Synthetic Benchmark Generator for the MOLEN Processor

Stephan Wong, Guanzhou Luo, and Sorin Cotofana
Computer Engineering Laboratory,
Electrical Engineering Department,
Delft University of Technology,
{Stephan,Guanzhou,Sorin}@Dutepp0.ET.TUDelft.NL

Abstract— **Most of current embedded processor designs utilize programmable processor cores augmented with specialized hardwired units. Recently, we are witnessing a shift in this design methodology towards the utilization of reconfigurable hardware (e.g., field-programmable gate arrays (FPGAs)) instead of fixed hardware. In this light, the MOLEN processor has been proposed that employs reconfigurable microcode ($\rho\mu$-code) to control both the reconfiguration and execution processes of the reconfigurable hardware. The $\rho\mu$-code is an extension to the traditional microcode which includes support for reconfigurable hardware. In this paper, we propose an extension to the generally accepted hardware/software co-design methodology that accomplishes the intended augmentation of reconfigurable hardware to programmable processors. Additionally, the utilization of $\rho\mu$-code requires on-chip storage for which the storage size, the organization, and other design parameters must be investigated. To this end, we propose to generate synthetic $\rho\mu$-code that only exhibit those characteristics that are relevant in the mentioned investigation.**

Keywords— **field-programmable gate array, synthetic benchmarks, reconfigurable computing, reconfigurable microcode.**

I. Introduction

In embedded processor design, we are clearly witnessing a shift in design methodology that is increasingly favoring the utilization of programmable processor cores over application-specific integrated circuits (ASICs). Such programmable processor cores can be either a general-purpose processor (GPP) or a programmable digital signal processor (DSP). This shift has been fuelled by the industry's need to shorten the lengthy design cycles of embedded processors and to reduce the associated high design costs. In particular, design cycles of 18 months or longer are rather the rule than exception. Programmable processor cores have been introduced, because their programmability allows them be utilized in subsequent design cycles and thereby reducing design time and costs. However, the general consensus is that such processor cores are unable to achieve the performance requirements of many (real-time) applications. In this light, such cores usually have been complemented with specialized hardware (units) that increase the performance of the overall system to an acceptable level. Summarizing, in subsequent design cycles the processor core can be re-used and only a small number of specialized units need to be (re-)designed. We have to note that the effort of the described design methodology is considerably less than in designing a full-fledged ASIC-based embedded processor.

Recently, we are witnessing a trend that is implementing the mentioned specialized units in reconfigurable hardware, e.g., field-programmable gate arrays (FPGAs), rather than fixed hardware. This approach increases the embedded processor design flexibility by allowing fast prototyping. Furthermore, continued technological advances in FPGA technology will certainly allow reconfigurable hardware structures to be utilized for increasingly more applications. With the purpose of tightly integrating a reconfigurable hardware structure with a programmable processor, the MOLEN processor [4] was proposed which has several advantages over other similar approaches[1], [3], [6], [2]. To this end, reconfigurable microcode ($\rho\mu$-code) has been introduced that extends the capabilities of traditional microcode[5] to include support for reconfiguration of FPGA structures.

In this paper, first we introduce the new hardware/software co-design methodology that forms the basis of the MOLEN approach. In this methodology, $\rho\mu$-code is utilized that controls the reconfiguration and execution processes of the reconfigurable hardware (unit). The MOLEN approach can support an infinite number of implementations on the FPGA structure as long as they fit on the FPGA structure. Consequently, it is illogical to store all the resulting $\rho\mu$-codes on-chip. Therefore, it is proposed to include a storage unit (called $\rho\mu$-code unit) on-chip that permanently stores frequently used $\rho\mu$codes and temporarily stores less frequently used $\rho\mu$-codes in order to diminish their loading times. To allow the MOLEN approach to be taken up by the industry, it is important to de-

termine several design parameters of this $\rho\mu$-code unit: its storage size and organization, and its 'caching' algorithms. To this end, $\rho\mu$-code is needed that serves as input for a $\rho\mu$-code unit simulation model. The needed $\rho\mu$-code can be obtained in two different ways. First, perform numerous hardware designs and derive their corresponding $\rho\mu$-code. The real $\rho\mu$-code generated in this way will result in a more precise determination of the $\rho\mu$-code unit's design parameters. The main disadvantage of the described method is that the process is quite lengthy. In addition, we will show that no actual (functional) $\rho\mu$-code is needed to determine the design parameters. Second, generate synthetic $\rho\mu$-code that exhibits only those characteristics that are needed in mentioned $\rho\mu$-code unit's design parameters investigation. The main advantage is that the characteristics of the synthetic $\rho\mu$-code can be easily changed and therefore allowing a much faster design space exploration. Consequently, when actual $\rho\mu$-code has been generated and its characteristics has been determined, a much narrower design space need to be explored in order to determine the required design parameters as the design space exploration which utilized the synthetic $\rho\mu$-code can be used as a starting point.

Subsequently, in this paper we propose a method to (semi-)automatically generate synthetic (non-real) microcode with the main purpose of determining the design parameters of the $\rho\mu$-unit. First, we establish the $\rho\mu$-code characteristics that are needed and then implement a software tool that generates the $\rho\mu$-code accordingly. This paper is organized as follows. In Section II, we briefly discuss the organization of the MOLEN processor. In Section III, we present the new hardware/software co-design methodology of the MOLEN approach that utilizes microcode in the augmentation of reconfigurable hardware to a general-purpose processor core. In Section IV, we discuss the requirements that are needed in order to perform the $\rho\mu$-code unit's design parameters investigation and how these are translated into synthetic microcode requirements. In Section V, we present the software implementation of the microcode generation tool. We conclude this paper by presenting some concluding remarks in Section VI.

II. The MOLEN Processor

In its more general form, the proposed machine organization, which is augmented with a reconfigurable unit, is depicted in Figure 1. In this organization, instructions are fetched from the main memory and are temporarily stored in the 'Instruction Fetch' unit. Subsequently, these instructions are fetched by the 'Arbiter' which partially decodes them before issuing them to their corresponding execution units. Instructions that have been implemented in fixed hardware are issued to the 'Core Processing Units', i.e., the regular functional units such as ALUs, multipliers, and dividers. The instructions related to the reconfigurable unit are issued to it accordingly. More specifically in our case, they are issued to the reconfigurable microcode unit or '$\rho\mu$-code unit'. The precise functionality of this unit is explained in Section III. At this moment, it is only important to recognize that it provides fixed/permanent and pageable storage for reconfiguration and execution microcode that control the reconfiguration and execution processes on the 'Custom Configured Unit'[1] (CCU), respectively. In the remainder of this paper, we refer to both reconfiguration and execution microcode as $\rho\mu$-code (explained in Section III). The loading of microcode to the '$\rho\mu$-code unit' is performed via the 'Arbiter' which accesses the main memory through the 'Data Fetch' unit.

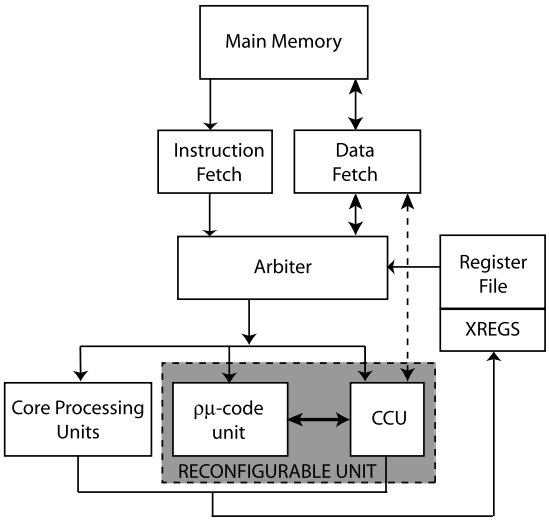

Fig. 1. General organization of the MOLEN processor.

Similar to other load/store architectures, the proposed machine organization executes on data that is stored in the register file and prohibits direct memory data accesses by instructions other than the $load$ and $store$ instructions. However, there is one exception to this rule, the CCU is also allowed direct memory data access via the 'Data Fetch' unit (represented a dashed two-ended arrow). This enables the CCU to perform much better when streaming data accesses are required, e.g., in multimedia processing. Finally, exchange registers (XREGS) are included in order to provide the input and output interface that is needed to communicate arguments and results between the implemented function and the remainder of the application code (see stage 4 in Figure 2). When only a small amount of

[1]Such a unit could be for example implemented by a Field-Programmable Gate Array (FPGA).

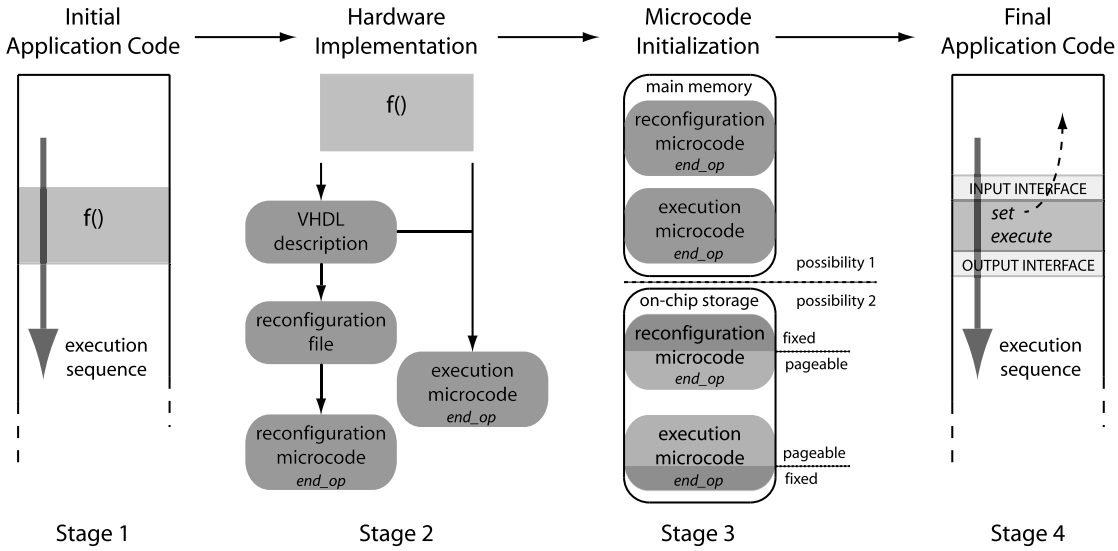

Fig. 2. Extending the hardware/software co-design with microcode concepts.

data needs to be communicated, the register file suffices. However, by architecturally including the exchange registers, a more general communication framework[2] can be provided in order to communicate an arbitrary number of arguments and results.

III. A New Co-Design Methodology

In the past, the augmentation of reconfigurable hardware with general-purpose processor cores has been done in an ad-hoc manner. In this section, we propose a more generic methodology (depicted in Figure 2) that extends the currently accepted hardware/software co-design methodology by re-introducing the microcode concept. In this methodology, we start by writing the application code intended to be executed solely on the general-purpose processor core (stage 1). Using this as a starting point, we continue by identifying 'bottleneck' functions that when are sped up will most likely result in an overall performance increase of the whole application. Such functions are then implemented in reconfigurable hardware by first writing high-level VHDL code and afterwards performing a synthesis targeting the utilized FPGA structure. We differ from the more traditional approach in that we generate reconfiguration and execution microcode that control both the reconfiguration (or setting) process and execution process of the reconfigurable hardware, respectively (stage 2). The generated $\rho\mu$-code is then stored in either the main memory or on-chip for fast accessing (stage 3). Finally, the original application code is modified by first introducing two new instructions, namely set and $execute$. By executing the set instruction, the setting of the FPGA to perform the required function is performed. By executing the $execute$ instruction, the actual execution of the function on the FPGA is performed. We have to note that both instructions do not specify the function(s) to be performed, instead the function(s) are performed by executing the $\rho\mu$-code (until end_op) which is pointed to by the set and $execute$ instructions. Additionally, an input/output interface is required in order to communicate the argument(s) and result(s) between the implemented function and the remaining software modules (stage 4). The innovation of the MOLEN approach lies in the fact that only two new instructions set and $execute$ are required to support any implementation of current and future functions. In other words, no new instructions need to be introduced every time a new function must be supported. Furthermore, this approach allows the emulation of any function, either being a single instruction or a piece of code, to be supported.

As already mentioned, the MOLEN approach utilizes $\rho\mu$-code to control the setting of the CCU (see Figure 1) to perform a specific function and the execution of that function on the CCU. The needed $\rho\mu$-code can be stored in either the main memory or on-chip close to the CCU in the $\rho\mu$-code unit. A simplified view of the internal organization of the $\rho\mu$-code unit is depicted in Figure 3. In this figure, the storage of the $\rho\mu$-code associated with the set and $execute$ instructions are stored separately in the SET and EXECUTE storage facilities, respectively. A more detailed organization of either storage facilities within the ρ-Control Store is given in Figure 4. The storage is further divided into a fixed part and a pageable part. The fixed part contains the $\rho\mu$-code that is commonly used throughout the supported range of functions and therefore will benefit

[2]The precise organization and communication mechanisms of the exchange registers is subject for future research. Therefore, no details are provided in this section.

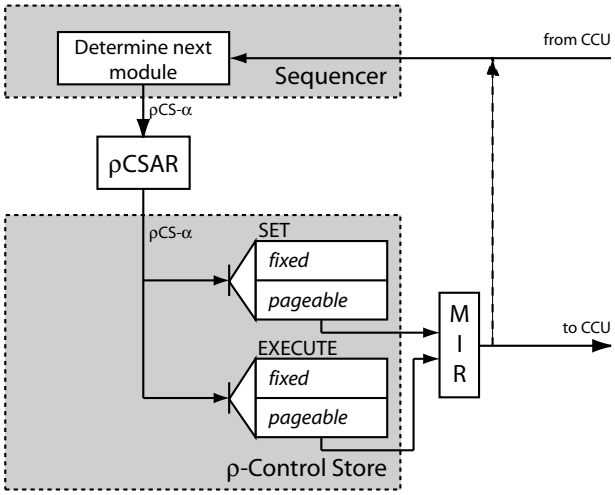

Fig. 3. Internal organization of the $\rho\mu$-code unit.

mostly when they are permanently stored on-chip. The pageable part contains the $\rho\mu$-code that is occasionally used and therefore only need to be temporarily available on-chip. Finally, it is important to note that the microinstructions are loaded into blocks which is similar to the fact that multiple words are loaded into a line in regular caches.

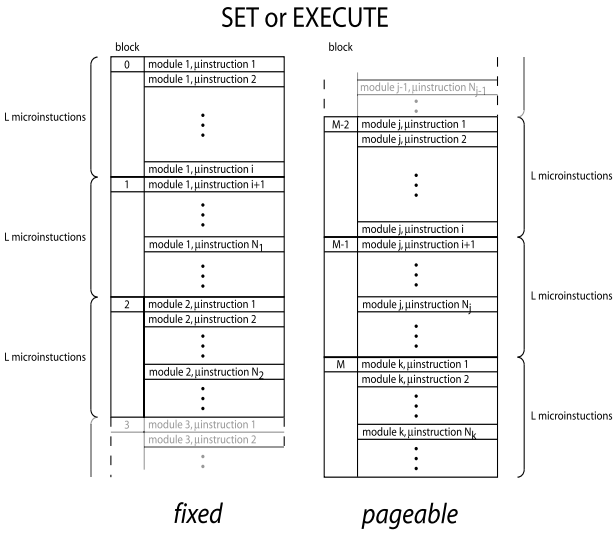

Fig. 4. Internal organization of the SET and EXECUTE storage.

The $\rho\mu$-code execution is performed as follows. First, using the ρ-Control Store address (ρCS-α), it is determined which block of microcode must be executed. Then, microinstructions within such a block are forwarded to the microinstruction register (MIR). Depending on the microinstruction executed or results from the CCU, the next microinstruction to be executed is determined by the sequencer. This process ends when an end_op microinstruction is encountered.

In this framework, it is important to investigate several design parameters of the $\rho\mu$-code unit, like its organization and storage size, and its loading and caching mechanisms. To allow a faster investigation, a simulation model of the $\rho\mu$-code unit is utilized and $\rho\mu$-code is needed to serve as input. The most accurate determination will be obtained when real $\rho\mu$-code is being utilized in the investigation, but performing numerous hardware designs in order to obtain the needed $\rho\mu$-code is a lengthy process. Therefore, we opted to generate synthetic $\rho\mu$-code that exhibit the required characteristics in order to perform the mentioned investigation. An additional advantage is that when real $\rho\mu$-code has been generated and its characteristics determined, an even faster design space exploration can be performed since we can limit the design space. The requirements to perform this investigation are discussed in the next section.

IV. Synthetic microcode generation

In this section, we derive the input parameters of the synthetic microcode generation tool as follows. First, we establish what design parameters of the $\rho\mu$-code unit must be determined. Then, we derive the characteristics of the generated synthetic microcode which in turn can be easily translated into input parameters of the envisioned microcode generation tool. The design parameters of the $\rho\mu$-code unit that must be determined are the following:

- **Storage size and organization:** It must be determined what the storage size is of on-chip storage for frequently used and non-frequently used $\rho\mu$-code. More specifically, we must determine the sizes of the fixed and pageable storages. Furthermore, we must determine the internal organization of the on-chip storage for which a possible organization (utilizing blocks) was suggested in Figure 4.
- **Caching mechanisms:** Having determined the storage size and organization of the $\rho\mu$-code unit, we must determine how this is utilized to permanently or temporarily store the $\rho\mu$-code. More specifically, we must determine the loading mechanisms and replacements strategies of $\rho\mu$-code in the $\rho\mu$-code unit.

Before we continue our discussion, we have to note that $\rho\mu$-code and microcode in general are composed of modules which in turn contain the microinstructions. Such modules can be perceived as small (micro-)programs that can call each other during execution. In the following, we describe two main characteristics of the $\rho\mu$-code that will certainly have an influence on the mentioned design parameters:

- **Mix of modules**: This characteristic mostly influences the storage size and organization of the $\rho\mu$-code unit. $\rho\mu$-code that contains many frequently used modules are likely to require a large fixed storage, because storing them in the pageable storage will most likely result in many re-

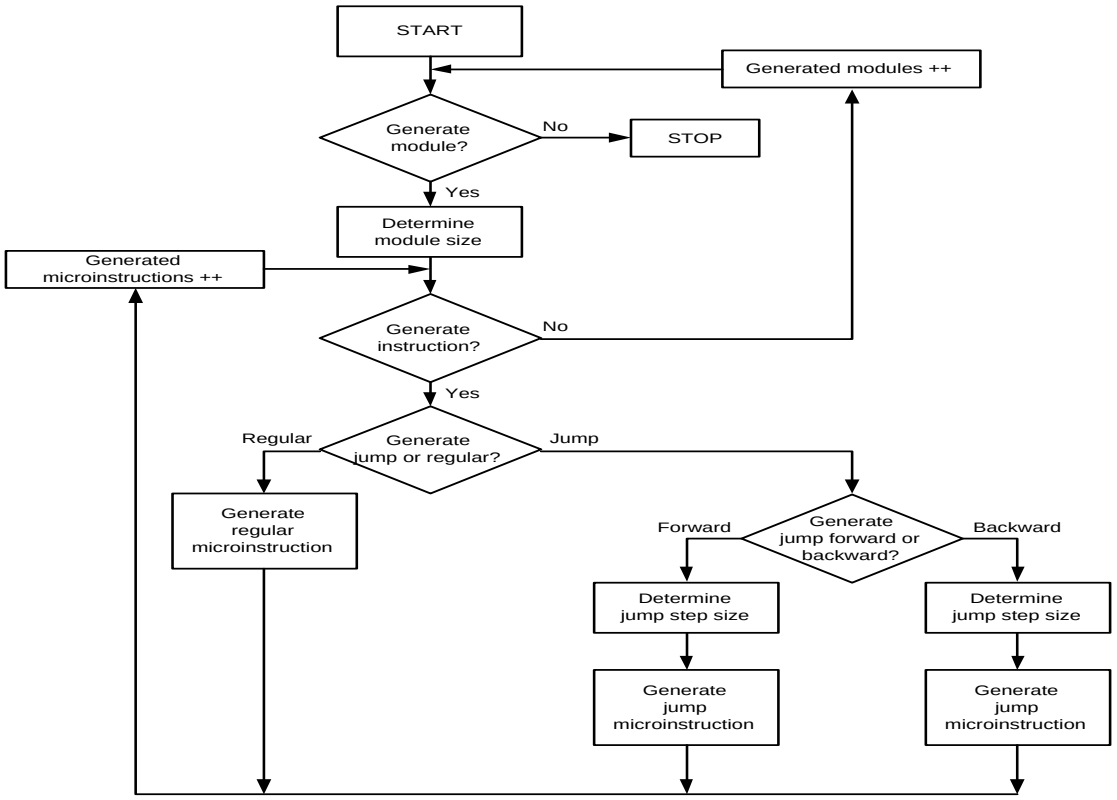

Fig. 5. The microcode generator flowchart.

placements[3]. Similarly, $\rho\mu$-code that contains many non-frequently used modules require a large pageable storage since a small pageable storage will again result in many replacements.

- **Frequency and type of module calls:** This characteristic mostly influences the caching scheme of the $\rho\mu$-code unit. When the execution of the $\rho\mu$-code contains a lot of calls from one module to another, it requires a more intelligent scheme for the loading and replacement of $\rho\mu$-code. Furthermore, it important to determine whether the calls are to already loaded modules or to new modules.

Having said this, we can determine several input parameters of the synthetic microcode generation tool. Regarding the first characteristic, we have to specify: the *number of modules*, the *range of module sizes*. Depending on how frequent the modules are called, we can determine their utilization frequency during execution. Another parameter that can be specified and that can have an influence on the storage size of the $\rho\mu$-code unit is the *microinstruction width*. Regarding the second characteristic, we have to introduce a distinction between regular microinstructions and call microinstructions (called jumps). This is done by specifying the *percentage of regular microinstructions* and the *percentage of jump microinstructions*. Furthermore, we specify the *range of jump steps*, because small jumps are likely to be within one module and thus not requiring the execution of another module. This in turn diminishes the need for loading another module. Finally, we specify the *ratio between forward and backward jumps*. Since backward jumps are more likely to call already loaded modules, this ratio influences the likelihood that new modules must be loaded (thus replacing already loaded ones).

[3]Having many replacements is detrimental to the overall performance since the loading of $\rho\mu$-code into the $\rho\mu$-code unit takes a considerable amount of time.

V. Software implementation

In this section, we describe the principle behind the software implementation of the microcode generation tool. We have to re-iterate that the tool's purpose is not to generate functional $\rho\mu$-code since its functional behavior does not affect the investigation into the design parameters of the $\rho\mu$-code unit. The input/output characteristic of the tools is depicted in Figure 6.

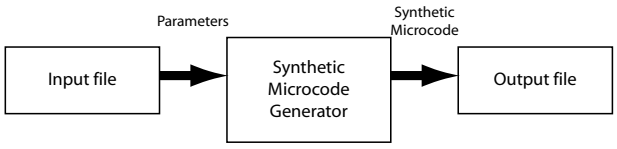

Fig. 6. Input/output characteristic of the benchmark generator.

The parameters determined in the previous section serve as the input to the microcode generator. Based on the given parameters (presented to the generator as a text-file), synthetic microcode is generated and stored in a text-file for further processing by, e.g., the $\rho\mu$-code unit simulator. The flowchart used to generate the $\rho\mu$-code is depicted in Figure 5. In this flowchart, a number of modules is generated until *number of modules* is reached. A module is created by generating a number of microinstructions that is equal to the module's size which is randomly determined based on the *range of module sizes*. Depending on *percentage of regular/jump microinstructions*, a similar ratio of regular and jump microinstructions are generated. For the jump instructions, the jump direction and step size are determined by *ratio between forward and backward jumps* and *range of jump steps*, respectively.

After the generation of the synthetic $\rho\mu$-code, the statistics of the generated $\rho\mu$-code is being gathered in order to reflect its actual characteristics. This is done due to the fact that a random number generator is used to determine the variables utilized in the blocks "Determine module size", "Determine jump step size", "Generate jump or regular?", and "Generate jump forward and backward". However, results have shown that the deviation from the input parameters is rather small.

VI. Conclusions and Future Work

In this paper, we highlighted that in the embedded processor design, the utilization of programmable processors is becoming more commonplace due to the possibility of re-using the processor core in subsequent design cycles. Such processor cores are augmented with specialized hardware (units) in order to increase the overall performance. In the past, such specialized units have been implemented in fixed hardware, e.g., application-specific integrated circuits, but we are witnessing a shift towards an implementation in reconfigurable hardware, e.g., field-programmable gate arrays (FPGAs). In this light, the MOLEN processor was introduced which tightly integrated support for such reconfigurable hardware by utilizing reconfigurable microcode ($\rho\mu$-code). In this paper, we have discussed the extended hardware/software co-design methodology that employs the $\rho\mu$-code. Consequently, the utilization of $\rho\mu$-code requires on-chip storage in order to improve the loading times of $\rho\mu$-code. For this purpose, the $\rho\mu$-code unit was introduced that incorporates fixed/permanent and pageable storage for frequently used and non-frequently $\rho\mu$-code, respectively. In order for the MOLEN processor to be taken up by the industry, several design parameters of the $\rho\mu$-code unit must be determined: organization and size of the fixed and pageable storages of the $\rho\mu$-code unit and caching schemes for the pageable storage. Due to the fact that real $\rho\mu$-code is hard to obtain, we opted to generate synthetic $\rho\mu$-code that exhibits the required characteristics in order to investigate the mentioned design parameters. An additional benefit is that a wide variety of microcode can be generated and thus allowing a much faster design space exploration of the $\rho\mu$-code unit. An additional advantage of this approach is that when real $\rho\mu$-code is available (after a hardware design) and its characteristics determined, we can reduce the design space based on the results of earlier design space exploration utilizing synthetic $\rho\mu$-code exhibiting similar characteristics. In this paper, we have shown what input parameters are needed for such a tool in order to generate $\rho\mu$-code that exhibits the required characteristics. Furthermore, we have shown a simplified way in how we have achieved this. Future work includes the development of a simulation tool for the $\rho\mu$-code unit that utilizes the generated $\rho\mu$-code as described in this paper.

References

[1] D. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling. Architecture Design of Reconfigurable Pipelined Datapaths. In *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, pages 23–40, March 1999.

[2] J. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the IEEE Symposium of Field-Programmable Custom Computing Machines*, pages 24–33, April 1997.

[3] R. Razdan and M. Smith. A High-Performance Microarchitecture with hardware-programmable Functional Units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–180, November 1994.

[4] S. Vassiliadis, S. Wong, and S. Cotofana. The MOLEN $\rho\mu$-coded Processor. In *Proceedings of the Conference on Field Programmable Logic 2001 (FPL2001)*, 2001.

[5] M. V. Wilkes. The Best Way to Design an Automatic Calculating Machine. In *Report of the Manchester University Computer Inaugural Conference*, pages 16–18, July 1951.

[6] R. Wittig and P. Chow. OneChip: An FPGA Processor with Reconfigurable Logic. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 126–135, April 1996.