Int. J. Appl. Math. Comput. Sci., *2010*, *Vol. 20*, *No. 3*, *581–589*
DOI: 10.2478/v10006-010-0043-1

LOOP PROFILING TOOL FOR HPC CODE INSPECTION AS AN EFFICIENT METHOD OF FPGA BASED ACCELERATION

MARCIN PIETROŃ [*], PAWEŁ RUSSEK [*,**], KAZIMIERZ WIATR [*,**]

[*] Department of Electrical Engineering and Computer Science
AGH University of Science and Technology, al. Mickiewicza 30, 30–059 Cracow, Poland
e-mail: {pietron,russek,wiatr}@agh.edu.pl

[**]Academic Computer Centre *Cyfronet AGH*
ul. Nawojki 11, 30–590 Cracow, Poland

This paper presents research on FPGA based acceleration of HPC applications. The most important goal is to extract a code that can be sped up. A major drawback is the lack of a tool which could do it. HPC applications usually consist of a huge amount of a complex source code. This is one of the reasons why the process of acceleration should be as automated as possible. Another reason is to make use of HLLs (High Level Languages) such as Mitrion-C (Mohl, 2006). HLLs were invented to make the development of HPRC applications faster. Loop profiling is one of the steps to check if the insertion of an HLL to an existing HPC source code is possible to gain acceleration of these applications. Hence the most important step to achieve acceleration is to extract the most time consuming code and data dependency, which makes the code easier to be pipelined and parallelized. Data dependency also gives information on how to implement algorithms in an FPGA circuit with minimal initialization of it during the execution of algorithms.

Keywords: HPC, HPRC (High Performance Reconfigurable Computing), loop profiling, Mitrion-C, DFG (Data Flow Graph).

1. Introduction

Our main goal is to accelerate HPC scientific applications (Russek and Wiatr, 2005; 2006). In this paper we concentrate on our approach to accelerating HPC applications to FPGA platforms. We try to check the possibilities of automated porting of HPC source codes to HPRC platforms. Our main objective is to build a universal tool that could be used in any scientific application and would enable it transform this source code to a chosen HPRC platform.

The main application on which we started developing and testing our system is Gaussian quantum-chemistry software. Gaussian is a Fortran application which simulates chemical molecules. Our working environment is SGI Altix 4700: an SMP system with the RASC (Reconfigurable Application Specific Computing) platform.

There is a gap between the existing HPC applications and the new HPC or HPRC hardware platforms which have been built. The new hardware platforms very often could not be used in an optimal way by HPC applications. The main reason for this is that there is a lack of automated tools able to port HPC applications to new HPC or HPRC hardware platforms (Gasper *et al.*, 2003). Since several HPC platforms with the FPGA were created, some publications were written which show results of implementing scientific algorithms on such platforms (Kindratenko *et al.*, 2007; Liu *et al.*, 2008). This shows that the implementation of some scientific algorithms on HPRC platforms can be faster than on CPUs. The methodology of speeding up HPC application in implementing single algorithms is quite inefficient. The huge amount of code needs automated analysis, transformation and implementation (Deng *et al.*, 2009). What has to be done first to speed up HPC application is the extraction of a code suitable for the FPGA acceleration. Therefore several mechanisms must be implemented to achieve this goal. These are *Loop Profiler* and *DFG* (Data Flow Graph) *Builder*. The former is necessary because research and practical software knowledge state that 90 percent of the execution time of programs is spent in loops. The latter is required to extract the dependency between data.

The paper is organized as follows: Section 2 provides a description of the hardware platform which our research is focused on, Section 3 presents the architecture of our tool—depicts the functionality and key software module of the system. Section 4 illustrates *DFG Builder*. Section 5 elaborates on the main part of our system—*Loop Profiler*. Section 6 provides a description of how data gathered by *Loop Profiler* can be used in the process of further speeding up of software applications. Section 7 and 8 present conclusions and directions further research, which will be performed in the near future.

2. Hardware platform

High-performance computing companies such as SGI® and Cray Inc. have produced several HPRC platforms. There are also new vendors on the HPC market including SRC Computers Inc. and Nallatech Ltd. with their own HPRC solutions. The SGI solution is in the SGI Altix family.

The Altix 4700 series is a family of multiprocessor distributed shared memory computer systems, and it currently ranges from 8 to 512 CPU sockets (see Fig. 1). Each processor has its own local memory as well as the ability to access very fast memories of other processors by a NUMALink connection. NUMALink allows dataflow of 6,4 GB/s. SGI RASC RC100 Blade consists of two Virtex-4 LX 200 FPGAs, with 40 MB of SRAM logically organized as two 16 MB blocks and an 8 MB block.

Each QDR SRAM block transfers 128-bit data every clock cycle (at 200 MHz), both for reads and writes. The RASC communication module is based on an application-specific integrated circuit (ASIC) TIO, which attaches to the Altix system NUMAlink interconnect directly. TIO supports the Scalable System Port (SSP) that is used to connect the Field Programmable Gate Array (FPGA) with the rest of the Altix system. The RC100 Blade is connected using the low latency NUMALink interconnect to the SGI Altix 4700 Host System. NUMALink enables a bandwidth of 3.2 GB per second in each direction.

Altix 4700 has its own built-in development platform which gives RASC API the ability to write programs on a host processor that invoke compiled VHDL source codes on FPGA circuits. The second possibility of the development of HPC application is to use the RC100 Blade to write the source code in an HLL (see Fig. 2). Mitrion-C is an HLL which facilitates this process. The Mitrion-C compiler generates a VHDL code from the Mitrion-C source. Then Mitrion sets up the instance hierarchy of the RASC FPGA design that includes the user algorithm implementation, the RASC Core Services, and the configuration files. The design is then synthesized using the Xilinx suite of synthesis and implementation tools. Apart from the bitstream generated, two configuration files are created: one describes the algorithm‘s data layout and the streaming capabilities to the RASC Abstraction Layer (bitstream configuration file), and the other covers various parameters used by the RASC Core Services. These files are required by the device manager to communicate with the algorithm implemented on the FPGA.

Implementing and invoking the algorithm on RASC consist of several functions which reserve resources from the host processor, queue commands and other preparation steps. The reason behind this is that the optimal structure of the hardware-accelerated application should contain as few initializations of the FPGA circuits as possible.

3. Architecture of the system

This section presents modules which are parts of the system. The system works on Fortran 77 and Fortran 95 applications. The source code is written in the Java language. Figures 3 and 4 describe the main modules and system classes, respectively.

The design of our system can be divided into three main parts of functionality. These include:

- *Front-end*: parses the source code and makes instrumentation, takes platform parameters;
- *Engine*: computes data dependency, analyzes loops, etc.;
- *Back-end*: generates the HPC source code with an HLL.

The input to our system is the source code of the application and data about the platform on which the speedup should be done. This is the input to the front-end part of our system. This part is responsible for parsing the source code and the instrumentation code for further profiling. The automated instrumentation is needed for time measuring of loops execution, the number of iterations counted and measuring the amount of data used during loop computation. While parsing the source code, the *Parser* class creates *Loop*, *Instruction* and *Variable* classes when a loop is recognized. During the creation of these classes the parsing module invokes the *Instrumentator* class, which is responsible for the whole instrumentation of the source code. The functionality of all these classes is described at the end of this section. Parsing gives the following information:

- extracts loops,
- loop iteration variables,
- list of instructions in a loop,
- sets of data used in loop computation.

After that, the data gathered during parsing are sent to *DFG Builder* and *Recompiling Module*. The latter is now able to compile the instrumented source code and run it to gather profiling data.

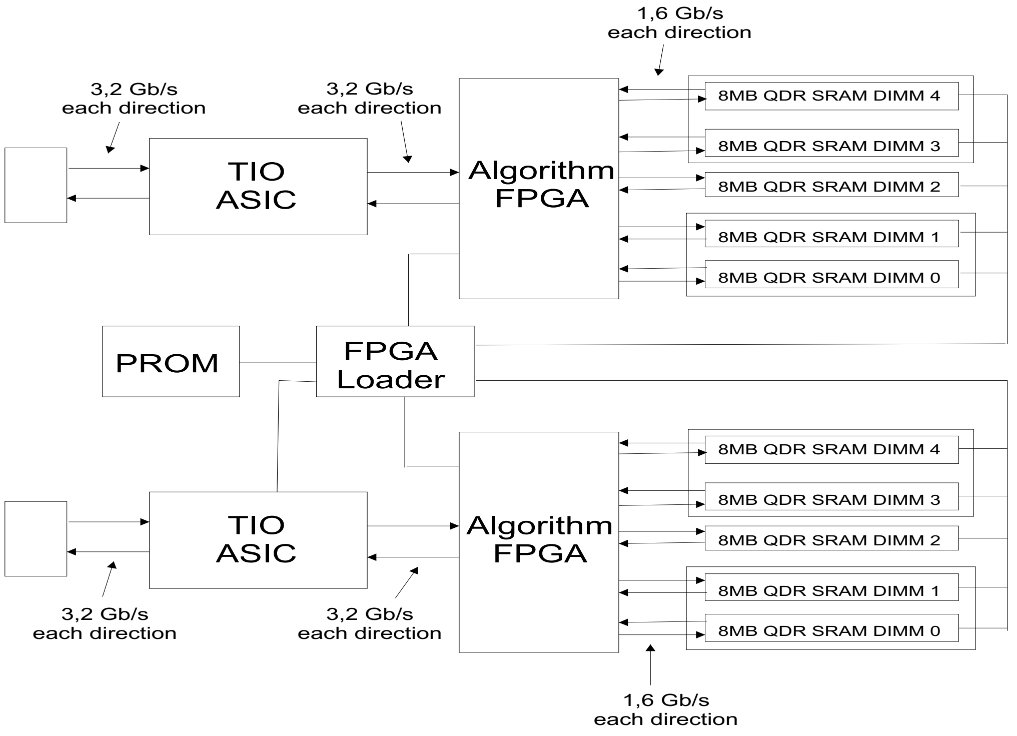


Fig. 1. SGI Altix 4700 Host System (SGI Altix 4700 documentation).

Loop DFG Builder, *Loop Execution Time Profiler*, *Loop Analyzer* and *Time Estimator* are the engines of our system. These modules gather all the essential data from which the system can extract parts of code that can be implemented in HLL.

DFG Builder creates graphs with data dependency inside each loop and data dependency between loops and loop nesting. Section 4 describes this module. *Loop Analyzer* is a system class responsible for analyzing data gathered by *DFG Builder*, *Loop Time and Data Profiler*. This analyzes parallelism in loops and helps loop optimization (Subsection 6.1). *Time Estimator* estimates execution time of loops in FPGA. It measures time as a sum of

- sending and loading the bitstream to an FPGA (created by an HLL),
- sending input data from the host to the FPGA,
- execution time of the algorithm in FPGA (time obtained from the HLL simulator),
- sending output data to the host program.

The chosen parts of the code are input data for the HLL generator, the last functional block—the back-end (*HLLGenerator*). This element generates parts of the HPC source code chosen by *Loop Analyzer*. In our case, this is the Mitrion-C generator. The methodology of generating Mitrion-C is described in Subsection 6.3.

Our goal was to build a system that can easily be adapted to different HPC platforms. Partitioning the architecture into the three described parts enables it to adapt to various HPC applications (replacing the front-end) or various HPRC platforms (replacing the HLL generator or the platform parameters).

The main classes of the system are (see Fig. 4):

- *Parser*: parses the source code of the HPC source code application;
- *DFG Builder*: builds data flow graphs of loops of the parsed code;
- *Instrumentator*: class instrumenting parts of the code, e.g., the execution time of loops, the size of tables;
- *Loop*: computes data dependency in a loop;
- *Instruction*: gives information about each instruction in a loop, e.g., the types of operation, the input data, the type of data;
- *Analyzer*: takes and analyzes the results of *Loop Time Profiler*, data dependency from *DFG Builder*, the loops data, etc.;

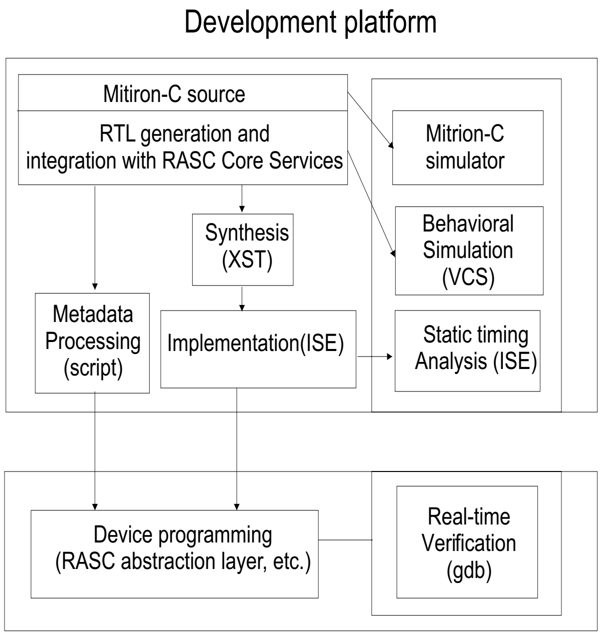

Development platform
Mitiron-C source
RTL generation and integration with RASC Core Services
Synthesis (XST)
Metadata Processing (script)
Implementation(ISE)
Mitrion-C simulator
Behavioral Simulation (VCS)
Static timing Analysis (ISE)
Device programming (RASC abstraction layer, etc.)
Real-time Verification (gdb)

Fig. 2. RASC FPGA platform design flow (Mohl 2006).

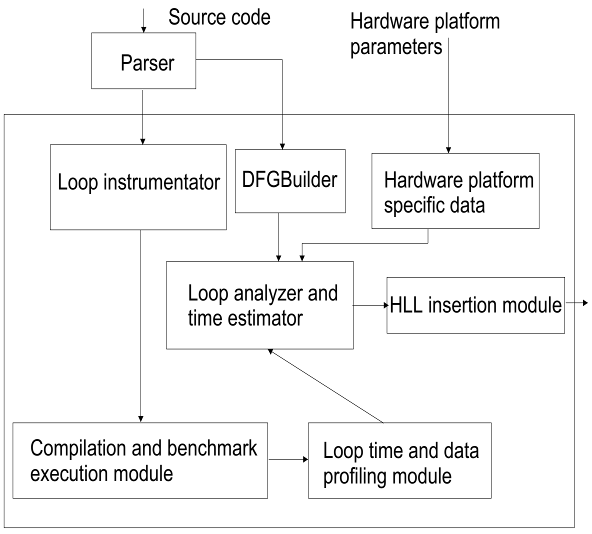

Source code
Hardware platform parameters
Parser
Loop instrumentator
DFGBuilder
Hardware platform specific data
Loop analyzer and time estimator
HLL insertion module
Compilation and benchmark execution module
Loop time and data profiling module

Fig. 3. Hyper profiling tool architecture.

- *Estimator*: estimates the time and area of chosen parts of the code implemented in the FPGA;
- *Visualizer*: visualizes the data flow graph;
- *Parameter*: class with parameters about the platform set by the user (e.g., data bandwidth of the link between the host and the FPGA circuits, SRAM memory size);
- *HLL Generator*: inserts an HLL to the chosen spots of the HPC source code.

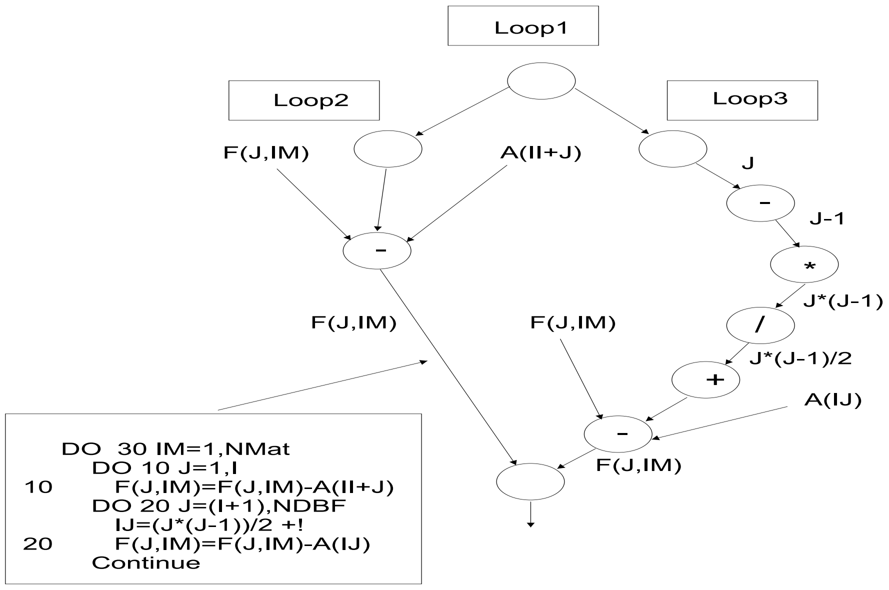
4. DFG Builder

As shown in Fig. 3, the hyper profiling module consists of *DFG Builder*, which gives necessary information about data dependency inside the loops and dependency between loops. The main purpose of the *DFG Builder* tool is to extract data and loop dependency. An example of the DFG is presented in Fig. 5. In our data flow, graph nodes are a single operation and edges are input variables. *DFG Builder* receives data from the parsing module and creates from a graph of dependency. It receives a set of loops identified during the parsing process. Each received loop has its own list of instructions. The instructions are delivered with all the operations and variables used.

The dependencies between loops are the most important data extractions in the case of HLL (Mitrion-C) mapping. As shown in Subsection 6.1, it depends on which type of Mitrion-C loop will be inserted. Figure 5 shows only some of the data during the analyzing of the source code (performed by *Visualizer*). The rest of the information on data dependency is saved in the log file. *DFG Builder* creates a loop call graph. From this graph we can obtain information about the loop's nesting, which makes it possible to ascertain which groups of loops can be implemented at once. Figure 6 shows the results of analyzing one of the main Gaussian libraries by *DFG Builder*. 'Sets of nested loops' on the X-axis means the type of set found by *DFG Builder*, e.g., '1' describes the set with a single loop, '2' means that set contains two loops that can be implemented at once, etc. The Y-axis describes how sets of loops are data dependent (wide parallel, partially parallel or sequential).

5. Loop Profiler

The first step we tried was to extract the code, which could be hardware implemented, on the FPGA platform was standard profiling. It reports the percentage of executed time in each function and subroutine. One of the best profilers used by us to achieve this goal was *oprofile*. It is mentioned by Pietroń *et al.* (2007a; 2007b) that oprofile gives the best results while profiling the Gaussian application. *Oprofile*, like other standard profilers, gives only limited results to functions and subroutines. This information is insufficient to find the most suitable code that could be sped up on the FPGA platform. Hence the next step is necessary. A special tool for hyper-profiling was built to extract the code for the speedup. The main part of the hyper-profiling tool is *Loop Profiler* (see Fig. 7). Loop



```
    DO  30 IM=1,NMat
      DO 10 J=1,I
10      F(J,IM)=F(J,IM)-A(II+J)
      DO 20 J=(I+1),NDBF
        IJ=(J*(J-1))/2 +!
20      F(J,IM)=F(J,IM)-A(IJ)
      Continue
```

Fig. 5. Generating a DFG graph from the F77 source code.

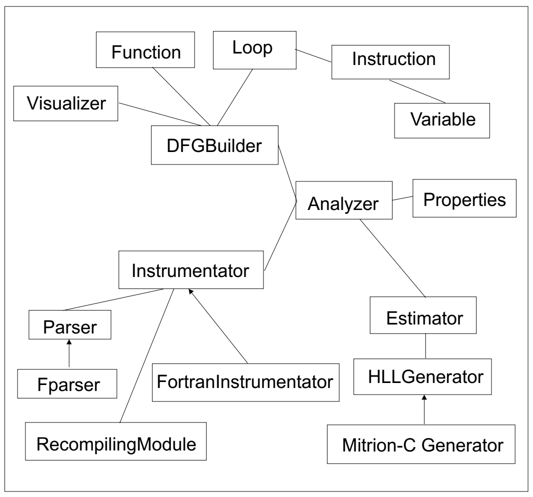


Fig. 4. Simple UML diagram.

profiling is a process which gives information about the execution time of loops. Apart from the execution time, a lot of various data can be collected during profiling. Some other information which can be gathered and used for parallelizing the code is, e.g., the number of loop iterations and the number of loop entries. There are two types of loop profiling: static and dynamic. Dynamic loop profiling (Moseley *et al.*, 2006) is done by dynamic instrumentation in the executed code, whereas in the case of static loop profiling instrumentation is done on the source code of the application. Dynamic loop profiling is compiler independent and is designed to work with a chosen family of processors (Moseley *et al.*, 2006). Static profiling, as presented in this paper, is language dependent.

In the present version of our hyper-profiling tool, loop profiling can be done on Fortran (F77 and F95). Language loops are instrumented as follows: *do*, *do while*. The module in the case of the Gaussian (f77 source code) application implements algorithms which analyze the Fortran 77 code and instrument the code (see Fig. 7). The instrumentation of the source code is done to obtain information about the execution time of loops. The results of profiling are saved in formatted files (as shown in Fig. 7). Apart from this, *Loop Profiler* gathers information about data used in the loop's computations. As Fig. 8 shows, *Loop Profiler* makes instrumentation for collecting the size of the input loop's data. This process is done by monitoring (instrumentation) the boundaries of a loop's iterations (Subsection 6.2).

The most important data gathered by the loop profiling data module are the following

- number of iterations of each loop,
- number of entries of each loop,
- execution time of each loop,
- size of data used in a loop's computations.

The *Loop Profiler* data can be used to speed up the HPC code in two ways. The first is implementing chosen loops in the HPRC platform libraries (in the case of SGI RASC it is the RASClib library), and the second is implementing the loops in an HLL (such as Mitrion-C). This

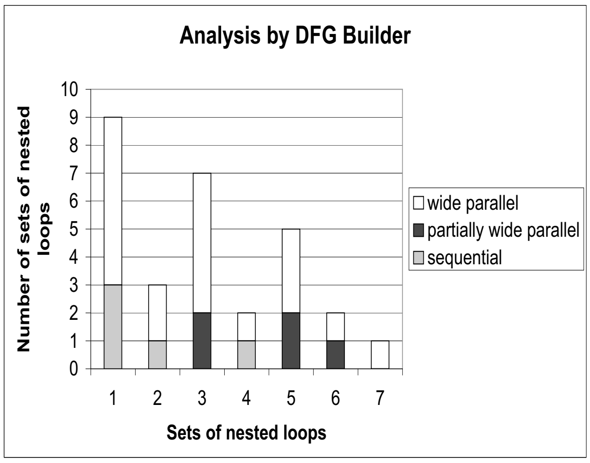
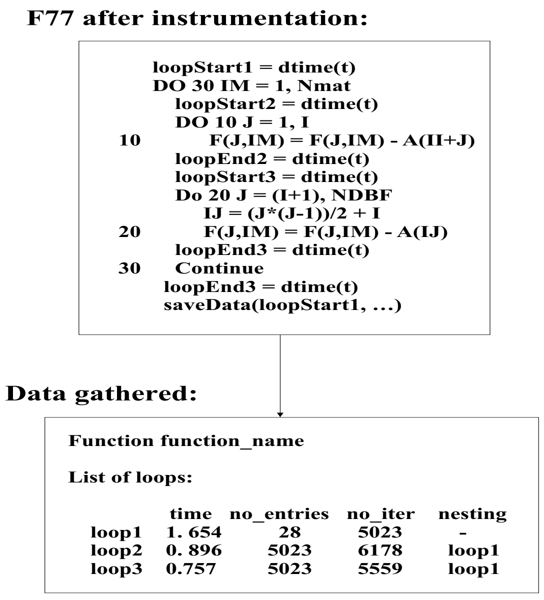


Fig. 6. Example data gathered during analyzing part of the main Gaussian libraries.



F77 after instrumentation:

```
      loopStart1 = dtime(t)
      DO 30 IM = 1, Nmat
         loopStart2 = dtime(t)
         DO 10 J = 1, I
10          F(J,IM) = F(J,IM) - A(II+J)
         loopEnd2 = dtime(t)
         loopStart3 = dtime(t)
         Do 20 J = (I+1), NDBF
            IJ = (J*(J-1))/2 + I
20          F(J,IM) = F(J,IM) - A(IJ)
         loopEnd3 = dtime(t)
30       Continue
        loopEnd3 = dtime(t)
        saveData(loopStart1, ...)
```

Data gathered:

```
Function function_name

List of loops:

         time    no_entries  no_iter   nesting
loop1   1. 654       28        5023        -
loop2   0. 896      5023       6178      loop1
loop3   0.757       5023       5559      loop1
```

Fig. 7. Writing results of loop time profiling.

Table 1. Mitrion-C loops (Mohl 2006).

	Vector	List
foreach	wide parallel	pipelined
for	unrolled	sequential

process is presented in Section 6. Before using both methods, the data flow graph should be generated as well as data profiling to find out about the dependency between loops iterations and data used in loops computing (*DFG Builder* and *Loop Analyzer*), see Fig. 9.

6. Loop profiling for speeding up the HPC code

As shown in the previous sections, loop profiling is a necessary step in the process of speeding up HPRC applications. This section describes further algorithms on loop and data optimization. It shows how data gathered from previous modules can be analyzed and used in the process of speeding up HPC applications. Subsection 6.1 elaborates on loops optimization using (*DFG Builder* and *Loop Analyzer*), Subsection 6.2 presents loop data profiling. Subsection 6.3 describes the process of incorporating the Mitrion-C language into the HPC source code.

6.1. Loop optimization. There are several methodologies to optimize loops. The most important are presented in this section. All of them are included in our system in *DFG Builder* and *Loop Analyzer*. The latter is able to find such loops and refactor them. Below we show each type of optimization and provide a short description of the algorithm:

- loop unrolling: increases parallelism within the loop body, reduces loop overhead per iteration, modifies the loop step, and appends as many copies to the loop body as needed;

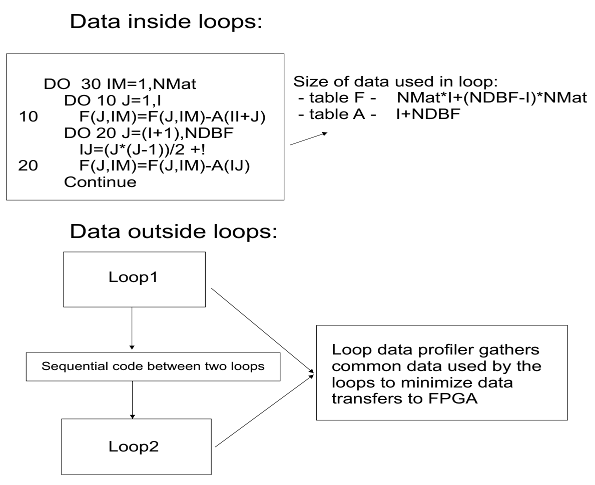
- loop fusion: reduces redundancy, eliminates loop overhead and redundant computations by combining the loops bodies into single loop;

- loop unswitching: removes *if* statements from within a loop when the test of the conditional is independent of the loop;

- loop peeling: enables loop fusion whenever the iteration counts of the candidate loops do not match;

- loop tiling: processes the data of the loop in tiles, optimization is used to improve data locality.

6.2. Loop data profiling. As mentioned in earlier sections, the frequency of data transfer from the host processor to the FPGA circuits and the amount of these data are critical issues in HPRC platforms. Consequently, it is necessary to measure and collect information about it while analyzing the source code. For each loop, data profiling is performed. This informs us about the amount of data which must be sent to the FPGA circuit while implementing the loops. An example of loop data profiling is shown in Fig. 8. Data reports from loop profiling are necessary to estimate the execution time of the implemented algorithm (*Time Estimator*, Section 3), see Fig. 9.


Data inside loops:
Size of data used in loop:
- table F - NMat*I+(NDBF-I)*NMat
- table A - I+NDBF
Data outside loops:
Loop1
Sequential code between two loops
Loop2
Loop data profiler gathers common data used by the loops to minimize data transfers to FPGA

```
     DO 30 IM=1,NMat
        DO 10 J=1,I
10         F(J,IM)=F(J,IM)-A(II+J)
        DO 20 J=(I+1),NDBF
           IJ=(J*(J-1))/2 +!
20         F(J,IM)=F(J,IM)-A(IJ)
        Continue
```

Fig. 8. Loop data profiling.

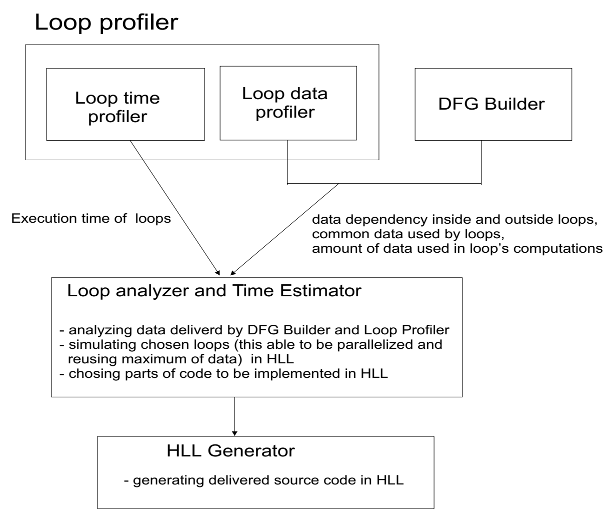

Loop profiler
Loop time profiler
Loop data profiler
DFG Builder
Execution time of loops
data dependency inside and outside loops, common data used by loops, amount of data used in loop's computations
Loop analyzer and Time Estimator
- analyzing data deliverd by DFG Builder and Loop Profiler
- simulating chosen loops (this able to be parallelized and reusing maximum of data) in HLL
- chosing parts of code to be implemented in HLL
HLL Generator
- generating delivered source code in HLL

Fig. 9. Functional diagram of *Loop Profiler* and *DFG Builder*.

6.3. Automated framework for the insertion of the Mitiron-C language in HPC applications. FPGA circuits are programmed using hardware description languages (the VHDL or Verilog). Apart from the HDL, there are several other tools that enable the compilation of a code written in a high-level language directly to the FPGA. These languages include Handel-C, Impulse C and Mitrion-C. In our system, the HLL is Mitrion-C. The Mitrion-C source code is compiled by the Mitrion-C compiler into a code for the Mitrion processor, followed by an automatic adaptation and the implementation of the processor in the FPGA, which targets a specific hardware platform, such as SGI RASC. This method eliminates the need for low-level hardware design.

Mitrion SDK consists of a Mitrion-C language compiler, an integrated development environment, a data dependency graph visualization and simulation tool, a Mitrion Host Abstraction Layer (MITHAL) library, and the target platform-specific processor configurator (see Fig. 10). The Mitrion-C source code is compiled into an intermediate virtual processor machine code. This machine code can be used by the simulator, the debugger or processed by a processor configurator to produce a VHDL design for the hardware platform. The Mitrion-C programming language is an intrinsically parallel language. Mitrion-C data types, such as vectors, lists, and language constructs, such as loops, are designed to support the parallel execution driven by the data dependencies.

The main mechanisms of parallelism are *foreach* and *for* loop constructs. By using them we can achieve paralleled or pipelined code execution in FPGA. The type of Mitrion-C loops and the type of parallelization are shown in Table 1. Additionally, Mitrion-C has a special API that can be used to invoke a Mitrion-C bitstream from the C or the Fortran language (Mitrion-C Host Application Layer). It enables the insertion of the Mitrion-C bitstream into existing C/C++ and Fortran codes, see Fig. 11.

The rules of optimal loops mapping to Mitrion-C need further research. One of the first approaches to using Mitrion-C for implementing an algorithm on Altix SGI RASC and comparing its execution time to the host processor is presented by Kindratenko *et al.* (2007), but because of the amount of the code the algorithm was fully implemented in Mitrion-C manually.

7. Summary

The results gathered from our current research show that the process of speeding up HPC applications on the FPGA platforms is quite complex and multi-leveled. Profiling using standard profilers has shown that this method does not give good results. Implementing elementary functions Pietroń *et al.* (2007), e.g., the exponential function, is also quite an inefficient methodology to achieve speedup of the whole HPC application. This is the reason why an automated loop profiler was created. The architecture of *Loop Profiler* is divided into functional modules. This architecture enables adapting the system to various HPC applications and HPRC platforms. The first module is dependent on the HPC application language (parsing and instrumentation), the next one is the system's engine (data and loop dependency, loop execution time profiling, etc.), the last one is the HLL generator, which depends on the language chosen to speed up HPC application.

The loop profiler analyzing data dependency is one of the most important processes in speeding up the HPC source code on FPGA. The main reason is that loops are potentially one of the easiest parts of the code to be pipelined and parallelized. It should be mentioned that loop profiling without any further analysis of the code is useless in the case of HPRC applications. Loop profiling with ad-

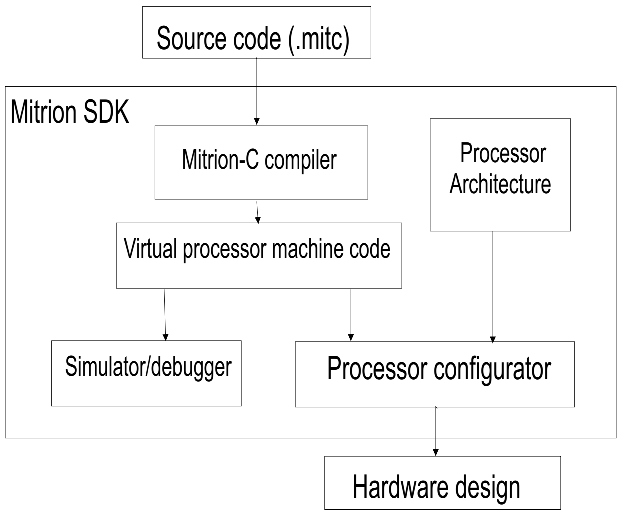
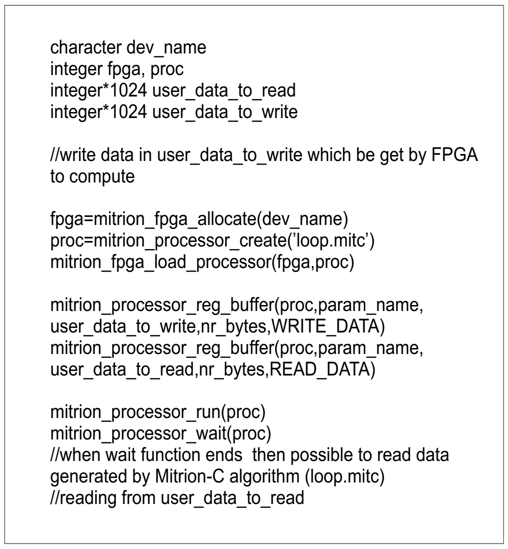


Fig. 10. Design flow.



```
character dev_name
integer fpga, proc
integer*1024 user_data_to_read
integer*1024 user_data_to_write

//write data in user_data_to_write which be get by FPGA
to compute

fpga=mitrion_fpga_allocate(dev_name)
proc=mitrion_processor_create('loop.mitc')
mitrion_fpga_load_processor(fpga,proc)

mitrion_processor_reg_buffer(proc,param_name,
user_data_to_write,nr_bytes,WRITE_DATA)
mitrion_processor_reg_buffer(proc,param_name,
user_data_to_read,nr_bytes,READ_DATA)

mitrion_processor_run(proc)
mitrion_processor_wait(proc)
//when wait function ends  then possible to read data
generated by Mitrion-C algorithm (loop.mitc)
//reading from user_data_to_read
```

Fig. 11. Mitrion-C executed from Fortran.

ditional data gathered, e.g., data dependency and the data flow, can be used to find a code that can be sped up.

8. Future work

Further research will especially be focused on developing and improving the automated framework and incorporating an HLL into existing HPC source codes. In particular, we will concentrate on improving the methodology and algorithms extracting the source code for the speedup. This work will focus on improving the time and area estimation of implementing HLL parts of the code in FPGA. The goal of our research is highly innovative because there have not been any such research results published. The second aspect of this is that, apart from the existing HLLs such as Mitrion-C, which make the development of algorithms dedicated to FPGA faster, there is no effective tool which can make the insertion of an HLL to the existing HPC source code and speed it up. In the near future, a comparative analysis of the automated insertion of an HLL to various HPC applications will be carried out. The research will also be focused on better estimation of the execution time of parts of the code chosen to be implemented.

Future work will also be focused on the possibilities of automated monitoring of the bit-width of data and the values of variables used in computation. As mentioned earlier, the amount of data and the bit-width needed for it in computation are very important issues while implementing algorithms on the FPGA platform.

The next challenge is to widen user interaction with the system. The system should work in two modes. The first one should be fully automated and the second one should allow the user to interact with the system. The user of the system should have the ability to choose the parts of source code to be sped up.

Acknowledgment

This work was financed through research funds by the Polish Ministry of Science and Higher Education as a research project in 2009.

References

Bennett, D., Dellinger, E., Mason, J. and Sundarajan, P. (2006). An FPGA-oriented target language for HLL compilation, *Reconfigurable Systems Summer Institute, RSSI 2006, Urbana, IL, USA*.

Deng, L., Kim, J.S., Mangalagiri, P., Irick, K., Sobti, K., Kandemir, M., Narayanan, V., Chakrabarti, Ch., Pitsianis, N. and Sun, X. (2009). An automated framework for accelerating numerical algorithms on reconfigurable platform using algorithmic/architectural optimization, *IEEE Transactions on Computers* **58**(12): 1654–1667.

Gasper, P., Herbst, C., McCough, J., Rickett, C. and Stubbendieck, G. (2003). Automatic parallelization of sequential C code, *Midwest Instruction and Computing Symposium, Duluth, MN, USA*.

Gong, W.,Wang, G. and Kastner, R. (2004). A high performance application representation for reconfigurable systems, *Conference on Engineering of Reconfigurable Systems and Algorithms, ERSA, Las Vegas, NV, USA*.

Kindratenko, V., Brunner, R. and Myers, A. (2007). Mitrion-C application development on SGI Altix 350/RC100, *International Symposium on Field Programmable Custom Computing Machines, FCCM 2007*, pp. 239–250.

Kindratenko, V., Myers, A. and Brunner, R. (2006). Exploring coarse- and fine-grain parallelism on a high-performance reconfigurable computer, *2nd Annual Reconfigurable Systems Summer Institute, RSSI 2006, Napa Valley, CA, USA*.

Liu, K., Cameron, Ch. and Sarkady, A. (2008). Using Mitrion-C to implement floating-point arithmetic on a Cray XD1 supercomputer, *DoD HPCMP Users Group Conference, HPCMP-UGC, Urbana, IL, USA*, pp. 391–395.

Memik, S.O., Bozorgzadeh, G., Kastner, R. and Sarrafzadeh, M. (2005). A scheduling algorithm for optimization and planning in high-level synthesis, *ACM Transactions on Design Automation of Electronic Systems* **10**(1).

Messmer, P. and Bodenner R. (2006). Accelerating scentific applications using FPGAs, *XCell Journal* **10**(1): 33–57.

Mohl, S. (2006). The Mitrion-C programming language, Mitrionics Inc., Second Quarter, pp. 70–73, `http://www.mitrion.com`.

Moseley, T., Grunwald, D., Connors, A., Ramanujam, R., Tovinkere, V. and Peri R. (2006). LoopProf: Dynamic techniques for loop detection and profiling, *Proceedings of the 2006 Workshop on Binary Instrumentation and Applications, WBIA, Lund, Sweden*.

Pietroń, M., Wiatr, K. and Russek, P. (2007(a)). Methodology of computing acceleration using reconfigurable logic technology in high performance computing, *University of Science and Technology in Cracow Automatica, 2007*, pp. 149–156.

Pietroń, M., Russek, P., Wiatr, K., Jamro, E. and Wielgosz, M. (2007(b)). Two electron integrals calculation accelerated with double precision exp() hardware module, *Reconfigurable Systems Summer Institute, RSSI, Urbana, IL, USA*.

Russek, P. and Wiatr, K. (2006). The prospect of computing acceleration using reconfigurable logic technology in huge computational power systems, *Proceedings of the IFAC Workshop on Programable Devices and Embedded Systems, PDeS 2006, Brno, Czech Republic*, pp. 44-49.

Marcin Pietroń holds the M.S. degree in electronics and telecommunications engineering (2003), and computer science (2005). Currently he is working toward his doctoral degree in computer science at the Department of Electrical Engineering and Computer Science at the AGH University of Science and Technology in Cracow. His research interests lie in hardware-software code-sign and high performance computing.

Paweł Russek received the M.S. degree in electronics engineering from the AGH University of Science and Technology, Cracow (1994), and the Ph.D. degree in electronics engineeering (2003). Currently he is an assistant professor with the Department of Electrical Engineering and Computer Science of the AGH University of Science and Technology in Cracow. His research interests include application specific hardware accelerators, hardware assisted image processing, and high performance computing on FPGAs.

Kazimierz Wiatr received the M.S. degree in electronics engineering from the AGH University of Science and Technology, Cracow (1980), the Ph.D. degree in electronics engineeering (1987), and the professorial title in electronics engineering (2002). Currently he is a professor with the Department of Electrical Engineering and Computer Science of the AGH University of Science and Technology in Cracow and the director of the Academic Computer Centre *Cyfronet AGH*. His research interests focus on image processing systems, multi-processor systems, and FPGA-based accelerator design.

Received: 22 November 2009
Revised: 26 March 2010