Software/Hardware Partitioner

Silviu Horia Baranga
Faculty of Automatic Control and Computer Science
University "Politehnica" of Bucharest
Romania, Bucharest
Email: silviu.baranga@gmail.com

Adriana Szekeres
Faculty of Automatic Control and Computer Science
University "Politehnica" of Bucharest
Romania, Bucharest
Email: adriana.szekeres@gmail.com

***Abstract*—The current trend in processor's design is to add multiple cores to increase the system's overall performance but this is not a solution to increasing the performance of serial applications. Due to its potential to greatly accelerate a wide variety of serial applications, reconfigurable computing has become a subject of a great deal of research. Its key feature is the ability to perform computations in hardware in order to increase performance, while retaining much of the flexibility of a software solution. In this paper, we address the problem of fully automating the process of selecting the code to be used for hardware acceleration. We present a software-hardware partitioning system that transforms Impulse C source code into blocks of C and VHDL code. The resulting C code will be run on the CPU, while the VHDL code will be implemented on a reconfigurable hardware, e.g. a FPGA.**

I. Introduction

There are two ways of implementing an algorithm: in hardware, using an ASIC (Application Specific Integrated Circuit) or in software, using a microprocessor. An ASIC is very fast and efficient but it is not intended for general purpose use. Once designed, an ASIC is optimized for a given computation and cannot be reconfigured. A more flexible solution is to implement the algorithm as a set of instructions which can be executed by the microprocessor. Without altering the hardware, the microprocessor can be reprogrammed, by changing the set of instructions. However, the cycles needed to read and decode each instruction induce a significant overhead.

Reconfigurable hardware has gained much popularity in the past years, especially in embedded design. It combines the flexibility of microprocessors with the speed of the ASICs. The most common reconfigurable devices are the FPGAs (Field Programmable Gate Arrays). An FPGA contains a large number of *logic blocks* which are connected through a *configurable network*. The canonical logic block is considered to be a 4-input LUT (lookup table) that can implement any 4-input logic function.

Reconfigurable hardware is used as a building block for reconfigurable computing. Reconfigurable computing refers to the possibility of reconfiguring on-demand a hardware agent that has previously been configured to carry out a specific task. With reconfigurable computing came the idea of augmenting a general-purpose processor with an array of reconfigurable hardware. In this manner, parts of a software application could be accelerated in hardware. However, the process of finding the best parts of an application to fit in the existing hardware is a very difficult task.

In this paper we address the problem of fully automating the process of partitioning an application into parts that will run on a processor and parts that will be implemented in hardware. Due to the existence of modern hardware description languages (HDLs) the process of partitioning an application written in a high-level language, like C, could be fully automated.

II. Previous Related Work

Since the development of modern hardware description languages (HDL), much research has been done towards achieving an automated hardware/software partitioning system. There are two approaches to software/hardware partitioning: fine- and coarse-grain. While fine-grain considers basic blocks, the other one means that whole functions or processes are moved from software to hardware or vice versa in order to find the best hardware/software partition. The first approaches to automated hardware/software partitioning, both fine-grained, are the well known VULCANII system [1] and COSYMA [2]. However, in [3], a new method has been proposed that dynamically changes the granularity.

The type of the partitioning problem is derived from its formulation. In the vast majority of cases, the resulting type of the problem is NP-hard, determining researchers to find efficient heuristics, [4]. In [5], [6], [7] and [8], the authors present different ways to search the solution space for the best possible partition, such as: genetic algorithms, ant colony optimization, SMT solvers and possibilistic programming.

In this paper, the coarse-grained approach will be considered with genetic algorithm based exploration of the solution space.

III. CPU-FPGA Coupling

The most important feature of an FPGA is that it can be used to create application specific circuits which can greatly increase the performance over traditional CPUs. However, today's applications are often too large to be implemented directly in hardware. A solution to this problem is to combine the speed of an FPGA with the complexity of a processor. This design would be appropriate for executing many modern applications.

Many such hybrid designs have already been implemented. One of them, which we also used as the target architecture for our application, is the MicroBlaze soft-processor implemented on a Xilinx FPGA. The MicroBlaze processor has three bus connections - the Local Memory Bus (LMB), the On-chip Peripheral Bus (OPB) and the Fast Simplex Link (FSL) interface (Fig. 1).

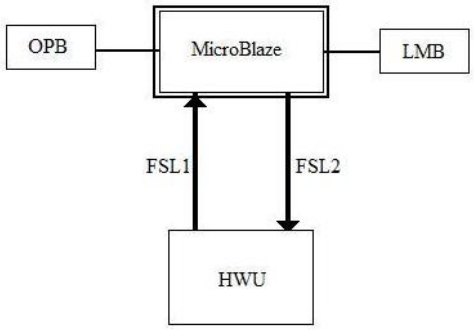

Fig. 1. MicroBlaze processor

FSLs are high-speed, point-to-point interfaces which can be used to connect high-speed hardware units (HWU). The FSL ports on MicroBlaze are accessed via simple get and put assembly instructions, unlike the LMB and OPB which are memory-mapped. Each FSL connection contains a master (write) and slave (read) port. There are maximum 16 FSL ports available on the MicroBlaze processor (8 master / 8 slave).

IV. Software-Hardware Partitioner

In this section we will describe SHP, our software-hardware partitioner. SHP searches for the best solution to partition a given application into functions that will be accelerated in hardware and functions that will run on a CPU. A partitoner is very important because the best solutions to this problem are very hard to obtain, and almost impossible to be deduced manually because of the very large solution space which must be searched.

A. Software/Hardware Processes

In order to be partitioned, the input application must be implemented as a set of communicating processes. Each process may be compiled as a set of instructions to be run on a CPU - *software process*, or may be synthesized in hardware - *hardware process*. Software processes communicate with hardware processes through FSL connections. If a process is moved from software to hardware, or vice-versa, new/existing FSL connections may be opened/closed.

With this constraint, we have chosen Impulse C as the language in which input applications should be written. Impulse C provides a C-compatible library that allows applications written in standard C to be mapped onto coarse-grained parallel architectures. The next three figures show an example of a possible input application, written in Impulse C.

```
void my_test_function(co_stream s_in, co_stream s_out)
{
    int i;
    int hi = 10000000;
    int32       err;

    co_stream_open(s_in, O_RDONLY, INT_TYPE(32));
    co_stream_open(s_out, O_WRONLY, INT_TYPE(32));

    HW_STREAM_WRITE(my_test_function, s_out, hi);
    HW_STREAM_READ(my_test_function, s_in, hi, err);

    HW_STREAM_CLOSE(my_test_function, s_in);
    HW_STREAM_CLOSE(my_test_function, s_out);
}
```

Fig. 2. Software Process

The application consists of a software process (Fig.2) and hardware process (Fig.3), both created in the configuration function (Fig.4).

```
void my_function(co_stream s_in, co_stream s_out)
{
    co_int32 fib;

    co_stream_open(s_in, O_RDONLY, INT_TYPE(32));
    co_stream_open(s_out, O_WRONLY, INT_TYPE(32));

    co_stream_read(s_in, &fib, sizeof(co_int32));
    co_stream_write(s_out, &fib, sizeof(co_int32));
}
```

Fig. 3. Hardware Process

```
void config_function(void * arg)
{
    co_process my_function_proc, my_test_function_proc;
    co_stream stream_in =
        co_stream_create("stream_in", INT_TYPE(32), 8);
    co_stream stream_out =
        co_stream_create("stream_out", INT_TYPE(32), 8);

    my_function_proc =
        co_process_create("my_function",
                          (co_function)my_function,
                          2,
                          stream_in,
                          stream_out);

    my_test_function_proc =
        co_process_create("my_test_function",
                          (co_function)my_test_function,
                          2,
                          stream_out,
                          stream_in);

    co_process_config(my_function_proc, co_loc, "PE0");
}
```

Fig. 4. Configuration Function

B. Architectural Design

The program is divided into several stages (Fig. 5), which will be described in detail in this section:

- analyze the given code, to build the process graph
- gather the necessary information for each process
- partition the application into software/hardware parts

1) Code Analysis: In the code analysis stage of the partitioning, the process graph is computed. This is done by analyzing the configuration function of the input program. The output of this phase is an orientated graph *G(V,E)*, in which the vertices are processes and the edges are data streams. The process graph is built by analyzing the calls co_process_create for process creation, co_stream_create for data stream creation and co_process_config for the explicit hardware implementation. This creates an unoriented graph. To determine the orientation of every edge or data stream, we analyze the calls to co_stream_open in order to determine if a process will write to a stream or read from it. The orientation of the edge will be from the process that writes data to the stream to the process that reads data from the stream.

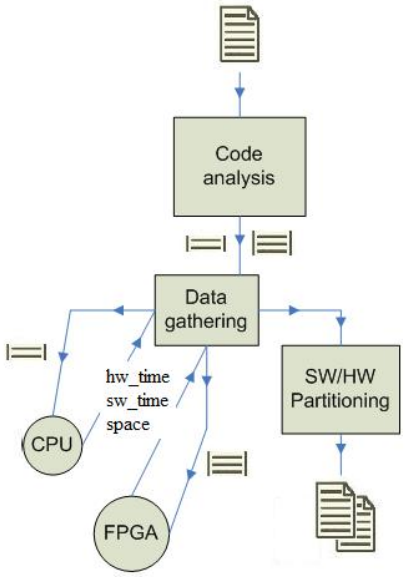

Fig. 5. Architectural Design

2) Data Gathering: The whole partitioning algorithm is based on some useful information about each process:

- software execution time
- hardware execution time
- space needed (in terms of the number of LUTs and slices occupied on the FPGA)

After building the process graph, each process is run individually on both the CPU, to obtain the software execution time, and the FPGA, to obtain the hardware execution time and the space occupied. In order to extract and run a single process from the entire application, a test function should be provided. Also, stub functions should be created to replace the missing parts that communicated with the extracted process.

3) Software/Hardware Partitioning: The software/hardware partitioner divides the code into hardware implemented processes and processes that will run on the local CPU. Most often, the quickest program will be the one that implements all its code into hardware. However, because of the space constraints, only a fraction of the program can be configured in the local hardware. Because of this, we must run the rest of the code on the available CPU. For this, we need a partitioner that will determine what code is best suited to be implemented in hardware. The partitioning algorithm is described in detail in the next subsection.

C. Partitioning Algorithm

The partitioning algorithm creates a subset of the vertex set of the process graph, denoted by *HW*: the hardware implemented processes. If *G(V,E)* is the process graph, *V - HW* will be the set of software implemented processes. The input data is taken from the data collection phase, and consists of the maximum amount of resources available on the FPGA(slices and look-up tables (LUTs)),the communication time of each process, the hardware/software execution time, and the process graph obtained by analyzing the configuration function of the input and the process functions. We assume that in the process graph there is a node which feeds input data to the application and a node which collects the results. We also assume that by removing these two vertices from the process graph, the remaining graph will be acyclic. The partitioning process must find in the solution space the best solution that fits our requirements: it must be physically implementable and must run in a minimum amount of time. Since the presented problem is a NP-hard one and for an input which consists of a considerable amount of software/hardware processes, the solution space becomes very large and an iteration through it will not complete in an acceptable amount of time. Therefore, we have chosen to search through the solution space using a genetic algorithm. The algorithm was implemented using the GAUL open-source framework. We encode the solution by using one chromosome. The chromosome contains an allele for each process. The allele values are boolean, because a process can be implemented either in hardware or software.

1) Evaluation operator: We evaluate a solution first by eliminating the chromosomes which are not physically implementable or contain processes which have been selected to be implemented in hardware but do no have such an implementation. A configuration is not phsically implementable if the total sum of LUTs and slices of its hardware and software components are grater then the available number. If one process is implemented in software, the Microblaze processor will be configured on the FPGA, and will cost hardware resources. We define this cost as lutMicroblaze it terms of LUTs and sliceMicroblaze in terms of slices. We define the hardware cost function

$$\phi$$

as such (by *P(V)* we denoted the set of all partitions of *V*):

$$\omega_{LUT} P(V) \rightarrow \mathbb{N},$$
$$\omega_{LUT}(X) = \sum_{v \in V-X} costLUT(v) \quad (1)$$
$$\omega_{SLICE} P(V) \rightarrow \mathbb{N},$$
$$\omega_{SLICE}(X) = \sum_{v \in V-X} costSLICE(v) \quad (2)$$

$$\phi : P(V) \rightarrow (\mathbb{N} \times \mathbb{N}),$$
$$\phi(X) = \begin{cases} (\omega_{LUT}(X), \omega_{SLICE}(X)) & \text{if } X = \emptyset \\ (lutMicroblaze + \omega_{LUT}(X), & \\ sliceMicroblaze + \omega_{SLICE}(X)) & \text{if } X \neq \emptyset \end{cases} \quad (3)$$

We define maxSLICES and maxLUTS the maximum amount of slices, respectively the maximum amount of LUTs of the device. If for a partition X we have

$$\phi(X) = (m1, m2) \wedge$$
$$(m1 \geq maxLUTS \vee m2 \geq maxSLICES) \quad (4)$$

then the design is not physically implementable and will be rejected as a solution.

If the solution is not rejected, we will use two functions to compute the fitness of the individual. These fitness functions approximate the total execution time. The first function, sums the execution time for all processes:

$$\alpha_1 : P(V) \rightarrow \mathbb{N},$$
$$\alpha_1(X) = \sum_{v \in V-X} costSW(v) + \sum_{v \in X} costHW(v) \quad (5)$$

The second function approximates the total execution time by adding the value of the maximum length path between a given source and destination to the sum of the execution times of all software processes that are not on this path (by *MLP* we refer to the set containing the nodes on the maximum length path):

$$\alpha_2 : P(V) \rightarrow \mathbb{N},$$
$$\alpha_2(X) = \sum_{v \in MLP-X} costHW(v) + \sum_{v \in X} costSW(v) \quad (6)$$

The maximum length path can be computed in *O(card(V)+card(E))* time, since we assumed that the graph will be acyclic (by *card(X)* we denoted the number of elements in set *X*).

2) Mutation operator: The mutation operator is implemented by changing one graph vertex from software to hardware or from hardware to software, and therefore preventing the lack of diversity in the chromosome population. The mutation operator is applied only if the mutation does not produce an unacceptable solution.

3) Crossover operator: The crossover operator creates two new solutions from two existing solutions. The new solutions will be included in the new generation in the genetic algorithm. The algorithm uses elitism in order to keep the best solutions. The crossover operator is implemented by generating a random cut of the process graph each time the operator is applied. We then combine the two solutions base on the generated cut by combining the first partition of the first chromosome with the second partition of the second chromosome and vice-versa.

V. Results

We tested our solution on an application with eight processes. The process graph of the application can be seen in (Fig. 6).

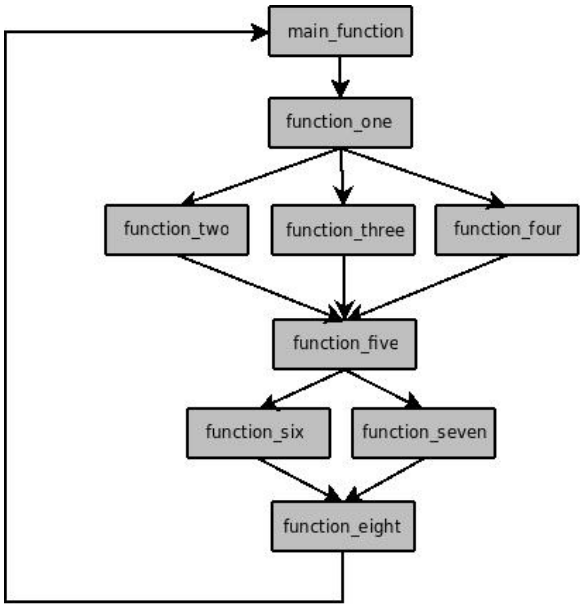

Fig. 6. Process Graph

The execution time of the data gathering process was very high, approximately 90 minutes, but it could be diminished, as we explained in the *Future Work* section. The results obtained by the data gathering process can be seen in (Fig. 7).

function	hw_time	sw_time	no_LUTs	no_slices
function_one	400	1401	1062	433
function_two	400	1400	1005	596
function_three	400	1400	1004	583
function_four	400	1400	1004	552
function_five	1200	4200	2033	1301
function_six	400	1400	1007	590
function_seven	400	1400	1007	564
function_eight	400	1400	1187	693

Fig. 7. Information about each process

The elitism rate in our genetic algorithm, for which we used a Darwinian scheme with linear selection, was 02. The

population size was 500, and the number of iterations 400. We processed the input data twice, once for each fitness function.

For the first fitness function we obtained the results shown in (Fig. 8). The set of hardware implemented processes, *HW*, was found to be: {*function_one, function_five, function_six, function_eight*}. As $function\_six$ was chosen to be implemented in hardware, it will execute in parallel with $function\_seven$, giving a total execution time of approximately 7600.

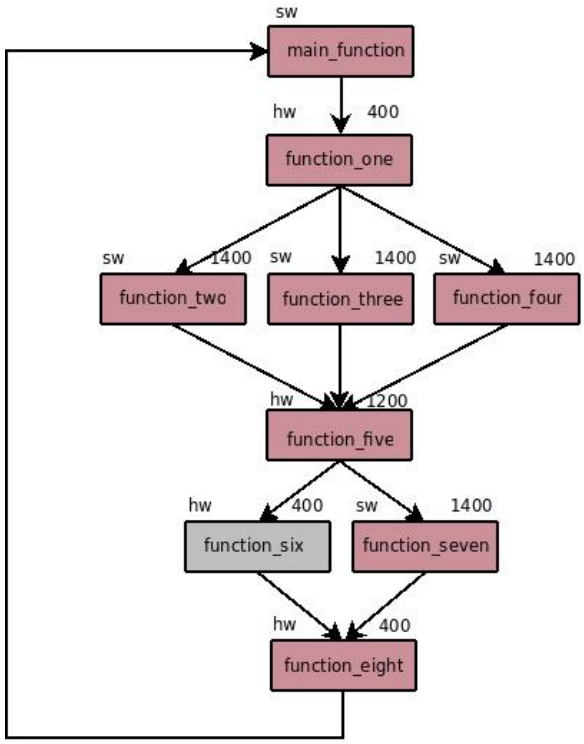

Fig. 8. Solution given by first fitness function

For the second fitness function we obtained the results shown in (Fig. 9). This time, the set of hardware implemented processes, *HW*, was found to be: {*function_two, function_three, function_five, function_six*}. As both *function_two* and *function_three* were chosen to be implemented in hardware, all three functions (*function_two*, *function_three* and *function_four*) will be executed at the same time, giving a total execution time of approximately 6800.

As expected, the function that considers the maximum length path yielded the best results, having the smallest total execution time and increasing the parallelism level of our application.

VI. Future Work

There are several possible continuations for the current work in order to fully eliminate human intervention from the partitioning process.

First, a recomposition stage can reassemble the full program after obtaining the result from the partitioner.

Second, a test generation program that will automatically generate test and stub functions for each process will greatly decrease the amount of work that is required from the user.

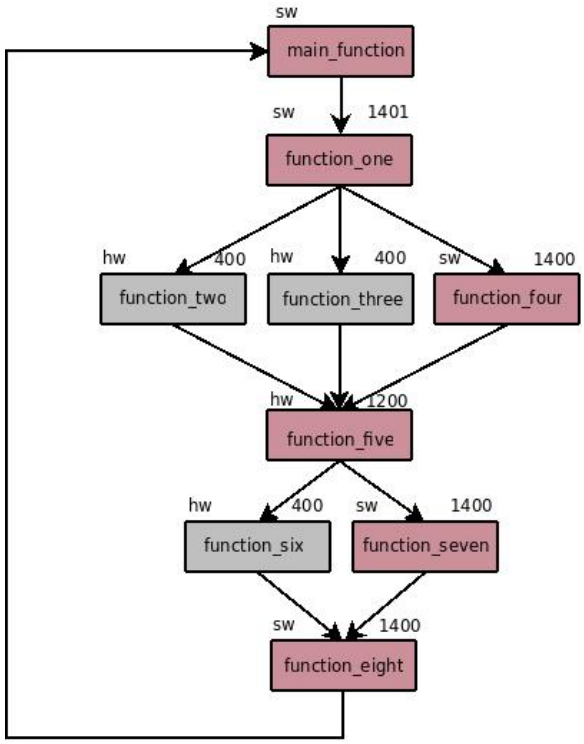

Fig. 9. Solution given by first fitness function

The third continuation possible is the transformation of software processes into functions that execute asynchronously in order to optimize the code that runs on the CPU. This is actually a transition from the process model, which best suits the hardware to a sequential model which runs much better on the CPU. This transition is easier to implement than the inverse transition, from a sequential model to a process model.

Fourth, the data gathering phase execution time must be reduced. This can be achieved by running the tests in a cluster or by empirically obtaining the process parameters.

VII. Conclusion

We have constructed a hardware/software partitioner that tries to solve the difficult problem of partitioning an application into processes that will run on a CPU and processes that will be accelerated in hardware. The algorithm used also considers the parallelization of as many parts of the application as possible.

The process of analyzing and determining the partition is fully automated. However, several additions to the architecture must be made in order to make the entire partitioning process automated. The resulting implementation can be used in practice if the execution time of the data gathering phase can be reduced. Also, the design must be tested on more architectures, for example one that has a PowerPC processor instead of the synthesized Microblaze one.

References

[1] R. K. Gupta and G. D. Micheli, "System-level synthesis using reprogrammable components," in *EDAC '92 : Proceeding of the European*

Conference on design automation. Brusseles, Belgium: ACM, 1992, pp. 2–7.

[2] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," *IEEE Des. Test*, vol. 10, no. 4, pp. 64–75, 1993.

[3] J. Henkel and R. Ernst, "A hardware/software partitioner using a dynamically determined granularity," in *DAC '97: Proceedings of the 34th annual Design Automation Conference*. New York, NY, USA: ACM, 1997, pp. 691–696.

[4] P. Arató, Z. A. Mann, and A. Orbán, "Algorithmic aspects of hardware/software partitioning," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 10, no. 1, pp. 136–156, 2005.

[5] M. Yuan, X. He, and Z. Gu, "Hardware/software partitioning and static task scheduling on runtime reconfigurable fpgas using a smt solver," *Real-Time and Embedded Technology and Applications Symposium, IEEE*, vol. 0, pp. 295–304, 2008.

[6] D. Wang, S. Li, and Y. Dou, "Collaborative hardware/software partition of coarse-grained reconfigurable system using evolutionary ant colony optimization," in *ASP-DAC '08: Proceedings of the 2008 Asia and South Pacific Design Automation Conference*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2008, pp. 679–684.

[7] G. Stitt, "Hardware/software partitioning with multi-version implementation exploration," in *GLSVLSI '08: Proceedings of the 18th ACM Great Lakes symposium on VLSI*. New York, NY, USA: ACM, 2008, pp. 143–146.

[8] I. Karkowski and R. H. J. M. Otten, "An automatic hardware-software partitioner based on the possibilistic programming." in *EDTC '96: Proceedings of the 1996 European conference on Design and Test*. Washington, DC, USA: IEEE Computer Society, 1996, p. 467.