A Processor Optimization Framework for a Selected Application

Jakub Podivinsky, Ondrej Cekan, Martin Krcma, Radek Burget, Tomas Hruska, Zdenek Kotasek
Brno University of Technology, Faculty of Information Technology,
Centre of Excellence IT4Innovations
Bozetechova 2, 612 66 Brno, Czech Republic
Tel.: +420 54114-{1361, 1361, 1360, 1320, 1239, 1223}
Email: {ipodivinsky, icekan, ikrcma, burgetr, hruska, kotasek}@fit.vutbr.cz

Abstract

A processor plays the main role in almost every electronic system. The use of a general purpose processor may not be suitable for a specific application, because the processor is designed for a wide set of applications. The Application-Specific Instruction-set Processors (ASIPs) are today applied in specific cases, where a single application or a certain group of applications is performed. This paper focuses on automatic optimization of an ASIP for a given application through checking the possible configurations of its key parameters (such as the number of registers, cache sizes, instruction set modifications, etc.). The paper also presents a designed framework which is able to optimize the given application in terms of speed, area or power consumption. The framework allows to use various optimization methods. For the processor modeling and evaluation, the Codasip Studio tool is used. It allows to generate all the tools necessary for compilation, simulation, and hardware mapping which are used in the process of the ASIP design. The experiments are carried out on a RISC-V (Reduced Instruction Set Computing) processor.

1. Introduction

In today's world of the Internet of Things (IoT), the vast majority of all systems is controlled by processors. The processors differ from application to application. General purpose processors (GPPs) are typically used in multiple different applications due to their versatile design. GPPs have an acceptable computation capacity which is paid by larger chip area and power consumption. A significant advantage is their price, which is low due to the large scale production. For embedded systems, on the other hand, small size and low power consumption processors are required, which have sufficient performance for the particular task. Such processors are designed in a completely different way, and their price is significantly higher due to their application for a limited group of tasks.

Recently, the main attention has been paid to the use of GPPs in embedded systems which are optimized for a given application domain. These processors are referred to as the Application Specific Instruction-set Processors (ASIPs) [10]. Their advantage is primarily in their price which is similar to GPPs and the possibility of their optimization in terms of different metrics (speed, area, power consumption) for the given application. The optimization of a processor involves reducing its key parameters and features such as the number of registers, the number of slots for Very Long Instruction Word (VLIW) processors, functional units configuration, cache configuration, or instruction set modification – removing unnecessary instructions to reduce the chip area or adding special-purpose instructions to accelerate computations. The complete settings of the individual processor parameters for the given application are referred to as the processor configuration.

The processor can be modeled using architecture description languages (ADLs) or by hardware description languages (HDLs) [15]. ADLs provide a more abstract way of the processor description (i.e. the designer does not have to pay much attention to hardware details) which is more suitable for fast processor prototyping. There exist various tools for automatic processor generation based on its abstract description. As an example, we can mention several of them. The *Synopsys ASIP Designer* [20] is a set of tools for ASIP design from a user-defined architecture to RTL description. Synopsys also offers highly configurable processor cores

978-1-5386-5710-2/18/$31.00 ©2018 IEEE

called ARC and the Synopsys ARC Designer [21]. The Cadence company provides a high-performance, configurable and extensible processor called Xtensa LX7 Processor and its development tools [5]. In our experimental work, we use the Codasip Studio provided by the Codasip company [6]. Codasip Studio is a development tool for processor design; the designer is able to describe the architecture of a processor and its instruction set and then, to generate a corresponding toolchain (compiler, simulator, etc.) Codasip also offers predefined configurable processor cores (eg. RISC-V based Codix-Bk processor [7]). It is possible to generate various processor configurations and test their usability for a selected application.

The search for the most appropriate processor configuration is currently mostly performed manually based on designer's knowledge and experience. This activity is time-consuming and does not guarantee to find the best configuration of the processor in a large set of possible configurations. With the use of Codasip Studio, our goal of research is automatic optimization of processor parameters and finding its best configuration for a given application which is optimized for desired metrics – speed, area and power consumption.

The paper is organized as follows. Section 2 describes the state of the art in the processor optimization area. The information about the Codasip Studio and its features is provided in section 3. The architecture of our framework for finding the most optimal processor configuration for a given application is presented in section 4. Our experiments with finding the most optimal processor configuration for the given application and their results are described in section 5. Finally, section 6 concludes the paper.

2. Related Work

In current research in the field of processor optimization, the vast majority of works are focused on design space exploration using a simulation based approach. The simulation based approach uses a simulation model of an automatically generated processor architecture and the target application is simulated on this model. Performance statistics are the result of the simulation which is used for the evaluation of the model quality.

Amir Hossein Ashouri et al [2] focus on searching an optimal compiler configuration to maximize the application performance. Compilers provide a set of transformations of the source code which may favorably or adversely affect the final performance of the application. The authors use Bayesian Networks together with the information about the application characteristics and micro-architecture features to create a complex distribution function for searching the optimal sequence of compiler transformations. The same author [3] combines an optimization of compiler transformations together with a reconfigurable Roof-Line (VLIW) processor architectural model. First, a set of promising VLIW architectural candidates is generated based on the application characteristics and subsequently, different compiler transformations are performed on the set of customized VLIW architectures. Swarnalatha Radhakrishnan et al [17] present an ASIP processor with multiple heterogeneous pipelines for parallelism at the instruction level. The authors have designed a system able to generate a processor with the number of pipelines suitable for the given application. Each pipeline has a specialized instruction set that can be executed in parallel.

Giuseppe Ascia et al [1] present the EPIC-Explorer framework for the simulation of a parametrized VLIW-based platform. The platform allows an embedded system designer to optimize the system for a given application in terms of performance, area, and power consumption. The framework is able to evaluate the impact of architectural and micro-architectural features on these metrics. EPIC-Explorer operates on the Trimaran framework [23] which presents a parametrized compiler and a library for VLIW architectures. Trimaran takes an input source code of the application together with the architecture description and performs the compilation (static scheduling of the operations). It also generates a simulator of the VLIW processor on which the compiled application is executed as well as the corresponding dynamic execution statistics. Based on the statistics, the Estimator estimates the area, power consumption, and performance for the actually processed configuration of the architecture. A pareto frontier containing the optimal configurations is produced at the end of the optimization process. Fig. 1 shows the communication of the EPIC-Explorer with Trimaran. The authors provide a list of explorable processor parameters that include the size of the register file, number of functional units and caches.

In comparison with the approaches presented above, our research differs in several key aspects. In our approach, we use the Codasip Studio and an architecture description that allows us to modify arbitrary architectural parameters in the processor specification. A complete toolchain (including the simulation and synthesis tools) may be generated based on this architecture description. Using the simulation and synthesis tools, we are able to obtain the exact values of the resulting chip area, power consumption and execution time (cycles) for the given application and processor configuration. Our optimization platform described below

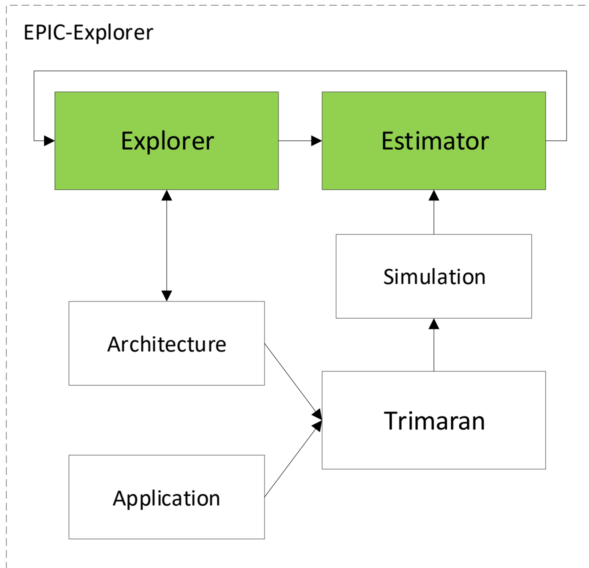

Figure 1. The architecture of EPIC-Explorer.

allows to choose the best optimization strategy for the selected processor thereby minimizing the total optimization time without losing the solution quality. Our framework is not limited to VLIW processors; it can optimize an arbitrary processor or a set of selected processors for the given application.

3. Codasip® Studio

The Codasip® Studio offers fully automatic retargeting of the programming and simulation toolchain, generation of a synthesizable RTL as well as a support for functional verification as shown in Fig. 2. It offers both the graphical user interface and the command line interface which is used by our system.

The Codasip Architecture description Language (CodAL) is a hierarchical and structured language that allows to define and describe processor cores at two abstraction levels. The higher instruction-accurate level of abstraction level is the key element for fast and accurate prototyping of an architecture. In early design stages, the processor core can be described at this abstraction level. The description is not limited to RISC processor architectures (ARM, MIPS) only; for the purpose of virtual prototyping, the CISC (e.g. x86) architecture can be modeled as well. Once the modeled instruction set is stable, the model of the microarchitecture can be created at the lower cycle-accurate abstraction level.

The simulator works in two basic modes: The instruction- or cycle-accurate simulation (more than 60 times faster when compared to the VHDL simulator on a single core), whose selection is based on the CodAL model. The debugging speed may be improved by using a retargetable compiled or a translated simulation, which offer a substantially higher simulation speed than the basic version of the interpreted simulator.

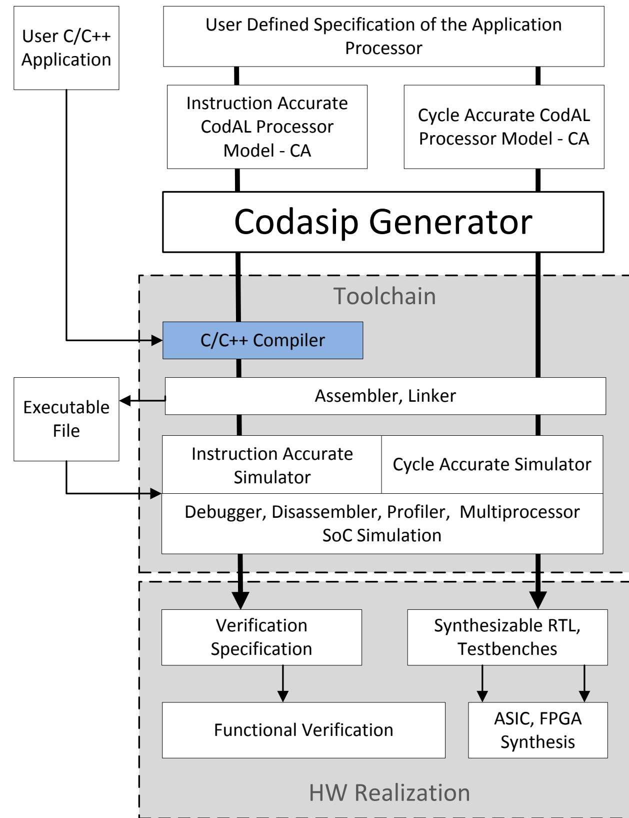

Figure 2. Codasip® Studio.

The generated profiler tracks and logs important information acquired during the simulation such as the most used instructions, unused instructions, cache misses, function call graph with instruction cycles, resources usage, etc. Based on this information, the optimization system can choose the next steps of the optimization.

When the architecture design is stable enough, a synthesizable RTL processor representation is generated. It is also possible to generate test benchmarks for the architecture, to generate asserts into the RTL description or optionally, to generate a support for the JTAG debugging interface. Note that the generated synthesizable RTL is well proven by third-party ASIC and FPGA synthesizers. Additionally, the equivalence between the simulation and the hardware has to be ensured bearing in mind that the behavior of the simulator should be the same as the behavior of the real

hardware. In case of Codasip®, this equivalence is guaranteed by the fact that all the principles and algorithms are based on formal models and are well proven. Moreover, the simulator and the hardware generators use the same algorithms for generation.

The Codasip® Studio allows to generate a C/C++ complier fully automatically from the processor definition. This allows us to avoid splitting the optimization process in two sequential steps, i.e. the compiler implementation as the first step and the optimization cycle as the second one. In other words, we are able to include the C/C++ compiler generation in one optimization cycle. This allows us to modify the instruction set in the tuning cycle or possibly, to change the processor completely.

4. Proposed Framework Architecture

We propose a solution of the processor optimization process based on an input application (program). We use processor models described in the CodAL language that have been developed by Codasip company. Any processor described in CodAL is configurable and can be optimized by changing its important parameters (number and size of registers, multipliers, caches, e.g.). The processor is optimized based on the following four metrics depending on the selected application:

- Number of cycles,
- Lines of assembler code generated by compiler,
- Area estimation,
- Power estimation (energy consumed during application run).

The values of these metrics may be obtained from the simulation using the generated toolchain or estimated from other provided values as discussed below.

The whole architecture of the proposed framework and the optimization process are shown in Fig. 3. We can divide the framework into several parts: The *input part*, the *process part*, and the *output part*.

4.1. The Input Part

The input part consists of three logical blocks which represent the setting of the optimization process. The *Application* is a C/C++ or assembly program for which we optimize. The application is later compiled and simulated by the *Codasip Studio Toolchain* generated for the given processor. The *Models of Processors with Changeable Parameters* represents the source models of processors (in CodAL language) which will be used

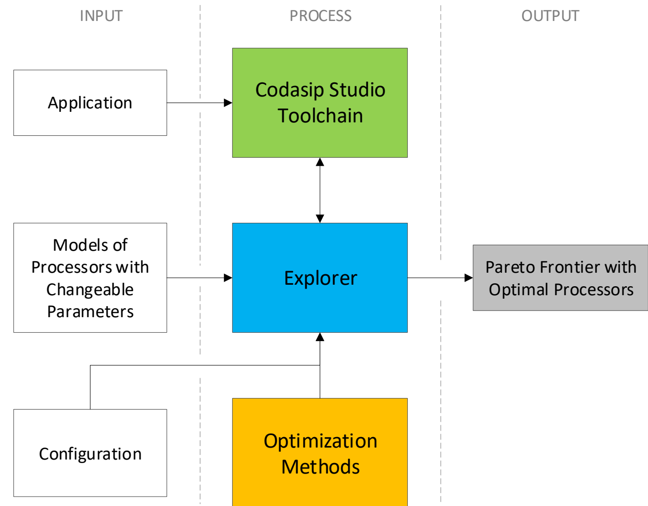

Figure 3. The architecture of the proposed optimization framework.

in the optimization process. For every processor, there is also a list of configurable parameters available with their possible values. In our tool, the configurable parameters of each processor are provided as an XML file with the following general structure:

```
<processor name="NAME" platform="MODEL"
 type="parameters">
    <parameter name="PARAM1">
        <value>VAL1</value>
        <value>VAL2</value>
        ...
    </parameter>
    <parameter name="PARAM2">
        ...
    </parameter>
    ...
</processor>
```

The *name* and *platform* attributes of the *processor* tag are used for further identification of the generated toolchain. The *type* attribute defines the type of the XML structure (input parameters in this case). The same structure with a different *type* is used for storing the final configuration of the processor. The difference is only in the number of possible values; in the final configuration, there is only a single value assigned to each parameter. Each parameter of the processor is represented by a *parameter* tag whose the *name* attribute is an identification used in the model of the processor. The *parameter* element contains a list of all possible *values* of the given parameter. From the list of parameters and their values, the individual configurations of the processor are obtained and used in the evaluation process.

The architecture is designed to use an arbitrary optimization strategy. Therefore, the last input is a *Configuration* that includes the specific parameters for the framework and for each optimization strategy. The specific parameters for the framework include the application name, processor names, result file names, maximal number of threads for parallel evaluation and the Pareto frontier configuration.

4.2. The Process Part

The *Explorer* is the core of the process part and it manages the entire optimization process. Based on the possible values of the changeable parameters, the Explorer creates individual configurations of the selected processor (in XML format) and sends them together with the adequate processor model into the *Codasip Studio Toolchain*. A selected *Optimization Method* decides how the specific configurations of the given processor will be created. The optimization method can be any simple algorithm such as a traversal of all possible solutions or sophisticated one such as a genetic algorithm. The Codasip Studio Toolchain modifies the processor model based on the given configuration. For the modification of the processor model, we use a Python-based templating system that allows custom modifications of the model source code according to the actual parameter values. After the final model has been created, the Codasip Studio Toolchain generates the corresponding toolchain containing the compiler and simulator. The *Application* is then compiled and simulated using the toolchain. From this step, we obtain the values of the *lines of code* and *number of cycles* metrics while the *area* and *power consumption* are estimated from the model definition itself.

The Explorer stores the currently processed configuration together with the measured metrics values into an XML result file and based on the used optimization method, it chooses the next processor configuration to be examined or finishes the optimization process. In the last step, the Explorer generates the graphs with the Pareto frontier for each metric based on the optimal configurations of the processors. The freely available Google Charts [8] framework is used for the graph rendering.

4.3. The Output Part

The output of the optimization process is a *Pareto Frontier with Optimal Processor Configurations.* It contains the XML files and graphs with the monitored metrics and the corresponding optimal configurations of the processors. The framework produces the Local Pareto frontiers and a Global Pareto frontier. The local Pareto frontiers compare the monitored metrics by pairs – always two metrics per a single view. The global Pareto frontier shows the optimal configurations across all metrics. Based on the chosen importance of the metrics, the designer is then able to select the best processor configuration from the perspective of the remaining metrics. The general structure of the result XML file follows:

```
<experiment>
  <run id="1">
    <processor name="NAME" platform="
       MODEL" type="configuration">
        <parameter name="PARAM1">
            <value>VALX</value>
        </parameter>
        <parameter name="PARAM2">
            <value>VALY</value>
        </parameter>
    </processor>

    <result>
        <cycles>...</cycles>
        <lines>...</lines>
        <area>...</area>
        <power>...</power>
    </result>
  </run>
  <run id="2">
      ...
  </run>
  ...
</experiment>
```

The result file is composed of *runs* which represent the individual processors configurations and their measured values. Each *run* has two parts – *processor* and *result*. The *processor* contains the actual processor configuration that was used. The *result* part provides the measured metrics values for the given configuration. The Pareto frontier is then composed of the runs which are optimal according to the given optimization method.

4.4. Optimization Methods

Our architecture is designed to operate with different optimization strategies which are incorporated into the framework through plugins. This design allows to create a set of plugins implementing a number of completely different optimization strategies. For example, a simple method of Cartesian product (full state space exploration) can be implemented as well as other more advanced methods like a genetic algorithm (GA) [16] or simulated annealing [19]. The activity of the optimiza-

tion methods lies mainly in the selection of processor parameters while the quality of the searched solution is preserved. The full state space exploration is discussed in the following subsection. The genetic algorithm or simulated annealing are just examples of alternative optimization strategies that can be used in case of a large state space where the full space exploration is not feasible.

Cartesian product is well known mathematical operation which returns the set of all ordered n-tuples from the n input sets. In our case, the input sets are the possible values of the individual configurable parameters of the given processor. An ordered n-tuple represents a particular configuration of the processor. The number N of all possible configurations can be easily calculated with the following formula:

$$N = \prod_{i=1}^{numOfParameters} numOfValues(i) \quad (1)$$

The resulting value of N is helpful for deciding which optimization method is suitable for the best performance of the framework with respect to the run time and the quality of the discovered solution. The full state space exploration is optimal for the use with a small number of possible configurations because it goes through whole state space in a short time. On the contrary, for high number of configurations, this method is very time consuming and therefore, some more sophisticated method is necessary. In our experiments, we use this method for number of configurations smaller than 1000.

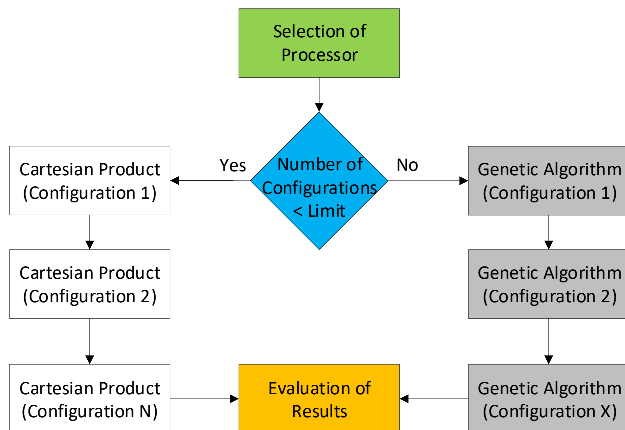

Figure 4. Proper selection of the optimization methods.

4.5. Implementation Details

The proposed optimization framework is implemented in Python. Python is a high-level programming language for general purpose programming. The implementation uses the practices of object oriented programming. Each block of the proposed framework structure (Fig. 3) is implemented as a separate class which groups the logical elements together. The Explorer (the controller of the whole optimization process) is extended for each optimization method that defines the specific optimization technique. Each optimization technique is obliged to implement the *run* method which defines the entry point of the process.

The optimization process starts by the initialization of the framework with a parameter defining optimization method. The initialization loads and processes the configuration file, where the target application and the processor(s) are selected and the parameters of the optimization methods are set. Also, it loads the source XML file(s) containing the configuration(s) of the processor(s) and creates a class instance for them and for saving the results. As the next step, the selected optimization method is executed.

The optimization method selects the most appropriate configuration and sends it to model pre-processor. Its responsibility is to modify the processor CodAL model according to the configuration. For this purpose, we use the *Jinja2* library [18] that allows to use special macros in the source files. Subsequently, special scripts are called to perform the model compilation and simulator generation through the Codasip Studio. The target application is simulated on the selected processor. After the simulation, Codasip Studio provides extensive statistics from which the monitored measured metrics are acquired back to the optimization method, which continues in its computation.

After the completion of the optimization method, the results are stored in an XML file and the graphs are plotted. An object, which controls the results generates a set of auxiliary results used for the graph creation. The auxiliary results are forwarded to the Google charts [8] framework for plotting the graphs.

5. Experimental Results

As the first test case, we used the Codix Berkelium processor which is a RISC-V processor implementation and its optimization is performed through Cartesian product. The aim of the experiments is to present the use of our framework to search the optimal configuration of the processor and the optimal compiler flags for the given application with respect to the monitored metrics.

We have selected seven parameters of the Codix Berkelium processor that may be changed. These parameters are governed by the user specification (User-

Level ISA Specification) of the RISC-V processor [24]. The parameters represent a total of 252 hardware configurations of the processor which have to be take into account during the optimization process. The list of the selected parameters together with their values can be seen in Table 1.

Table 1. The list of the changeable parameters for the Codix Berkelium processor.

EXTENSION_E	true, false
EXTENSION_M	true, false
EXTENSION_C	true, false
ENABLE_ICACHE	true, false
ICACHE_LINE_SIZE	16, 32, 64, 128
ICACHE_SIZE	4, 8, 16, 32, 64
ENABLE_PARALLEL_MUL	true, false

The EXTENSION_E parameter sets the size of registers to 16-bits when set to true; 32-bit registers are used otherwise. The EXTENSION_M parameter ("M" Standard Extension for Integer Multiplication and Division) enables or disables the use of instructions for multiplying or dividing values held in two integer registers. The EXTENSION_C parameter ("C" Standard Extension for Compressed Instructions) enables or disables the use of compressed instructions for common operations. The compression offers 16-bit versions of 32-bit instructions. ENABLE_ICACHE enables or disables the instruction cache of the processor. If the instruction cache is enabled, the instruction cache line size (ICACHE_LINE_SIZE) and instruction cache size (ICACHE_SIZE) can be also configured. The ENABLE_PARALLEL_MUL enables or disables parallel multiplication. Parallel multiplication is faster than the sequential one but the chip area is larger.

The framework is able to work not only with the processor parameters, but also with the compiler flags. All the changeable parameters can be specified in the input description as well. There are lot of standard flags of the used LLVM compiler [13] that can be set. By including all the compiler flags, we would get a huge amount of configurations for evaluation and the whole experiment would be very time consuming due to performing the Cartesian product for all the possible combinations. Therefore, we chose only a small subset of compiler flags which are frequently used for compiling applications. The list of the selected compiler flags can be seen in Table 2.

The `-o0` flag disables all optimizations. The `-o1`, `-o2` and `-o3` flags perform level-1, level-2 and level-3 optimizations. The `-os` flag optimizes for size while `-ofast` performs the level-3 optimizations and disregards strict standards compliance. The `-ffast-math` flag sets several other flags for the fastest math mode optimization. The `-ffunction-sections` flag places functions into their own sections and in combination with linker options, it can remove all unused code. The `-finline-functions` flag inlines suitable functions for a fast access like macros. [22] The `-ffast-math` can be combined with the `-o2` flag, the `-ffunction-sections` flag can be combined with `-o3` flag and `-finline-functions` flax can be combined with `-ofast` flag. There is 9 options for compiler. In total, there is 2268 configurations (252 HW configurations multiplied by 9 compiler options).

Table 2. The subset of flags for the LLVM compiler.

-o0, -o1, -o2, -o3, -os, -ofast	optimization level
-ffast-math	fastest math mode
-ffunction-sections	functions in its own sections
-finline-functions	declare functions as inline

As the testing applications for which the processor is optimized, we have selected several different implementations. The applications differ in the computational complexity, memory requirements, used instructions and functional units:

1. The *Face detector* [12] (FACES) is a C/C++ application. Its aim is to detect faces of people in various images using a neural network. The application uses the OpenCV library [4] for image manipulation. The face detector can be used in two ways – in training mode and detection mode. In the training mode, the detector is trained on a given set of images. There are 10 images in the source files of the detector which were taken from Face Detection Data Set and Benchmark (FDDB) [11]. In the detection mode, the detector is able to detect faces in an image based on the information obtained in the training mode. In our optimizations, the training mode is selected, because the detection mode is data-dependent on currently processed image, and therefore the results will not be meaningful. Processing more different images in the training mode is more interesting in terms of uniform utilization.

2. The *Audio codec G.722.1* (DECODE) is a real broadband audio codec standardized by ITU-T (International Telecommunication Union – Telecommunication Standardization Sector)

[9]. It is a voice encoding and decoding codec for VoIP (Voice over Internet Protocol) and similar applications. It offers good audio quality with reasonable bitrate. The application has a standard ITU-T implementation in C/C++ that has been customized for our framework. Only the decompression of the previously encoded data which is stored in the memory of the application memory is performed. The application uses the data which the ITU-T has released with the codec as the reference.

3. The *Advanced Encrypt Standart* [14] (AES) is a benchmark based on a modern AES 128 cipher. It encrypts the data field contained in the memory and then re-encrypts it. Two rounds of encryption and decryption are performed. Both rounds are done in memory, the first one in the CBC (Cipher Block Chaining) mode and the second one in the ECB (Elektronic CodeBook) mode. The data field can be easily extended and can easily call multiple encryption consecutive cycles.

Two types of charts can be generated as the output of our framework. The first chart is a Pareto frontier which shows a single output metric (cycles, lines, area, power) per axis. In case of four metrics, a 4D would have to be created. Therefore, we generate partial charts for every pair of metrics (lines vs. power, lines vs. cycles, area vs. power etc.). A Pareto frontier is generated for each of these pairs and the appropriate solution can be found in these charts depending on the user preferences and requirements.

The second type of charts shows the values of all parameters for the individual tested configurations. On the x-axis, there are the configuration identifiers (sequential indexes) and on the y-axis, we may see the values of all metrics (cycles, lines, area, power). The configurations can be ordered according to a selected metric. This allows to watch the trends of the remaining metrics. This chart is just a supporting tool for the user that complements the Pareto frontier.

The chart in Figure 5 shows the number of cycles for the three benchmarks mentioned above – FACES, DECODE and AES. The configurations are ordered by the number of cycles for the Faces application. Only the configurations with the `-o2` optimization flag are taken into account in this chart. The chart shows that different configurations have different impact for the individual applications.

The next two charts are related to the DECODE application and show the impact of various compiler parameters. The first one in Figure 6 shows the number of program lines. The order of configurations is the

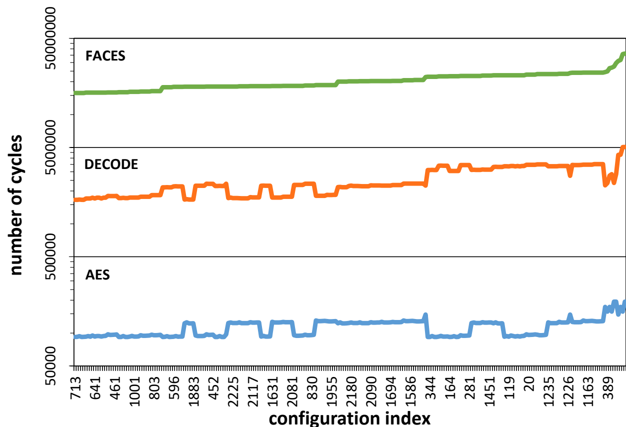

Figure 5. The number of cycles for the AES, DECODE and FACES applications when using the `-o2` optimization only.

same as in the previous chart. The top line shows the number of lines of code for configurations with the `-o0` optimization flag (without optimizations) which leads to the solution with high number of program lines. The second and the third line in the chart represent the configurations for the `-o3` (same as `-ofast`) and `-o2` optimization flags. These two lines are almost the same, the optimizations have similar effects in case of the DECODE application. The same situation occurs for the `-os` and `-o1` optimization flags. The best effect is achieved by the `-o3` flag with the `-ffunction-section` optimization flag which indicates a large amount of unused code in the application.

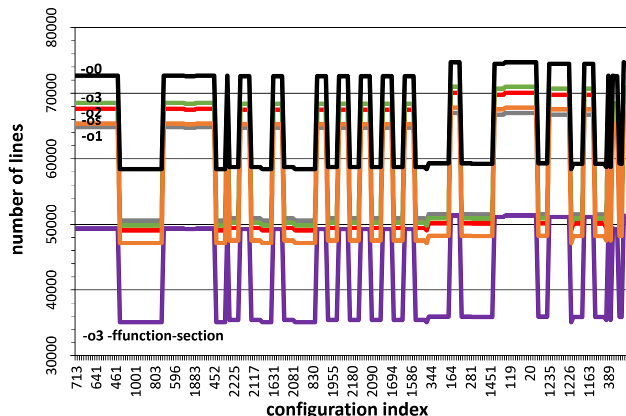

Figure 6. The number of lines for the DECODE application with different compiler flags.

Figure 7 shows another view on the same application (DECODE) – the number of cycles for each configuration. The lines correspond to the same groups of optimization flags but only three of them are clearly visible. No optimizations (`-o0`) leads to the worst solutions similarly to the previous chart. Then, there is a big step

to the `-o1` optimization. The last line stands for the overlapping lines for the `-o2`, `-o3`, `-ofast`, `-os`, `-o2` with `-ffast-math`, `-o3` with `-ffunction-section`, and `-ofast` with `-finline-functions` optimization flags. If we take both charts into account, it can be deduced that `-o3` with `-ffunction-section` optimization is the most advantageous configuration for the selected application from the point of view of the given metrics.

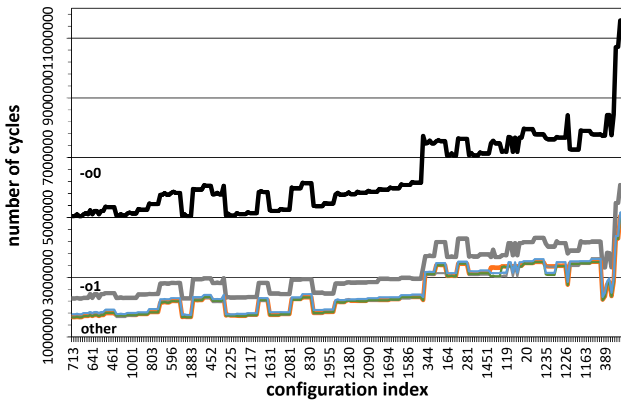

Figure 7. The number of cycles for the DECODE application with different compiler flags.

One of the most important outputs is the Pareto frontier that shows the most interesting solutions for further choice. Figure 8 shows a partial Pareto frontier for the number of cycles vs. the chip area where chip area is a dimensionless number computed by the generated Codasip profiler tools. This number can be used to compare the chip area among different configurations.

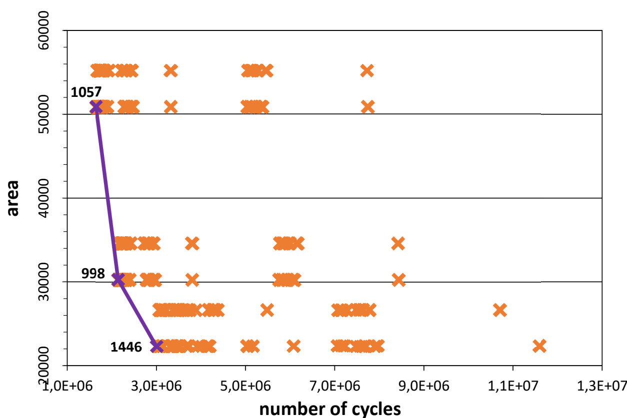

Figure 8. The number of cycles and the chip area for the DECODE application with the Pareto frontier highlighted.

The designer may choose any of the configurations; however, only the configurations on the Pareto frontier are optimal. In the example in Fig. 8, we may see three points on the Pareto frontier. The user can use the configuration with number 1057 if speed is preferred or, on other hand, the configuration number 1446 if area is more important. The configuration number 998 is a compromise. Of course, there are more metrics (lines, powers) which must be taken into account. The detailed information about the given Pareto frontier configurations is summarized in Table 4. The configurations number 1057 and 998 differ in the PARALLEL_MUL parameter which enables the parallel hardware multiplication. It leads to higher speed but also to higher chip area. The lines *area* and *power* are numbers without units which are provided by Codasip Tools and serve just for comparison between configurations.

Times needed for generating a toolchain for an one configuration and for simulating the selected application are shown in Table 3.

Table 3. Times needed for generation and simulation of an one processor configuration.

Codasip Tools generation [min:sec]								
≈ 2:30								
Application simulation [min:sec]								
FACES			DECODE			AES		
min	max	avg	min	max	avg	min	max	avg
4:30	17:35	7:08	0:28	3:18	0:51	0:02	0:48	0:06

6. Conclusion and Future Work

The framework for searching the most suitable configurations of processor parameters and compiler flags for a selected application was presented in this paper. The framework is based on the simulation of processors and evaluation of the obtained results. As a modeling and simulation tool, the Codasip framework is used. The presented framework is divided in three main parts: The input part for the specification of the processor configurations, the processing part for searching and evaluating the possible solutions and the output part that saves the results and transforms the obtained results into charts. A Pareto frontier of the possible solutions is the main output of proposed system. The experiments with RISC-V-based Codix Berkelium processor and the full state space exploration were also presented. Three benchmarks were used as the sample applications on which the whole framework was demonstrated. The main part of the experimental results are the obtained charts which demonstrate the possible outputs of our system.

Our proposed system is able to use various types of optimization algorithms; currently, only the Cartesian product is implemented. One of the goals of our

Table 4. The Pareto frontier configurations.

configuration index	1057	998	1446
EXTENSION_E	1	1	1
EXTENSION_M	1	1	0
EXTENSION_C	0	0	0
ENABLE_ICACHE	1	1	1
ICACHE_LINE_SIZE	128	128	128
ICACHE_SIZE	32	32	32
PARALLEL_MUL	1	0	0
OPTIMIZATION	-O3	-O3	-O3
OPTIMIZATION	-ffunction-sec	-ffunction-sec	-ffunction-sec
cycles	1654286	2151840	3011164
lines	49284	49284	51120
area	50817	30243	22335
power	79444470468	59067459084	58843776132

further research is to use genetic algorithms or other artificial intelligence and optimization methods to reduce the search time when using multiple compilation flags. Since some similarities may be observed in the output charts for various applications, we also plan to automatically discover different classes of similar applications and to develop an automatic assignment to these classes as our future work.

Acknowledgment

This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II); project IT4Innovations excellence in science – LQ1602.

References

[1] G. Ascia, V. Catania, M. Palesi, and D. Patti. Epic-explorer: A parameterized VLIW-based platform framework for design space exploration. In *First Workshop on Embedded Systems for Real-Time Multimedia, ESTImedia 2003, October 3-4, 2003, Newport Beach, California, USA, co-located with CODES-ISSS 2003, Proceedings*, pages 65–72, 2003.

[2] A. H. Ashouri, G. Mariani, G. Palermo, and C. Silvano. A bayesian network approach for compiler autotuning for embedded processors. In *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, pages 90–97, Oct 2014.

[3] A. H. Ashouri, V. Zaccaria, S. Xydis, G. Palermo, and C. Silvano. A framework for compiler level statistical analysis over customized VLIW architecture. In *2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 124–129, Oct 2013.

[4] G. Bradski and A. Kaehler. *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008.

[5] Cadence. *Xtensa LX7 Processor*, accessed 5/2018. 13 pages, <https://ip.cadence.com/uploads/1099/TIP_PB_Xtensa_lx7_FINAL-pdf>.

[6] Codasip. Processors for the connected world, accessed 5/2018. <http://www.codasip.com>.

[7] Codasip. RISC-V processors, accessed 5/2018. <https://www.codasip.com/risc-v-processors/>.

[8] Google. Google charts: Quick start, accessed 5/2018. <https://developers.google.com/chart/interactive/docs/quick_start>.

[9] Y. Hiwasaki and H. Ohmuro. ITU-T G.711.1: extending G.711 to higher-quality wideband speech. *IEEE Communications Magazine*, 47(10):110–116, October 2009.

[10] M. K. Jain, M. Balakrishnan, and A. Kumar. Asip design methodologies: survey and issues. In *VLSI Design 2001. Fourteenth International Conference on VLSI Design*, pages 76–81, 2001.

[11] V. Jain and E. Learned-Miller. FDDB: A benchmark for face detection in unconstrained settings. Technical Report UM-CS-2010-009, University of Massachusetts, Amherst, 2010.

[12] R. Juranek. LBP detector. <git@gitlab.fit.vutbr.cz:ijuranek/lbpdetector.git>, 2016.

[13] LLVM Developer Group. The LLVM compiler infrastructure, 2007. <https://llvm.org/>.

[14] F. P. Miller, A. F. Vandome, and J. McBrewster. Advanced encryption standard. 2009.

[15] P. Mishra and N. Dutt. *Processor description languages*, volume 1. Morgan Kaufmann, 2011.

[16] M. Mitchell. *An Introduction to Genetic Algorithms.* A Bradford book. Bradford Books, 1998.

[17] S. Radhakrishnan, H. Guo, and S. Parameswaran. Customization of application specific heterogeneous multi-pipeline processors. In *Proceedings of the Design Automation Test in Europe Conference*, volume 1, pages 6 pp.–, March 2006.

[18] A. Ronacher. Jinja2 (The python template engine), 2014, accessed 5/2018. <http://jinja.pocoo.org/>.

[19] B. Suman and P. Kumar. A survey of simulated annealing as a tool for single and multiobjective optimization. *Journal of the operational research society*, 57(10):1143–1160, 2006.

[20] Synopsys. *ASIP Designer Application-Specific Processor Design Made Easy*, accessed 5/2018. <https://www.synopsys.com/dw/ipdir.php?ds=asip-designer>.

[21] Synopsys. *DesignWare Processor IP Portfolio*, accessed 5/2018. 8 pages, <https://www.synopsys.com/dw/doc.php/ds/cc/arc_processor_solutions.pdf>.

[22] The Clang Team. Clang command line argument reference, 2015, accessed 5/2018. <https://clang.llvm.org/docs/ClangCommandLineReference.html>.

[23] Trimaran. An infrastructure for research in backend compilation and architecture exploration, 2010, accessed 5/2018. <http://www.trimaran.org//>.

[24] A. Waterman, Y. Lee, D. Patterson, and K. Asanovic. The RISC-V instruction set manual. *volume I: User-level ISA, version 2.0, EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54*, 2014.