

# Prototyping Pipelined Applications on a Heterogeneous FPGA Multiprocessor Virtual Platform

Antonino Tumeo<sup>1</sup> Marco Branca<sup>1</sup> Lorenzo Camerini<sup>1</sup> Marco Ceriani<sup>1</sup>  
 Matteo Monchiero<sup>2</sup> Gianluca Palermo<sup>1</sup> Fabrizio Ferrandi<sup>1</sup> Donatella Sciuto<sup>1</sup>

<sup>1</sup>Politecnico di Milano, Dipartimento di Elettronica e Informazione  
 {tumeo,gpalermo,ferrandi,sciuto}@elet.polimi.it

<sup>2</sup>HP Labs, Palo Alto  
 matteo.monchiero@hp.com

**Abstract**— Multiprocessors on a chip are the reality of these days. Semiconductor industry has recognized this approach as the most efficient in order to exploit chip resources, but the success of this paradigm heavily relies on the efficiency and widespread diffusion of parallel software. Among the many techniques to express the parallelism of applications, this paper focuses on pipelining, a technique well suited to data-intensive multimedia applications. We introduce a prototyping platform (FPGA-based) and a methodology for these applications. Our platform consists of a mix of standard and custom heterogeneous cores. We discuss several case studies, analyzing the interaction of the architecture and applications and we show that multimedia and telecommunication applications with unbalanced pipeline stages can be easily deployed. Our framework eases the development cycle and enables the developers to focus directly on the problems posed by the programming model in the direction of the implementation of a production system.

## I. INTRODUCTION

The recent advent of multicore processors strongly motivates the investigation of new programming models. Being able to easily write efficient parallel programs is indeed the key point that may determine the commercial success (or the failure) of these architectures [12]. Nevertheless, prototyping parallel applications is still a complicated and cumbersome process.

Pipelining is an appealing programming paradigm that allows to exploit the data-level parallelism of many applications. It is especially well suited to streaming applications, like audio/video compressors or data packet coding/decoding, i.e. the core applications of most embedded systems.

This paper presents a virtual platform (based on Field Programmable Gate Array - FPGA - technology) oriented at making easy the development and the analysis of these applications. FPGA based Systems-on-Chip (MPSoCs) are an appreciable platform for prototyping and sometimes even for final implementation of low volume, mission critical embedded systems. Several commercial toolchains offer easily connectable, pre-made components, and allow the design of production ready systems. Thanks to the ever increasing size of reconfigurable logic devices, such solutions are today becoming interesting for multiprocessor systems prototyping [2]. They allow early evaluation of the challenges posed by new programming paradigms, accelerating the development cycle of complex production systems.

Adapting an application to a pipeline model is not a trivial task. The stages of the application must be tuned to balance the utilization of each processing unit and to maximize the throughput. The performance of pipelined applications is heavily dependent on the communication mechanisms offered by the architecture. Furthermore, a correct implementation requires a detailed analysis of the communication patterns among the different computational kernels. Even the emerging partitioning tools that tries to automatically perform the

pipelined decomposition, requires feedback from a realistic embedded platform to evaluate if they operated correctly.

The main contributions of this paper can be summarized as follows:

1. A heterogeneous multiprocessor platform on FPGA, that integrates commercial-off-the-shelf (COTS) and ad hoc components. This platform is composed by soft and hard cores, connected together in a master/slave fashion and exclusively synchronized via interrupts.
2. A software layer that runs on top of this platform, allowing the decoupling of the different stages and the effective deployment of pipelined applications. This layer is sufficiently generic to be exploited by an automated exploration flow.
3. Several case studies of applications with unbalanced pipeline stages that run on this platform.

The paper proceeds with Section II which presents some related works. We then introduce the basic architecture of our platform in Section III and detail the kernel that manages the communication and synchronization mechanisms in Section IV. Section V describes the applications chosen as case studies, and, finally, Section VI concludes the paper.

## II. RELATED WORKS

A multitude of works proposed architectures and methods for pipelined applications [5], [7] both in stream-oriented fine-grain processors (like Stanford Imagine) and massive multiprocessors (StreamIt+RAW effort at MIT). This paper is not intended to contribute to the state-of-the-art of these architectures or compilers, but want to demonstrate a flexible framework for the validation of new ideas in the hardware and the software. Our objective is to investigate heterogeneous pipeline parallelism from the system level point of view, while these works address architectural aspects of homogeneous systems.

Several works discussing multiprocessor systems on FPGA have appeared. Among them, we cite the ones from Clark *et al.* [4] and James-Roxby *et al.* [10], who present shared memory multiprocessor case studies with *ad hoc* mechanisms for synchronization implemented with the Xilinx toolchain. A related approach has recently been proposed by Tumeo *et al.* [15], who use a FPGA multiprocessor to prototype and evaluate real-time scheduling algorithms and applications. The Altera toolchain has also been exploited to evaluate solutions to the problems posed by the multiprocessor SoC approach, with works focused on synchronization [6] and cache coherency [9]. However, these solutions are more limited in scope, since they do not present a generalized support for streaming or pipelining in terms of hardware mechanisms and software layer, but rather focus only on (specific) data parallel applications. Recently, Gaisler Research AB has presented LEON 3 [1], a synthesizable VHDL model of a SPARC V8 compliant architecture with full support for symmetric



This section will describe in detail the main features of this micro-kernel which, thanks to its high flexibility, can be customized by the developer to target specific pipelined applications.

### A. Micro-kernel setup

Several steps are performed to configure the architecture and obtain the desired behavior for the target applications.

1. *Micro-kernel parameters setting.* The developer determines which elements and features of the virtual platform are used by the target application. First of all, he decides the number and the sequencing of the workers. He selects, for each stage of the pipeline, if a specific implementation of the MicroBlaze or a PowerPC should be used. Then, he specifies the granularity of the data which the micro-kernel will distribute among the processors. The input data set will be partitioned taking this value as the basic block for all the computations and data movements. Each computational kernel will accept as input a block of these dimensions, will operate on it and will generate a transfer request (through interrupt) to the master processor as soon as an output data block of such dimensions has been produced.

2. *Data structures definition.* The developer determines input and output data structures for each active worker. The micro-kernel allows specifying different types for input and output structures, depending on the requirements of the stages mapped to each processors. The data streamed to each processing element can be expanded or reduced, and the kernel will take care of the type conversion among the different stages of the pipelined application.

3. *Data flow setting.* The developer defines the actual data flow of the application. Maximum flexibility for data distribution is allowed. Workers can share the same input data or work on different data sets, depending on the pipeline model of the application. Workers can execute the same kernel, and thus distribute among themselves input data from the same streams, read streams generated by another processing element or read different streams from shared memory.

### B. Pipeline management

In our Virtual Platform, the decoupling of the streaming kernels is obtained by simply allocating portions of the external memory for the locations that store the data to be passed among the different processing elements. Depending on the application, it is possible to decide from which locations processors read the input values and where they output the produced results. This allows broad flexibility in organizing how the streams move along the multiprocessor: processors can read from the same location (i.e. they can share a input stream), or one can read the output of the other (i.e. an output stream is the input stream of another processing element). They can even reuse the input location as the output location, and queues of different dimensions to store the streams can be easily defined. Each worker processor uses a local buffer, in which data are moved from the shared queues for processing. Local buffers can have 1 or 2 slots for data, depending on the local memory available in each worker, of equal size. Local dual buffers are useful to reduce the stalls, overlapping communication and computation as much as possible, and hiding the latencies due to the sequencing of operations through interrupts.

The master processor sequences the execution flow through interrupts. Data are moved from the external memory to the local buffers of each worker depending on the throughput of its predecessor in the pipeline. When a worker finishes to compute a buffer, an interrupt is generated and the master starts the DMA transfer. The DMA then signals when results from a stage are saved in external memory and new data are loaded in the worker. The master can thus start a worker when its new data are ready, launching an interrupt signal towards it.

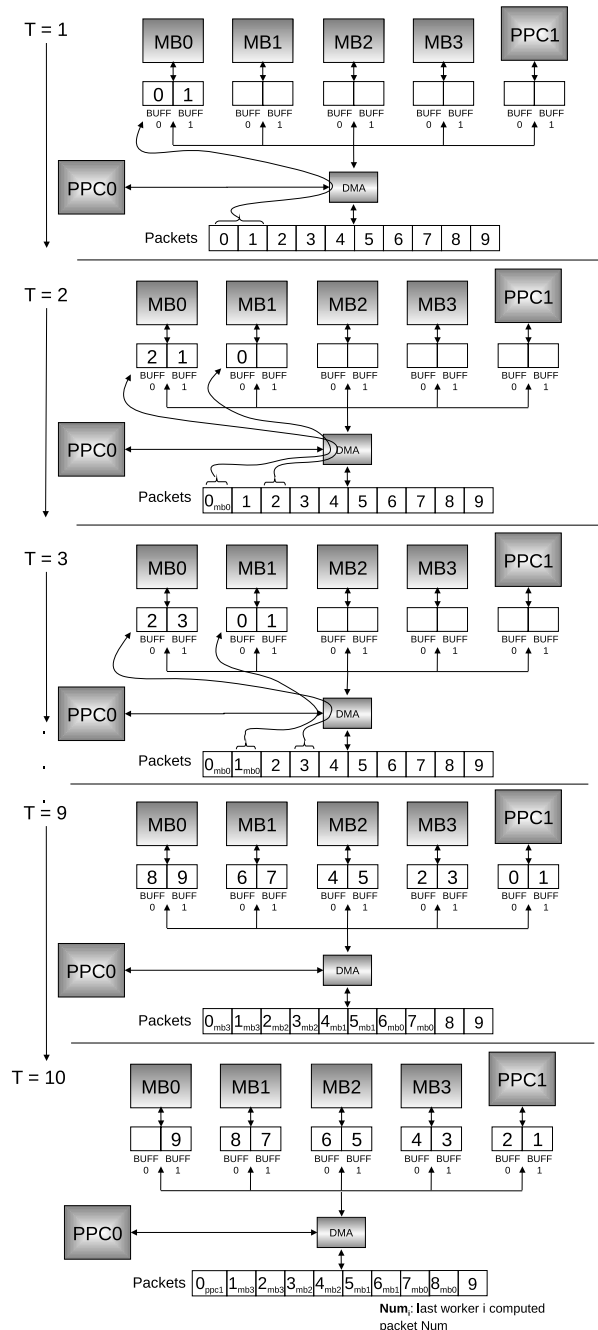


Fig. 2. Pipeline management example

Figure 2 shows a simple example on how communication and computation can be managed in our architecture. In this example, we suppose that each worker performs a different stage of the pipeline, and the last worker is a PowerPC. Furthermore, pipelined computation is managed through a global queue, which also acts as a commit buffer for each stage of the pipeline. At time 1, the master processor sends the data from the first two slots of the global queue to the first worker in the pipeline. When the interrupt signaling the termination of the first DMA transfer reaches the master, processing on the worker is started. Then, transfer of block 1 is performed. When computation on block 0 has finished (time 2), the worker generates an interrupt and the master moves data block 0 back to the

global queue. The output data from the first stage of the application replaces the old input data, both in the local memory and, after the DMA transfer, in the global queue. At the same time, if the second worker in the pipeline has free slots, data are moved into them until they are all filled or no more ready data are available in the local queue. A mechanism like this can manage situations in which a processing element is faster than the next one: data are simply committed back to the global memory, and sent to the next worker only if it signals, through interrupt, termination of the computation on its own previous block. This model allows to reduce memory occupation, since for each stage the same locations are reused, but can be limiting when the pipelining involves more than one processing element per stage.

### C. Micro-kernel primitives

Data flow from and to the worker processors is controlled by some simple primitives executed on the master. We briefly describe them here.

*VP\_send*. This function is used by the master processor to move data blocks from the storage locations in external memory to buffers of any of the workers, specified through a parameter. If dual slot buffers are enabled on the workers, it can be used to fill both the buffers.

*VP\_receive*. This function is used to move data blocks from local buffers of any workers, specified by a parameter, to the storage locations in external memory.

*VP\_receive\_and\_send*. This function is invoked by the master when a worker signals that a block of data has been produced and it is ready to be transferred back to a location in external memory. At the same time, it checks if new data are available for the worker from the input stream and if so, it sends them to it. If dual slot local buffers are used, it checks to and from which buffer the transfer should be performed. If the worker has completed its computation, and no new data are available, it simply returns the control to the master processor.

*Pipeline*. This function provides the main mechanism to enable pipelining. It keeps trace of the status of each worker and synchronizes the data flow using all the functions described above. When a worker has produced a block of data, it forwards an interrupt to the master processor. The master executes a receive, saves the output stream in the external memory, and sends a new input stream to the worker, which is then restarted with another interrupt. After the buffers of each element of the pipeline have been filled with the *VP\_send*, *VP\_receive\_and\_send* can be used to coordinate the data flow. *VP\_receive* is used to flush the pipeline.

Note that the simple DMA component integrated in the architecture does not support multiple outstanding transfers in hardware. However, we can decouple master operations from DMA transfers through a software transfer list supported by interrupt mechanisms. Send and receive requests are queued in a circular buffer. When the DMA finishes a transfer, an interrupt is generated and, if other transfers are queued in the circular buffer, the next is launched before leaving the interrupt routine.

## V. CASE STUDIES

The goal of this section is to show how our framework can be used. We chose three sample applications taken from the MiBench [8] suite, and using our platform we designed their pipelined version. The approach followed is the same for all these applications, but this same process took us to very different solutions from the point of view of the data structures, communication and synchronization mechanisms exploited. Nevertheless, the framework is sufficiently flexible to support all the requirements.

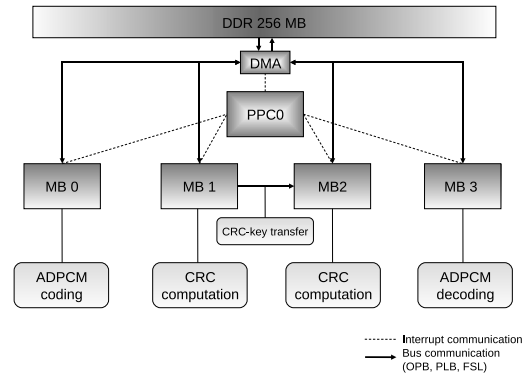


Fig. 3. ADPCM - CRC mapping on the Virtual Platform

The main phases of our approach can be summarized as follows:

*Application profiling*. This phase aims at providing to the designer information about the sections of the application that require the biggest computational effort to the processors. It allows the designer to recognize the critical points of the applications and to map the different sections identified to the various computational units of the heterogeneous architecture.

*Identification of the pipeline stages*. This phase provides a partition of the application in single tasks to be assigned to the stages of the pipeline. The main purpose of this design step consists in finding the optimal balance of the application in terms of computational effort.

*Implementation of the pipeline stages*. This step consists of rewriting the application, taking into consideration data dependences, to obtain the desired data flow.

*Experiments and validation*. Finally, we put the redesigned application and the kernel of the platform together, to evaluate if the pipelining has been correctly modeled. This is the most important aspect of our work from the point of view of the designer. In fact, he will be able to test his code on a working platform that can be adapted to several circumstances thanks to its hardware potential and to the generality of the kernel running on it.

### A. ADPCM - CRC

ADPCM (Adaptive Differential Pulse Code Modulation) is an extension to the standard PCM coding for audio signals, widely used in telecommunication techniques. ADPCM predicts the value of a sample starting from the values of preceding samples, and then transmits the error (difference) of this prediction with respect to the real value of the sample. The receiver makes the same prediction and then sums to it the received value. ADPCM on the transmitter can be followed by a Cyclic Redundancy Check (CRC), which in turn can be used to validate the data received. The complete application is composed as follows:

1. *Transmitter* ADPCM coding on a set of PCM samples
2. *Transmitter* CRC algorithm on the ADPCM coded samples
3. *Receiver* CRC algorithm on the ADPCM coded samples
4. *Receiver* ADPCM decoding, to obtain the initial PCM samples back

PPC0 acts as master of the platform managing all transfers from and to external memory, while only the MicroBlazes are used as slave processors. MB0 and MB3 respectively execute the ADPCM coding and encoding, while MB1 and MB2 calculate the CRC. The CRC key is transferred from MB1 to MB2 through the FSL connection.

Figure 3 shows how the application is mapped to the architecture. ADPCM-CRC is very suitable for pipelining, since all the phases can

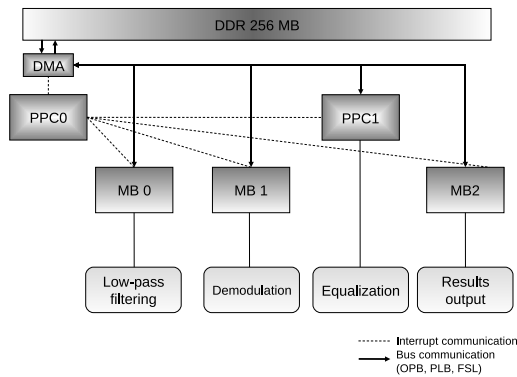


Fig. 4. Communication system and data flow for the Frequency Modulation (FM) case study.

be run on independent blocks. The pipeline is filled after four steps and the advantages of pipelined execution become more evident as the size of the input stream increases, allowing the workers of the platform to remain active for a larger percentage of time.

To maximize the quantity of data transferred at each step, we redesigned the algorithm and made it working on packets of samples with the maximum dimension allowed by the architecture (16 KB). Furthermore, dual buffers in local memories are used to hide as much as possible the communication latency. We devised a data structure able to store both the input and the output values, like the one presented in the example of Figure 2, making the data transfers as regular as possible. At each interrupt event, PPC0 either transfers a block of data from the external memory to the local memory of a worker, or transfers an entire block of data from the local to the external memory, rewriting the old input data with the new results from each stage.

### B. Frequency Modulation

This benchmark consists of four distinct phases: *Low-pass filtering*, *Demodulation*, *Equalization* and *Result writing*. These four phases, even if logically different, are not balanced. Furthermore, data transfers from the external memory have not a regular pattern like ADPCM-CRC. Profiling also shows that equalization is the slowest stage.

Figure 4 shows how the Virtual Platform is used in this case study. The profiling showed that the computational effort required by the equalizer and the data structures used during this phase were not compatible with both computational capabilities and local memory constraints of the MicroBlaze. This fact suggested to map this phase on the second PPC of the platform. Furthermore, we designed the data structures used by the workers and the communication and synchronization mechanisms in order to minimize the number of data transfers from/to central memory as described below.

The input value elaborated at each iteration in a single step of the pipeline is represented by a floating point number. This datum can be discarded in the successive stage of the pipeline. In this way, once the data has been transferred from the external memory to the local memory of one of the workers, it can read it and overwrite it with the result of the computation. Such a method allows to transfer as many data as the local memories of the workers can store, without reserving additional space.

In this case study, the communication and synchronization mechanisms mostly used are the interrupts. The master has to account for the number of bytes returned by each worker. This is because of the asymmetry between the input and output data size of the stages

of Frequency Modulation (FM). Once a stage of the pipeline has produced enough data to allow the master to fill a buffer of the subsequent worker in the pipeline, it asks for a data transfer to the DMA.

### C. JPEG

For this case study, we implemented a streaming version of the JPEG standard baseline compression algorithm. The computation starts with the loading of a .PPM image and terminates with the generation of a compressed image in the JFIF format. The algorithm, when applied to color images, is composed by 5 stages:

1. RGB to YCbCr color-space conversion. A pixel, represented by three 8-bit Red, Green and Blue values, is converted to other three 8-bit values that represent Luminance (Y) and differential Chrominance (Cb and Cr).
2. Chrominance channels downsampling: the chrominance is horizontally and vertically subsampled with a factor of 2, leaving 2 values of chrominance, 1 per channel, for each 4 values of luminance, with a 4:2:1 ratio.
3. 2D-Discrete Cosine Transform (2D-DCT): executes the transformation to the frequency domain on blocks of 8x8 component (Y, Cb or Cr) values.
4. Quantization: each component value in the frequency domain is divided by a constant.
5. Entropy coding: 8x8 pixels blocks are organized following a "zigzag" ordering and are then coded with the RLE and Huffman algorithms.

All these steps, except for the entropy coding, can be independently performed on blocks of at least 8x8 pixels each. In the fifth step, the first value of each block of 64 pixels is represented as the difference from the previous one, so the only constraint is to scan the image in the right order. It is therefore straightforward to pipeline the application assigning each step to each processor, as illustrated in Figure 5. Phases 1 to 4 are well balanced, but phases 5 is heavier, and it is thus assigned to the fastest slave processor, PPC1.

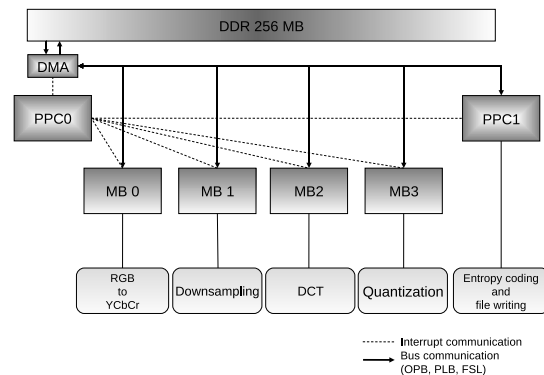


Fig. 5. JPEG decomposition

To allow easier and more efficient pipelization, the starting image is partitioned in blocks of 16x16 pixels, so the downsampling can generate a full block of 8x8 Y, Cb and Cr components, and three blocks composed of only 8x8 Y components. These are moved to the DCT, which can then transform Y, Cb and Cr channels for the first block and only the Y channels for the remainings. Workers operate and exchange data using structures of 16x16x3 (768) bytes, which are filled as required by each phase.

TABLE I  
PERFORMANCE (THROUGHPUT) FOR ALL APPLICATIONS AND PIPELINE STAGES.

Compute Element	ADPCM-CRC	FM	JPEG
MB0	1.3947	0.1929	0.2716
MB1	0.0524	0.1715	0.2617
MB2	0.0712	3.2498	0.2629
MB3	0.0810	-	0.2210
PPC1	-	0.0005	0.0192
TOTAL [MB/s]	0.052	0.0005	0.0188

#### D. Results

Table I shows the results we obtained on our platform. We detail the throughput of each application and each compute element. Note that a compute element is mapped to a given pipeline stage. ADPCM-CRC takes advantage of the pipelining because of the even stages, JPEG has a relatively slow stage while FM has a very slow stage that, even if mapped on the fastest processing element available (PPC1), dominates the performance. Our goal is not to show a high performance solution for these applications, but to illustrate the usefulness of our framework. Our solution can be considered as the middle point between *Software-in-the-loop* (i.e., the use of a software simulator of the target architecture to validate the applications, which is cheap and configurable, but slow) and *Hardware-in-the-loop* (i.e., the use of the real target hardware, modified for the testing, to verify the applications, which is fast, but expensive and not flexible). In fact, the developer can use our platform, after enabling the required number of processing elements, to partition an application and obtain a working pipelined solution. This solution can then be refined on the prototyping platform, until all the stages are almost similar in performance, and then moved on the final, higher performance, target architecture. The developer will retain the partitioning, the data flow and the now validated new algorithm design that supports pipelining, without requiring substantial rewriting.

Furthermore, our platform can easily be used to perform design space exploration for pipelined applications. The communication primitives that manage the streaming of the data among the processing elements and the shared memory are independent from the architecture of the processors executing the code. The developer, or an automated design flow, simply has to decide the sequencing (single or multiple parallel stages) and the mapping (target processing elements) of the tasks in the pipeline to meet the required performance. It is then easy to insert the primitives and use our faster *hardware platform* to evaluate which is the most favorable pipeline model for the application, without resorting to a slow software simulator. If the required performance of the application is not reached, it is then possible to perform a new mapping, augment a processing element with ad hoc accelerators exploiting the programmability of the FPGA (e.g. hardware implementations of frequency transforms) or, finally, change the task partitioning of the application. The use of reconfigurable logic even enables the easy integration of hardware probes to monitor specific parts of the architecture.

We want to remark that our architecture has been assembled mainly using standard COTS, and complex communication and synchronization sequencing mechanisms have been implemented using only the simple means of interrupt signals: this approach is easily reproducible and adaptable to other FPGA soft cores or ASIC multiprocessors.

#### VI. CONCLUSIONS

We presented a heterogeneous multiprocessor FPGA virtual platform for prototyping pipelined applications. FPGA multiprocessor

prototypes allow validation of software applications in reduced times with respect to software simulators, and can easily be adapted to different classes of applications. We showed how some multimedia applications can be ported and converted to a pipeline programming model and then validated using our framework. Our results show that two of these applications can be successfully pipelined, while one (Frequency Modulation) has a bottleneck in one of the stages dominating the performance.

We believe that this work is an important step in the area of fast prototyping, since it proposes a flexible and customizable platform that can be very useful to embedded system designers and researchers. It is the first proposal of a general multiprocessor-on-FPGA platform targeted to pipelined applications. Our future works are directed towards the extension of the framework to account for a variety of architectures and programming models, in order to build one of the first comprehensive fast prototyping platform on FPGA.

#### ACKNOWLEDGMENTS

Research partially funded by the European Community's Sixth Framework Programme, hArtes Project.

#### REFERENCES

- [1] Leon3 Processor. available at <http://www.gaisler.com>.
- [2] Research Accelerator for Multiple Processors (RAMP), <http://ramp.eecs.berkeley.edu/>.
- [3] P. Bonnot, F. Lemonnier, G. Gaillat, O. Ruch, G. Edelin, and P. Gauget. Definition and SIMD Implementation of a Multi-Processing Architecture Approach on FPGA. In *DATE '08: Design Automation and Test in Europe*, pages 610–615, March 2008.
- [4] C. R. Clark, R. Nathuji, and H.-H. S. Lee. Using an fpga as a prototyping platform for multi-core processor applications. In *WARFP-2005: Workshop on Architecture Research using FPGA Platforms*, February 2005.
- [5] W. J. Dally, U. J. Kapasi, B. Khailany, J. H. Ahn, and A. Das. Stream processors: Programmability and efficiency. *Queue*, 2(1):52–62, 2004.
- [6] P. Gai, G. Lipari, M. Di Natale, M. Duranti, and A. Ferrari. Support for multiprocessor synchronization and resource sharing in system-on-programmable chips with softcores. In *IEEE International SOC Conference*, pages 109–110, September 2005.
- [7] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, October 2006.
- [8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC-4: Proceedings of the 4th IEEE Annual Workshop on Workload Characterization*, pages 3–14, December 2001.
- [9] A. Hung, W. Bishop, and A. Kennings. Symmetric multiprocessing on programmable chips made easy. *DATE '05: Design, Automation and Test in Europe*, pages 240–245, March 2005.
- [10] P. James-Roxby, P. Schumacher, and C. Ross. A single program multiple data parallel processing platform for FPGAs. *FCCM 2004: 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 302–303, April 2004.
- [11] R. K. Karanam, A. Ravindran, and A. Mukherjee. A stream chip multiprocessor for bioinformatics. In *dasCMP '07: Workshop on Design, Architecture and Simulation of Chip Multi-Processors*, December 2007.
- [12] C. O'Hanlon. A conversation with John Hennessy and David Patterson. *Queue*, 4(10):14–22, 2007.
- [13] K. Ravindran, N. Satish, Y. Jin, and K. Keutzer. An fpga-based soft multiprocessor system for ipv4 packet forwarding. pages 487–492, August 2005.
- [14] S. L. Shee and S. Parameswaran. Design methodology for pipelined heterogeneous multiprocessor system. In *DAC '07: 44th annual Conference on Design Automation*, pages 811–816, July 2007.
- [15] A. Tumeo, M. Branca, L. Camerini, M. Ceriani, M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto. A dual-priority real-time multiprocessor system on fpga for automotive applications. In *DATE '08: Design Automation and Test in Europe*, pages 1039–1044, March 2008.