FPGA area allocation for parallel C applications

Vlad-Mihai Sima, Elena Moscu Panainte, Koen Bertels
Computer Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
Phone: +31 15 2786249 Fax: +31 15 2784898

Abstract— **In this paper we present an FPGA area allocation algorithm for parallel OpenMP application that aim to improve performance for a specific reconfiguration area. The algorithm will be implemented in Delft Workbench, a toolchain developed at TU Delft with focus on reconfigurable architectures. The hardware platform used to gather the experimental results is a Xilinx Virtex II Pro board with a PowerPC 405 processor and an FPGA. Using profiling information and the structure of the application we construct a mathematical model which is then used by a modified ILP (Integer Linear Programming) solver to choose the optimal mapping.**

I. Introduction

With the growing need of faster, low price devices, computer systems have grown to be significantly complex and demanding for the system architects, hardware designers and software engineers. New strategies have been proposed to use efficiently the resources while new problems have been introduced that have to be solved with the given resources. Reconfigurable computing is one of the approaches that allows a wider range of problems to be solved, in a fast, but also economic and efficient way. Parallelization is another key concept, that becomes increasingly important as the frequencies of the processor reach their physical limits so the application are not speedup just by increasing the frequency of the processor. In this paper we present an algorithm that simplifies the development process by selecting, based on execution traces,the efficient area allocation for the reconfigurable hardware in context of parallel algorithms.

In the context of embedded devices one of the most used languages remains C programming language. To allow programmers to write parallel applications while using most of their previous programming experience, one of the common solution is to use shared memory parallel programming paradigm. OpenMP is such an API that allows simple, portable and scalable shared memory programming, which is widely used by industry and research community.

The paper is organized in seven sections. We present the background and related work in the following section. The problem is outlined and a motivational example is presented. The allocation algorithm is presented in detail in section 4. Finally we present some results, the conclusion and future work.

II. Background and related work

The development of the algorithm presented is based on the MOLEN programming paradigm [1], which for Field-programmable Custom Computing Machines (FCCMs). This paradigm addresses a general purpose processor (GPP) and a reconfigurable hardware (usually FPGA). The reconfigurable component is controlled by special instructions for reconfiguration and the execution of hardware components of the application.

Previous work related to area allocation in reconfigurable system has been done, but it was focused on sequential code without parallel sections. The first algorithm presented in [2] tries to determine the best partitioning between a fixed and a reconfigurable part using ILP, while the second algorithm presents a partitioning between fixed, reconfigurable and software execution.

Other research has focused on determining the optimal number of kernels that have to be configured considering the memory bandwidth [3].

III. Motivational example

As a motivational example we use a simple application which was constructed to capture the patterns from an application used for audio processing. The real application uses a microphone array to amplify the signal from one source while suppressing signals from others - process called beamforming. Assuming the reduced C application:

```
proc_A();

#pragma openmp parallel
```

```
for(i=0;i<8;i++) {
  kernel_1();
}

proc_B();

#pragma openmp parallel
for(i=0;i<4;i++) {
  kernel_2();
}
```

We notice that there are several possibilities of execution. Let's suppose we have the area and execution times as in table 1.

TABLE I
VALUES FOR AREA AND EXECUTION FOR SAMPLE CODE

Function	Area	Execution time		Reconf time
		SW	HW	
proc_A	-	60	-	-
kernel_1	20%	14	2	30
proc_B	-	20	-	-
kernel_2	30%	8	3	45

For the software and hardware features of the application presented in Table I we describe two basic scenarios. Our algorithm will try to determine automatically the best scenario.

Scenario A

The first kernel is configured 4 times in hardware, so the execution is the one depicted in Figure 1. With this configuration $kernel_i$ kernels are executed in parallel thus the total execution time is 176 cycles.

Scenario B

The first kernel is configured 2 times in hardware and the second kernel is configured 2 times in hardware, so the execution is the one depicted in Figure 2. In this scenario the total execution time is 128. Fro the multitude of possible scenarios our algorithm selects the efficient solution using ILP.

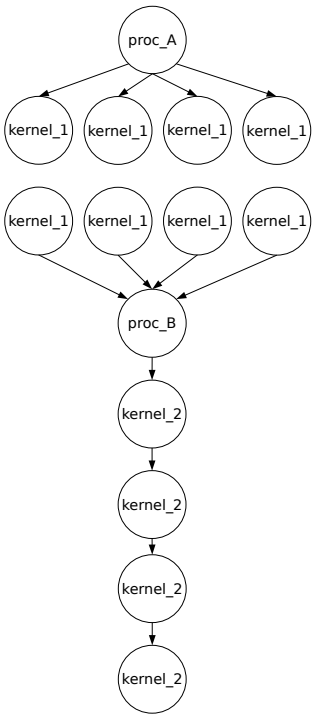

Fig. 1. Execution of sample in scenario A

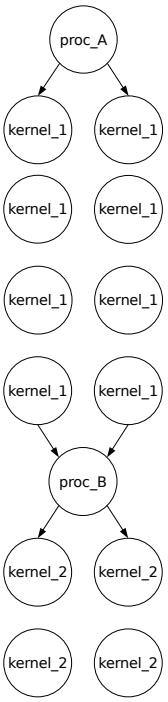

Fig. 2. Execution of sample in scenario B

IV. Problem overview

We will denote by *kernel* any function that is executed inside of a parallel loop and consumes a significant part of execution time from the sequential program execution.

The kernel can be implemented in hardware (using manual or automated VHDL generation) usually resulting in improved execution times. However, the area of the FPGA is limited and also the reconfiguration time is considerable compared with the execu-

tion time. The problem is to determine the number of kernels for the reconfiguration, and the points in the program such that the overall execution time is improved.

The known factors that are taken into consideration are:

- area of the kernels
- execution time of one kernel in software and hardware
- configuration time of a kernel
- an execution trace (or static profiling execution trace, in which case the results will not be as good)

V. Algorithm

For the kernels executed on the reconfigurable hardware, the software execution times is usually orders of magnitude ? than the hardware execution time. Based on this feature, in order to reduce the algorithm complexity. we consider that the hardware execution in parallel with the software execution does not significantly increase the application speedup when compared with exclusive parallel execution in hardware. A motivation for this is presented in thefollowingg.

Let's consider a kernel that can be executed in hardware in time h and in software in time s and is executed in the application in a loop with n iterations, without data dependencies. Let x be the number of execution in software, y the number of executions in hardware, and p the number of instances configured then we have the following equations:

$$x + y * p = n \tag{1}$$

$$x * s - y * h < h \tag{2}$$

The second equation states that the difference between the software execution time performed in parallel with the hardware and the hardware execution time should not exceed one hardware execution. Otherwise more tasks should be executed in hardware. From the above equations we can compute the following equation for the number of hardware executions:

$$y > \frac{n * s - h}{s * p + h} \tag{3}$$

In order to achieve parallel software and hardware execution the number of hardware executions is:

$$y = \lceil \frac{n * s - h}{s * p + h} \rceil \tag{4}$$

For the purposes of the computations we can approximate y to be:

$$y \simeq \frac{n * s - h}{s * p + h} \tag{5}$$

If we approximate the total execution time of the kernel with the time needed for the hardware executions and compute the speedup compared to the case where all executions are in hardware we obtain:

$$speedup = \frac{\lceil \frac{n}{p} \rceil * h}{y * h} \simeq \frac{\frac{n}{p}}{\frac{n*s-h}{s*p+h}} = \frac{n * (s * p + h)}{p * (n * s - h)} \tag{6}$$

We are interested in the significant gain that could be obtained in the software-hardware scenario. This is obvious for the case where the parallelism p is minimum so we consider $p = 1$. We also must make an estimation regarding the speedups of kernels. From previous work [4] we conclude that the range of speedups for kernels that will be candidates for kernel implementations is between 2 and 10. We summarize the gain that could be obtained for some speedup values using the following formula:

$$speedup \simeq \frac{n * (\frac{s}{h} + 1)}{(n * \frac{s}{h} - 1)} \tag{7}$$

TABLE II
Possible speedup considering software/hardware parallel execution for a specific kernel

kernel speedup	n	speedup
3	32	1.35
3	64	1.34
6	32	1.17
6	64	1.17
10	32	1.1
10	64	1.1

For Table II, we notice that the speedup due to parallel execution between hardware and software is not significant. moreover is we consider $p = 2$, the speedup becomes even lower in the range of 1.05 to 1.17. In conclusion we consider that the parallel execution between software and hardware is not a key issue for our study.

In order to solve the problem presented in Section 3 we use Integer Linear Programming (ILP) to find an efficient solution to the above problem.

An ILP problem represents a mathematical model composed of: variables, constraints and objective function. To be able to map our problem to the above mathematical model, we use as a starting point a simplified control flow graph. The simplified control flow graph is constructed as follows:

- start from the call graph of the application
- replace each function with it's control flow graph (in case of recursive functions or loops in the call graph eliminate those functions). Keep special nodes for:
  - OpenMP parallel pragma-s
  - kernel function calls
- collapse all nodes of the graph that do not contain parallel nodes, or hardware nodes, while accumulating the profile information

All the reconfigurations for the hardware functions must be done at the start of software or hardware nodes. The decision that has to be taken is in which nodes and in what numbers the hardware kernels must be configured.

For multiple level of parallelism (i.e. imbricated OpenMP parallel pragma-s) the inner levels must be 'transformed' to by multiplying their loop counts with the outer level loop count.

For each kernel we construct multiple *webs* - collections of nodes in the graph. Webs are constructed in the following way: for each node in which there is a kernel execution (which we will call from now the *root kernel* of the web) we apply the following steps:

- start from the analyzed node
- if there is a path from the node to another kernel of the same type, then the current web and that path generates another web
- for each dominator node of the analyzed node, the current web and all the nodes from the dominator node to the analyzed node generate another web
- make all the graph another web

A web has the following properties:

- there is at least one node that dominates the kernel execution
- one of the predecessors of the dominator node is a kernel execution or the beginning of the program
- one of the successors of a node that belongs to the web a kernel execution or the end of the program
- the web includes all the nodes that do not use the hardware differently - ie if there is another kernel execution of the analyzed kernel this will be included in the web

A. Variables

For each identified web i in a parallel section with n executions we will have the following variables:
- SW_i - if 1 this means that all kernels of the type of the root kernel contained in that web will be executed in software
- HW_i - represents the number of implementations in hardware of the root kernel contained in that web i (in the entry point)

B. Constraints

The constrains of our ILP problems address:
- area constraints
- hardware/software execution
- time constraints

The area constraints should guarantee that the solution obtained from the ILP will be implementable on the board, ie. at any given node in the SCFG the area used by the kernels configured is not greater than the area of the reconfigurable fabric. Let p be a node where the hardware configuration can be changed (the reunion of the dominator nodes for all the webs). We will have the set of webs of interest defined as:

$$X = \{\forall w \in W / p \in w\} \tag{8}$$

where W is the set of all webs

With this definitions we can express the constraint as:

$$\sum_{i \in X} (HW_i * A_i) <= A \tag{9}$$

Where A_i represents the area occupied by one root kernel of web i. The hardware/software execution constraints ensures that if the algorithm decides for software execution there will be no hardware executions and viceversa. Let p be a node that contains a kernel execution. The set of webs of interest is defined as:

$$Y = \{\forall w \in W / p \in w \wedge p \text{ is root kernel of } W\} \tag{10}$$

With this definition we can express the constraint as:

$$\sum_{i \in Y} (SW_i + \frac{HW_i}{n_p}) <= 1 \tag{11}$$

$$\sum_{i \in Y} (SW_i + HW_i) > 0 \tag{12}$$

where n_p represents the number of iterations for the kernel execution.

The time constraints guarantee that the reconfiguration overhead does notnegativelyy affect the total execution time. For this purpose, we first determine the unfeasible webs - the reconfiguration done in the first node and the execution overcomes the software execution time. There are webs and values for which this can't be determined as there can be other hardware or software executions. For all the paths in the web i from the dominator to the execution we write the following equations (one for each of the iterations that could be implemented)

$$T_{hw} \leq T_{path} + n_i * t_{sw_i} \tag{13}$$

In both T_{hw} and T_{path} we have to include the time of the execution in hardware. For the current FPGA, T_{hw} can be safely approximated to the hardware configuration time.

The total time on the path T_{path} is defined next:

$$\sum_{s}^{SW_p}(t_{sw_s}) + \sum_{e}^{E_p}(\sum_{x}^{X}(SW_x * t_{sw_x} + t_{conf_x} * HW_x)) \tag{14}$$

where SW_p is the set of software nodes along the path, E_p is the set of executions along the path and X is the set of webs containing execution *exec*.

C. Objective function

The objective is to minimize the time in which the application is executed, but we must take into account the fact that several execution paths will be taken. So the function represents the sum of all execution times (hardware or software) scaled by a factor f_i which represents the percent from the total execution time spent in the kernel associated with web i.

$$min(\sum_{i=1}^{N}(t_{sw_i} * SW_i - t_{hw_i} * HW_i)) * f_i) \tag{15}$$

VI. Results

The web-s constructed for the example from Section 2 are depicted in Figure 3. Web 2 is considered to be the web for the execution of kernel_1 and containing the whole control flow graph, while web 5 is considered to be the web for the execution of kernel_2 and containing the whole graph. Applying the algorithm we obtain the following constraints:

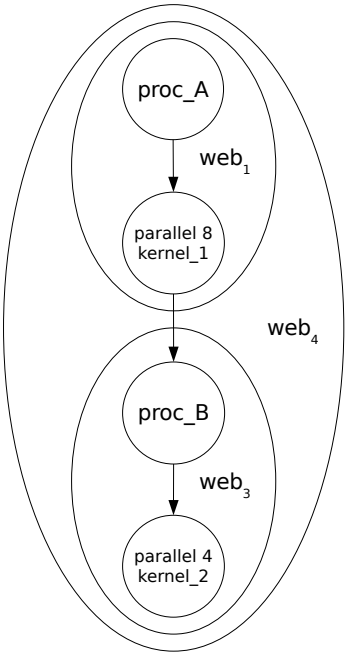

Fig. 3. An example SCFG with web-s

$$\begin{aligned}
20 * HW_1 + 20 * HW_2 + 30 * HW_4 + 30 * HW_5 &\leq 100\\
30 * HW_3 + 30 * HW_4 + 30 * HW_5 + 20 * HW_2 &\leq 100\\
SW_1 + HW_1 + HW_2 &> 0\\
SW_2 + HW_3 + HW_4 + HW_5 &> 0\\
SW_1 + 0.125 * HW_1 + 0.125 * HW_2 &\leq 1\\
SW_2 + 0.25 * HW_3 + 0.25 * HW_4 + 0.25 * HW_5 &\leq 1\\
30 * HW_1 - 60 - 112 * SW_1 &\leq 0\\
45 * HW_3 - 20 - 112 * SW_1 -&\\
32 * SW_2 - 30 * HW_1 &\leq 0\\
45 * HW_4 - 80 - 14 * SW_1 -&\\
32 * SW_2 - 30 * HW_1 &\leq 0
\end{aligned}$$

The objective function (we have computed $f_1 = 0.5$ and $f_2 = 0.14$) is:

$$\begin{aligned}
min(56 * SW_1 - 1 * HW_1 - 1 * HW_2&\\
+4.48 * SW_2 - 0.42 * HW_3&\\
-0.42 * HW_4 - 0.42 * HW_5)&
\end{aligned}$$

Solving the equations with a ILP solver we obtain the following solution:

$$SW_1 = 0 \quad HW_1 = 0 \quad HW_2 = 3$$
$$SW_2 = 0 \quad HW_3 = 0 \quad HW_4 = 1$$
$$HW_5 = 0$$

The interpretation of the solution is the following: kernel_1 will be implemented in the FPGA 3 times, before the application start, while kernel_2 will be implemented in the first node of web_1. The total execution time for this solution is 98.

VII. Conclusion and future work

Parallel algorithms pose a new challenge to compilers and optimizers because of the exploding search space for mapping, scheduling and allocation. Our algorithm, while making some conservative assumptions at cost of performance, does provide a solution to the scheduling and allocation problem, that takes into account the most important factors involved - area and execution time.

The limitations of the current algorithm can be eliminated by considering multiple levels of parallelism in the control flow graph and the memory bandwidth impact of executing in parallel several instances of the same kernel. The reconfiguration capabilities of the device should also be taken into consideration, for example parallel configuration or configuration of heterogeneous fabric.

References

[1] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The molen polymorphic processor," *IEEE Transactions on Computers*, pp. 1363– 1375, November 2004.

[2] E. M. Panainte, K. Bertels, and S. Vassiliadis, "Compiler-driven fpga-area allocation for reconfigurable computing," in *Proceedings of Design, Automation and Test in Europe 2006 (DATE 06)*, March 2006, pp. 369–374.

[3] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Parlgran: parallelism granularity selection for scheduling task chains on dynamically reconfigurable architectures," in *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation.* New York, NY, USA: ACM Press, 2006, pp. 491–496.

[4] Y. D. Yankova, G. Kuzmanov, K. Bertels, G. N. Gaydadjiev, J. Lu, and S. Vassiliadis, "Dwarv: Delftworkbench automated reconfigurable vhdl generator," in *In Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL07)*, August 2007, pp. 697–701.