Reconfigurable General-purpose Processor Idea Overview

Igor Zarzycki

***Abstract*—This paper presents the idea of the reconfigurable general-purpose processor implemented as dynamically reconfigurable FPGA (called "reconfigurable processor" in the rest of this document). Propesed solution is compared with currently available general-purpose processors performing instructions sequentially (called "sequential processors" in the rest of this paper). This document presents the idea of such reconfigurable processor and its operation without going into implementation details and technological limitations.**

The main novelty of reconfigurable processor lays in lack of typical for other processors sequential execution of instructions. All operations (if only possible) are executed in parallel, in hardware also at subistruction level. Solution proposed in this paper should give speed up and lower power consumption in comparison with other processors currently available. Additionally proposed architecture does not requires neither any modifications in source codes of already existing, portable programs nor any changes in development process. All of the changes can be performed by compiler at the stage of compilation.

***Index Terms*—processor; FPGA; dynamic reconfiguration; reconfigurable computing paradigm**

I. State of the art

A. Sequential processors

SINCE few years there was no large breakthrough in performance of sequential processors. Such approach to processor architecture appears to reach its limits. The maximal frequency of their clocks is at the level of about 3GHz and stopped increasing. The overall improvement in performance is gained by minor changes in architecture (like size extension of branch prediction tables), enlarging cash memory or by increasing the number of cores.

Unfortunately the performance improvement causes increase of power consumption. For example Opteron 6386 SE[1] (released on 5th of November 2012) made in 32nm technology is having 16 cores with 2.8GHz clock rate, 16MB of L3 memory (2 * 8MB) and 140W of TDP (Thermal Design Power). Opteron 6274[1] (released on 14th of November 2011) made also in 32nm technology similarly with 16 cores and the the same amount of L3 cash memory but working with 2.2GHz and having 115W of TDP. Comparison of those two processors produced by AMD shows that power consumption increased about 30% (within one year of development).

Currently (November 2012) the fastest supercomputer (Titan located at Oak Ridge National Laboratory) is using 18688 Opteron 6274 processors and achieved 17590.0 TFlop/s of computation power (theoretical peak 27112.5 TFlop/s) and over 8MW of power consumption[2].

In sequential processors there is no possibility for parallel execution within single core (excluding Hyper-Threading Technology by Intel where one physical core may behave as two virtual processors - each of them with sequential processor limitations). Hence the need to increase the number of cores. Dual-core and multi-core processors are constantly enlarging their market share. Even many single-core processors are produced as dual-core (or generally multi-core) ones with one (or more) of the cores not active. Similar trend can be observed even for mobile devices market although low power consumption is crucial in their case while modern architectures are constantly increasing it.

B. Computation accelerators

To increase the performance computation accelerators offering massively parallel execution are being used. However they are mainly based on sequential processors possessing extremely large number of cores and hence providing massively parallel execution of calculations.

For example currently (November 2012) the fastest supercomputer is using 18688 Tesla K20X computation accelerators (one per each CPU processor)[2]. Tesla K20X computation[1] accelerator is having GK110 sequential processor with 2688 cores working at the frequency of 732MHz. Its maximal power consumption (including memory and other elements of board) is 235W. To justify such enormous power consumption Tesla K20X is supposed to provide[1] maximally 1.31 TFlop/s of computation power using double precision and 3.95 TFlop/s at single precision calculations.

Unfortunately computation accelerators such as mentioned Tesla K20X are requiring software to be written in special way enabling usage of their abilities. Hopefully such software often can run without computation acceleration on normal sequential processor. Therefor programs using OpenCL or other APIs (Application Programming Interfaces) allowing usage of computation accelerators are being more and more popular.

C. Software requirements

Used software is constantly increasing its demands on computation power. Not only because of higher complexity of problems being solved but unfortunately often also because

I. Zarzycki is with Department of Microelectronics and Computer Science (DMCS), Lodz University of Technology, Łódź, Poland (e-mail: izarzycki@dmcs.pl)

[1]Data taken from manufacturer's data shits.
[2]Data taken from TOP500 list (November 2012).

of not optimal code. To increase software efficiency currently there is trend for parallel execution.

Many tasks like graphical computations, database operations or scientific computations can be divided into large number of sub tasks being independent on each other and hence being suitable for parallel execution. Modern compilers are capable to create programs to be in total or partially executed by arbitrary number of threads. This approach urges us to increase abilities of parallel execution.

D. Reconfigurable devices

On the other hand reconfigurable devices are commonly used for specialized ([1]) or scientific calculations ([2]), often as coprocessors ([3]). Reconfigurable hardware can be easily adapted to fulfill nearly any application demands by changing its functionality.

Additionally often some of the sequential processors architectures are being simulated in hardware of reconfigurable devices. This allows to test and switch between different architectures choosing the most suitable one for particular problem being solved and find optimal solution without need of physical implementation of all available solutions. This maybe implemented already at the stage of prototyping.

Hence dynamic reconfiguration (during run of the device) is supported in commercially available FPGAs (Field Programmable Gate Arrays - most common reconfigurable device type), several different concepts for usage them as acceleration of computations have appeared ([3]). Firstly used as standalone coprocessors, later (thanks to technology development) were equipped witch sequential processors cores combined with reconfigurable resources.

Moreover reconfigurable devices can operate as SIMP (Single Instruction Multiple Data) devices as described in [1] or [2]. Such approach significantly increase the parallelism of execution and hence improve performance. Another advantage of reconfigurable devices in comparison with sequential processors is low power consumption per operation.

E. Reconfigurable computing paradigm

Reconfigurable computing is computing paradigm that fills the gap between software and hardware. In [4] there is presented a Unix-like system, slightly modified to run on PowerPC core being part of Virtex. This allows programmers to treat tasks implemented in hardware in the same way as processes being run by sequential processor core. However creating new software requires low-level approach being far from modern programming trends. Even software being portable for other platforms can not be simply ported and requires to be rewritten. All this being far from modern approach to software creation strongly reduces usefulness of this solution and hence stops its development.

Since reconfigurable devices requiring description in some HDL (Hardware Description Language) are becoming more and more commonly used there have been created compilers being able to compile some common programming languages describing software to HDLs. For example compilation of C to Verilog. However such compilers are strongly limited because currently available reconfigurable devices are not supposed to work as non sequential processors. Because of that such elements of language like pointers are not supported. Those solutions assume rather compilation of simple functions describing elements like hash algorithms, scientific calculations or coding/decoding simple formats. There is no possibility for them to compile whole programs. Lack of compilers is a huge limitation for reconfigurable computing paradigm disabling its development.

F. The purpose to create reconfigurable processor

The aim presented in this paper is to create processor that could be used as general purpose one and would provide all benefits of reconfigurable computing. It should be conceptually compatible (not necessarily with particular type of socket or similar physical elements cause those parameters are varying even among sequential processors) with modern computers and operating systems architectures. Such processor could be used as CPU, GPU or other type of processor in computers or mobile devices to increase their performance and hence satisfy growing requirements put by software.

Additionally thanks to lower power consumption of reconfigurable devices with comparison to sequential processors, reconfigurable processor would be a good solution for mobile devices. Mainly because for them low power consumption is crucial parameter. High performance achieved thanks to massively parallel execution is another important advantage making reconfigurable processor a perfect, widely applicable solution.

II. General description of idea

A. Lack of sequential computing

Although ASIPs (Application Specific Instruction set Processors) are currently being actively researched, reconfigurable processor described in this paper is assumed to operate as general purpose one. Hence it should be able to perform any kind of application not only some of them solving only on some particular problems. Additionally there is assumed no sequential instruction processing hence there should be no sequential processor core implemented internally nor simulation of such in hardware.

Reconfigurable processor should be implemented as particular kind of FPGA. Such FPGA should be able to operate as typical processor. This means reading or writing data from address, responding to hardware interrupts, etc. The only exception from traditional sequential processor operation is that reconfigurable one is taking bitsreams instead of instructions. Such approach would make it compatible with currently available computers and operating systems architectures and hence make it usable.

B. Partitions

The concept is to generate partitions solving particular, simple tasks from which program is composed and switch them in the run-time if needed. Such partitions should be responsible for solving program or in most cases its parts.

Programs should be divided into several partitions. This is because complexity of average program make it unable to be solved within single partition.

Partitions should be loaded on demand or when their execution is approaching. Of course during compilation often it can not be clearly predicted which partitions should follow the active one and there are more candidates than one. In such case it may be useful to load all candidates (if it is only possible) before one will be selected. If loading all candidates is not possible, one (or some of them) should be selected according to some brunch prediction mechanism (or other mechanism making the decision). Such approach is supposed to load next partition during the execution of the previous one and hence to hide the time necessary for configuration.

Division into partition changes the program from a list of instructions to graph of partitions responsible for performing its parts. This concept of working applies also to such elements as operating system, drivers, shared libraries or software interrupt handlers. Under those conditions reconfigurable processor can be considered as Extreme RISC (Reduced Instruction Set Computing) processor being able to perform only one instruction — load and execute partition.

C. *Large parallelism*

This approach enables not only parallel execution of large number of threads but also huge parallelism of operations within single thread if only there are no dependencies between them. This may give speed up even if compared with massively parallel sequential processors currently available like modern GPGPUs (General Purpose Graphic Processing Units) or other computation accelerators being able to perform hundreds or even thousands of operations in parallel.

In proposed concept the only limitation of the number of partitions being independently executed in parallel is the amount of resources. The reasonable amount would be about thousands of cells from which partitions may be composed.

D. *Configuration time hiding*

The execution of program can be divided to configuration and execution of its partitions. For traditional sequential processors configuration time does not exist hence only execution time determines their performance. In dynamically reconfigurable processor time spent on configuration have also influence on performance and should be hidden to maximally reduce the latency between executions of partitions.

The main approach to hide configuration time is to configure one partition when other is being executed. However in such case configuration time should be lower or at least of the same order as execution time.

Currently available FPGAs are equipped with complex cells requiring several kilobits for configuration. Although each cell has high flexibility its configuration is far longer than execution of partitions. Despite that flexibility, a large part of cell functionality would not be used extending only the amount of data necessary to configure cell, enlonging configuration time. According to initial estimations in currently available FPGAs configuration would be from about 10 to 100 times longer than execution.

Because of that in architecture of reconfigurable processor proposed in this paper the amount of data necessary to configure single cell should be strongly reduced and be of order of tens to one hundred bits. Although average partition would use larger number of cells the total time required for its configuration should be on average lower than time spent on execution. In worst case configuration time is of the same order as execution time.

This approach should not increase execution time. Mainly because although average partition is composed of larger number of cell, cell is smaller and hence physical size of partition remains similar. Average register to register delay may remain similar or even smaller thanks to smaller fan-out. Register to register delays may be considered as the only ones influenting partition operation clock and due to it the execution time.

E. *Partitions management*

At the moment when particular partition ended its operation and is no longer necessary its resources should be possible to be utilised by other partitions. Of course it may be useful for some partitions to remain after end of their calculations. Small partitions such as loop bodies or some others frequently executed in program should remain after end of one their execution, so they could be rerun with only minimal configuration. Such approach may strongly reduce overall time spent by program on configuration without execution of any partitions.

To manage the partitions control unit is necessary. Control unit should be able to load partitions and provide basic control. Configuration, freezing/resorting partitions as well as receiving basic signals (like end of operation) must be provided. Advanced partitions management may enlarge and slow down control unit which is crucial for whole circuit. Because of that such advanced logic like finding out if particular partition should remain and not be utilised after end of its operation, should be determined by the compiler during compilation stage not by the control unit.

More advanced partition management algorithms including such elements like branch prediction mechanisms or others may also strongly influence the overall performance. Extending control unit may increase performance without lose of backward compatibility. However this should be empirically checked.

F. *Memory accessing*

Partitions themselves should not be directly responsible for memory accessing. There should be separate unit (later in this document called “memory accessing unit”) responsible for this task taking into account memory virtualization and other aspects. The partition provides data or place for data, address, control bits and calls for memory access. Control bits are responsible for representing access type (read or write), amount of data (word, byte etc.) and signalising data being ready. Memory accessing unit receives call and provides access according to access type control flag. After a successful data read or write data ready this flag should be down. It signalise for partition that data was written or read and automatically does not mark another fake call for memory access. In case of all access types rising data ready control flag by partition generates call for memory access.

Each memory call should be processed in parallel to others by several independent subunits hence one call do not have to wait for another to be processed. Otherwise achieved parallel execution within partitions could be spoiled be memory accessing unit. Additionally memory accessing unit should work with higher frequency than average partition. This unit performance is important for all partitions. Hence it is realised by dedicated circuit, it can operate faster than any partition. This allows to maximally reduce time necessary to process memory access algorithm and hence partition waiting time.

During configuration of the partition additional data is sent to memory accessing unit. This data contains detailed information like which cell may call memory access (to avoid fake calls), detailed access type (like direct, indirect or others), priority of the call and more.

G. *Floating point calculations*

Floating point calculations are crucial for different type of operations like scientific calculations, graphic processing, gaming or similar. Often speed of their execution is being considered as measure of performance. There for efficient implementation of those operation is important and can not be neglected.

Implementing floating point calculations in software within partitions would make them very large, complex and slow. Equipping cells with floating point units would significantly enlarge cell and hence spoil the performance. Because of that there should be several FPUs (Floating Point Units) working independently in parallel to each other. FPUs should be separate from cell matrix. Each partition would be able to call their functions via memory accessing unit or similar mechanism. This solution allows for fast and simple floating points operations without huge, complex partitions and without additional communication mechanisms extending size of cell.

The call for floating point operation could be received by simple manager sending it to free FPU. Although there would be additional manager, because of its low complexity it would not take much space nor consumes large amount of power. It also would be able to work with high speed and hence to be transparent for communication between partitions and FPUs. Of course partition may implement floating point operation realised in software and not be forced to use FPUs.

H. *Overall gain*

Thanks to better adaptation to solved problem and higher parallelism than in traditional sequential processors, gained speed-up should significantly reduce execution time and hence increase the performance even including configuration time that was not possible to be hidden.

Additionally reconfigurable processor in comparison with sequential one should have lower power consumption. Even including additional elements, the overall power consumption at maximal work load should be still lower than for currently available solutions. Combining high performance with low power consumption can make the reconfigurable processor perfect solution especially for mobile devices.

III. Hardware problems and solutions

A. *Cell architecture*

Modern FPGAs often assumes that bitwise operations are dominating. This is proper approach when building any kind of state machines used by control logic. However programs mostly perform arithmetic calculations and operate on whole words rather than single bits.

Additionally, currently available FPGAs try to give maximal flexibility. This approach significantly extends the physical size of cell and makes their configuration far more complex than it should be. Complex configuration gives long configuration time, that may be hard or even impossible to be hidden, especially with larger number of partitions following the one being currently executed. This may significantly spoil the performance.

Those are some of the problems forcing new FPGA architecture to be created for purpose of reconfigurable processor. Each cell should be designed to mostly operate on words, but also to be able to operate on single bits for partition control logic. Each cell may contain more complex dedicated elements like multipliers or similar however this may enlarge cell and as it was mentioned spoil the overall performance of processor. Additionally this may make cell configuration time not short enough.

Since cell size and functionality should be strongly reduced in comparison with currently available FPGAs there should be possibility to place thousands of cells in reconfigurable processor and hence to create far more complex partitions. Additionally, despite partitions using more cells, configuration time of average partition should become far shorter mainly because of smaller amount of configuration data.

B. *Amount of configuration data*

Cell used only for routing the internal signals for partition requires smallest amount of configuration data. Cells being more complex elements of their partition may require more data to be configured. However even in worst case the amount of this data should be not larger than one or two hundreds bits. Average cells in partition should require amount configuration data being in between the minimal and maximal one — tens to hundred bits.

Average partition composed of larger number of cells may require several kilobits of total configuration data. In such case average configuration time should be similar to execution time. Dividing program to small partitions should simplify them and hence make them require less configuration time and operate with higher clock frequency. Additionally smaller partitions may be placed in cell matrix in higher number increasing the parallelism of execution. This should give higher overall performance.

C. Clock trees

To obtain maximal performance from each partition its clock frequency should be adapted to it. Of course whole processor could operate with the same low frequency so even slow complex partitions would be able to operate properly. However such solution would force all partitions to operate with the speed of the slowest one. Such approach would seriously spoil the performance.

Because of that each partition should have possibility to choose best clock frequency for its needs. However this requires a number of clock frequencies to be available for partition to be chosen. The more the better. However placing large number of clock trees would significantly enlarge the whole circuit. Additionally designing the circuit with large number of clock trees would be a difficult task.

Since partition configuration does not have to consider signal flow within the one being configured it have possibility to be performed with far higher clock frequency than partition execution clock. Additionally configuration clock frequency does not depend on partition but on cell architecture and hence is constant. During configuration cells may operate according to configuration clock and when partition is ready they should switch to receive partition execution clock signal.

D. Word size

Many sequential processors are operating using word being 32 or 64 bits long. Similar assumption should be made of reconfigurable processor. Although single cell is operating on words of particular length, partition may operate on words having nearly any length. This may be simply achieved thanks to adaptation possibility.

Thanks to this possibility some specific, complex calculations on huge numbers that currently have to be performed in software could be fully implemented within single partition. Hence those calculations can be executed with far lower amount of clock cycles and in shorter time. This should strongly increase performance of applications operating on huge numbers like coding/decoding signals, cryptography and similar ([3]).

IV. Software problems and solutions

A. Compiler

One of the most crucial element for reconfigurable processor is possibility to create of new programs efficiently. Additionally simple porting of already portable programs is also necessary. Creation of processor without software is pointless and need to rewrite all programs from the very beginning would make reconfigurable processor useless. Difficult way of software creation would also make proposed solution not very useful and hence stop its future development.

Because of that there is need to create new compiler or extend some of currently available ones. Such compiler should be able to compile any high level programming language being used and compile it either to any HDL or better directly to bitstream (since HDL also should be later translated to bitstrem). To maximally simplify porting and creation of software popular programming languages should be supported. However the larger number of supported programming languages the better.

One of the best possible solution would be creation of back-end to some commonly used compilers like GCC. Mainly because there are many existing front-ends to them translating programming languages into their internal languages. Additionally popular compilers are well performing language dependent optimisations and are constantly being updated.

B. Available solutions

Currently there are several projects of compilers [5][6] being able to compile popular programming languages like C++ to popular HDLs like Verilog. However all of them are strongly limited and hence are unable to perform full conversion. This makes them useless for reconfigurable processor.

Additionally all those solutions would require compilation of HDL to bitstreem suitable for this processor. As described in [7] this would be a non trivial task of similar complexity as compilation of programming languages. And since reconfigurable processor is supposed to operate as general-purpose one, compilation from programming language to HDL and then to bitstream (even if possible) would be not optimal.

Hence there is need for compiler compiling programming languages directly to bitstream. Creating such compiler from the very beginning would significantly reduce the number of supported programming languages. Additionally in existing compilers there are optimisations that are also useful for project reconfigurable processor like dead code elimination or others. Because of that some already existing compilers with available source code like LLVM or GCC should be modified to be able to support bitstream of reconfigurable processor as one of possible outputs. This would require adding another back-end and several architecture specific optimisations performed on internal representation of code being compiled.

Since architecture of reconfigurable processor should be adapted to operate as typical one with precisely defined elements like memory accessing or partitions switching there should be no difficulty with compilation of internal code representation to bitstream. This should allow for compilation of any programming language supported by chosen compiler front-ends.

C. Architecture specific optimisations

Compiler for processor presented in this paper should analyse the code to perform architecture specific optimisations. Each basic operation can be expressed as number of bitstream templates. The choice of particular one should be base on several parameters. For example particular operation may be computed often and hence use larger number of cells to perform in shorter time or may be computed once in longer time but should not waste resources. Even combining operations requires analyse of where to place the input and output registers. Additionally such elements as access priority also should be considered during optimisation.

In general in reconfigurable processor operations may be executed in parallel if only there are no dependencies making it impossible. However usage of the resources has also to be considered because of large but limited amount of them. In some cases lose of performance caused by huge size of partition may spoil the gain of parallel execution. In such case elements responsible for those operations should be reordered not to produce long signal paths. If this still can not solve the problem then sequential execution of some of the operations should be considered. Additionally in worst case there may appear the situation in which there is not enough of resources to perform all of the operations in parallel. However reconfigurable processor is designed for massively parallel execution of programs and hence such situations should be rare.

Another important optimisation is reduction of dependencies between operations. This may significantly increase parallelism of program and hence increase performance. Hopefully some optimisations for increasing the parallelism of execution are already available in compilers. Even if the will not be suitable for this project, they can be considered as base for specific optimisation.

Because partitions switching mechanism is one of the most complex element and the one that has large influence on overall performance, switching of the partitions should also be minimized. Some of the partitions (for example the ones being bodies of loops or calls to frequently used functions) are executed frequently. In their case it may be not efficient to remove them at the moment they end their execution just to load them soon. Such partitions may be frozen after one of their execution just to be slightly reconfigured or just read from memory new parameters and rerun when needed. Of course such solution does not release the resources. Hence this optimisation should depend not only on the frequency of usage of partition, but also on its size. Large partitions staying idle will only consume resources (mainly space on cell matrix) that could be utilised by other partitions other partitions.

Unfortunately all mentioned architecture specific optimisation will require additional empirical research. Because of that no details about them can be currently known. Hence now they can be discussed only as general concept of what will be required from compiler.

D. Operating system

Another important software problem is operating system. There is extremely large variety of available ones and reconfigurable processor should be conceptually compatible with them. However existing systems may be not well suited for reconfigurable processor and hence require to be modified or extended. Similarly to compiler's architecture specific optimisations mentioned before, currently (without empirical research) no details can be found. Additionally the need for neither extension nor modification of currently available operating systems may not be certain. However, since the novelty of this project puts high demands on software, such need has to be taken into account.

V. Summary

This paper briefly presents the concept of dynamically reconfigurable general-purpose processor realised as dynamically reconfigurable FPGA. The main novelty of proposed solution is lack of sequential execution being typical for currently available processors. Additionally presented solutions implement all the operations in hardware.

To fulfill those assumptions there is need to create new architecture being different from both currently available FPGAs and sequential processors. Additionally there is necessary dedicated compiler being able to compile source code directly to suitable bitsreaem in optimal way, not to spoil performance and to fulfill high demands put on the software. To create such there is need for additional, empirical research that can not be performed currently.

References

[1] H. V. J. M. Moreno, A. Villa, A. Napieralski, G. Sassatelli, and E. Lavarec, "Perplexus: Pervasive computing framework for modeling complex virtually-unbounded systems," *Proceedings of the 2007 NASA/ESA Conference on Adaptive Hardware and Systems*, pp. 587–591, Aug. 2007.

[2] L. Zhuo and V. K. Prasanna, "Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, pp. 433–448, Apr. 2007.

[3] M. H. Tarek El-Ghazawi, Esam El-Araby, K. Gaj, V. Kindratenko, and D. Buell, "The promise of high-performance reconfigurable computing," *Computer*, vol. 41, pp. 69–76, Feb. 2008.

[4] A. T. Hayden Kwok-Hay So and R. Brodersen, "A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph," in *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, 2006, pp. 259–264.

[5] Y. D. Y. Arcilio J. Virginia and K. L. Bertels, "An empirical comparison of ansi-c to vhdl compilers: Spark, roccc and dwarv," *ProRISC, 18th Annual Workshop on Circuits, System and Signal Processing, Utreht 2007*, pp. 388–394, Nov. 2007.

[6] D. Jain, "Object oriented programming constructs in vhsic hardware description language 'why & how'," *Journal of Theoretical and Applied Information Technology*, vol. 3, pp. 30–37, Jan. 2007.

[7] F. W. Wibowo, "Interoperability of reconfiguring system on fpga using a design entry of hardware description language," *Proceedings of International Conference on Advances in Computing, Control, and Telecommunication Technologies 2011*, vol. 2, pp. 79–83, Mar. 2011.

Igor Zarzycki recieved the MSc degree in the field of Computer Science at Lodz University of Technology in 2010. He continues his education as a PhD student at Department of Microelectronics and Computer Science. His interests include software developement and compiler construction. His recent research concerns compilation of software description languages to hardware description ones.