIX II devices.

5.2. Multiprocessor Execution Mode

The multiprocessor execution mode directs all its processors to execute the same instruction at the same cycle but with different data: SIMD machines. In our case, frames are input data of the application. So, we start by decompose each frame to blocks to get different data for SIMD machine. Then each processor will execute the 6 functions (same instructions) on the associated blocks. The neighbor's blocks have shared pixels. So we define a mailbox between neighbor processors in order to communicate the shared data. For example, if we decompose the frame into 4 blocks *(B0,B1,B2,B3)* and we propose to use 4 processor cores (P0,P1,P2,P3). So we deduce the following association group : *{(B0,P0), (B1,P1), (B2,P2) (B3,P3)}* and the neighbor group: *{(P0,P1), (P0,P2), (P1,P3),(P2,P3)}*. Thus, for each neighbor pair, we instantiate a hardware mailbox communication module on AVALON multi master bus.

Table 4 shows the experimental results of multiprocessor execution mode using different numbers of processor cores. We can deduce that 4 processor solution is more efficient for FPGA device implementation because it gives low power dissipation and it takes a good portion of resources. Also the multiprocessor solutions are in the same case 2.5× more efficient than custom instruction solution in term of execution time and 3.6× in term of power dissipation. Indeed, in Tables 5 and 6, we show the performance speed up and power gain of multiprocessor architecture by calculating the ratio between ASIP measurements and multiprocessor ones.

Table 4. Multiprocessor Execution Mode

	Execution time (µs)	Power dissipation (mw)	Resources usage
STRATIX II device			
2 CPU	11340	362.87 mW	30 %
4 CPU	9240	254.36mW	65%
CYCLONE II device			
2 CPU	10360	97.5 mW	25 %
4 CPU	7012	76.7mW	63%

Table 5. Speed up Between ASIP and Multiprocessor Modes

	ASIP Fine grain version	ASIP Coarse grain version
STRATIX II device		
2 CPU	1.86×	1.07×
4 CPU	2.29×	1.31×
CYCLONE II device		
2 CPU	1.66×	1.001×
4 CPU	2.46×	1.47×

Table 6. Power Gain Between ASIP and Multiprocessor Modes

	ASIP Fine grain version	ASIP Coarse grain version
STRATIX II device		
2 CPU	1.55×	2.54×
4 CPU	2.2×	3.6×
CYCLONE II device		
2 CPU	1.67×	2.31×
4 CPU	2.13×	2.9×

6. CONCLUSIONS

In this paper, we propose a method for custom instruction identification and integration in SoC design using ASIP and multiprocessor architectures within reconfigurable processor core and FPGA technology. Our identification method is based on HCDFG description and profiling information. The integration process of custom instruction in a processor core was done using coarse and fine granularity. The multiprocessing execution is done with the data input split into independent data block, and the communication between processor units was implemented using hardware mailbox and Avalon multimaster communication prototcols. For experimentation, we propose to use NIOSII processor core on STRATIX II and CYCLONE II FPGA devices to prototype ASIPs and multiprocessor SoC. 3D synthesis application was chosen as a case study to validate the identification method and the integration process for ASIPs and multiprocessor system. The results of several implementations show that the multiprocessor architectures can be 2.5× more efficient than ASIPs in execution time and 3.6× in power dissipation.

REFERENCES

[1] K. Wakabayashi, “C-Based Synthesis Experiences with a Behavior Synthesizer, ‘Cyber’,” in *Proc. Design Automation & Test Europe Conf.*, March 1999, pp. 390–393.

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

[3] W. Wolf, “A Decade of Hardware/Software Codesign,” Computer, **36(4)**(2003), pp. 38–43.

[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*, April 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.*, February 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*, November 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*, October 2003, pp. 137–147.

[10] M. Girkar and C. D. Polychronopoulos, “Automatic Extraction of Functional Parallelism from Ordinary Programs,” *IEEE Trans. Parallel & Distrib. Systems*, **3(2)** 1992, pp. 166–178.

[11] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II,” *IEEE Trans. Evolutionary Computation*, **6(2)**(2002), pp. 182–197.

[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*, December 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*, March 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”, *Proc. of the 19th International Conference on VLSI Design*, (VLSID 2006).

[18] NiosII Development Kit User Guide, Altera web site, www.altera.com.