Compiler and OpenMP framework to allow dynamic hardware allocation on reconfigurable platforms

Vlad-Mihai Sima, 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 the compiler and OpenMP runtime library extensions needed to allow runtime decisions regarding area allocation on a reconfigurable platform in a multi application context. Using a strong interaction between the operating system and the compiled applications, our framework will allow operating system algorithms to decide which allocation fits best the current environment. System calls will be inserted at essential point in the application, to inform the operating system about the execution status, and to obtain the decisions done at operating system side. We will profile and analyze the overheads introduced. For estimating the results obtained with this framework we will use a beamforming application The testing platforms is based on Xilinx Virtex 4 ML403 development board running Linux 2.6 and supporting MOLEN programming paradigm.**

I. Introduction

As computer systems become more complex there is also a demand for faster and smaller devices. One possible solution is to use reconfigurable which allows designs to investigate new strategies using limited resources. Reconfigurable computing can be especially used in the context of parallel applications, as the hardware implementation can adapt to different levels of parallelism at runtime.

In this paper, we propose modifications to the compiler and runtime OpenMP library for executing some of the functions in parallel. We do this in conjunction with the MOLEN paradigm, which shows its flexibility in various contexts.

The paper is organized as follows: in Section II we briefly present the MOLEN programming paradigm for reconfigurable architectures and related work. Next, we give a real motivational example and also present the problem overview. A detailed description of the proposed modifications is given in Section IV. Some profilling results are shown in Section V, with a comparison to other related possibilities and analyses of the factors that influence the results. In Section VI, we present conclusions and outline new research directions.

II. Background

The programming model for a reconfigurable platform must offer an abstraction of the available resources to the programmer. The role of the **MOLEN programming paradigm** [1] is to abstracts the hardware and allows the programmer and the compiler to use efficiently the underlying hardware. The extension need to the existing system is minimal involving a set of fixed instructions or operating system calls that allow the designer to use any number of hardware units in any configuration. To support parallel execution, the minimal extension include the following primitives: SET, EXECUTE, BREAK and MOVTX and MOVFX.

OpenMP is set of compiler directives, library functions and environment variables that can be used to specify parallelism in applications that use shared memory architecture. Because of this OpenMP is the obvious choice for describing parallelism in a Molen paradigm context.

Run-time solutions for allocation are presented in [2] from the services point of view, without relating too much to the actual implementation.In [3] an algorithm based on early partial reconfiguration and incremental reconfiguration is presented but without involving parallelism.

III. Motivational example

With the increase of available parallelism more and more applications use OpenMP to specify what sections of the code can benefit from parallelism. One of the most common uses is applying the same computations on multiple vectors, if there is no data dependencies between the computations or the results. Assuming we have such a case in an sound processing application, where filters must be applied on waveforms coming from multiple microphones as in Figure. 1.

We observe that the code contains two annotations. The first one *#pragma omp parallel for* is a pragma defined by the OpenMP standard and means that the iterations of the next loop can be executed in parallel - ie. there are no data dependencies between them.

The second annotation, *#pragma molen 1*, is a pragma specific to the MOLEN programming paradigm and specifies that the *filter* function should be run on the reconfigurable fabric and it is identified as the operation *1*.

Even if with OpenMP you specify how many threads are created for a specific parallel section this is not recommended. The reason is that by allowing the runtime system to decide the number of threads the application is written in a portable way - as it doesn't make any assumption on the number of

```
int func(int **in, int **out, int l)
{
  int i;
#pragma omp parallel for
  for(i=0;i<l;i++)
  {

#pragma molen 1
     filter(in[i],out[i]);
  }
}
```

Fig. 1. Example code

threads - and also the best number of threads can be chosen, depending on the overall system situation.

Assuming the implementation fits only once on the available reconfigurable fabric, we need to use just one thread so the code generated will be equivalent with the code generated for the C program in Figure. 2. (that is, without any OpenMP constructs).

```
int func(int **in, int **out, int l)
{
  int i;
  for(i=0;i<l;i++)
  {
    molen_movtx(1,0,in[i]);
    molen_movtx(1,1,out[i]);
    molen_execute(1);
    molen_break(1);
  }
}
```

Fig. 2. Example code

The parameters for *molen_movtx* are: the identifier of the implementation, the parameter number and the parameter value. For the other two molen functions the only parameter is the identifier of the operation.

Obviously in case we have several threads we can't use the above code as for each thread another instance of the implementation has to be run. So, our **problem** can be formulated as follows: given an annotated C code that contains both OpenMP and MOLEN pragmas, information about the size of implementations for the functions that can be executed in hardware and the total size of the reconfigurable fabric, instantiate as many as possible hardware modules and run them from the software using the MOLEN operating system calls.

IV. Implementation

An overview of our framework is presented in Figure. 3. We will present next each of the important elements.

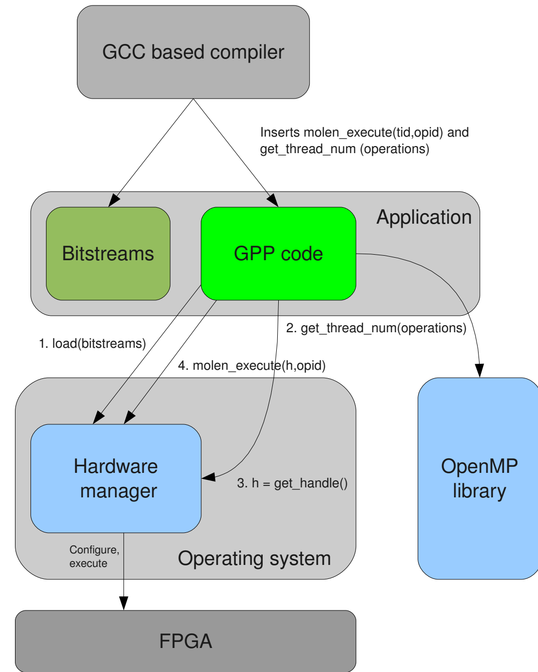

Fig. 3. Framework for executing using OpenMP and MOLEN

The **hardware manager** is a part of the operating system and has the following functions:

- loads the bitstreams available from the application elf file. This is done when the applications calls a system call with parameter an array with bitstream addresses and the operation id for those bitstreams. This is needed as a specific operation could be placed at various positions inside the reconfigurable area. Doing the relocation of hardware tasks at runtime would be time consuming so we decided to provide the hardware manager a set of possibilities. The system call signature is *molen_load_bitstreams(bitstream *p, int n)*.
- return the maximum number of instances that can be configured on the hardware for a specific *opid*. This has to take into account the current state of the hardware, and the various placing options associated with those operations. The call for this is *molen_get_max_operations(int *p, int n)*, where *p* is an array of operation id-s that will be used in the parallel section.
- give a handle to an instance of a specific operation. Further invocation with the same id will provide another instance, if this is possible. The call to get a handle is *int molen_create_operations(int *p, int n)* and the call to deallocate the handle is *int molen_destroy_operations(int h)*
- send parameters to a specific operation using a handle. The function prototype is *int molen_movtx(int h, int par, int value)*, where *par* is the parameter number and *value*

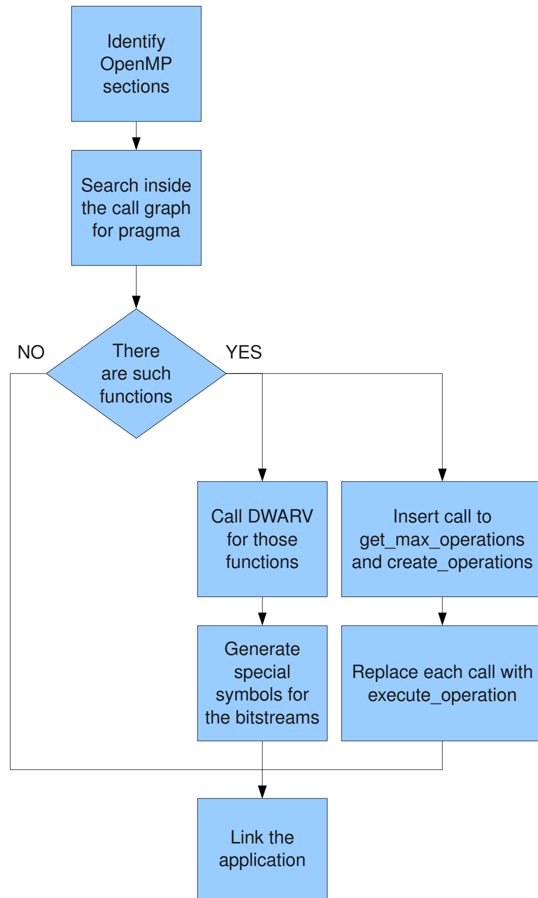

Fig. 4. Compilation flow

is it's value

- execute using a handle returned by the hardware manager. The function is *int molen_execute(int h, int opid)*
- waits for the execution of a certain instance to finish. The function is *int molen_break(int h, int opid)*

The **compiler** contains several modifications in order to allow the execution of multiple CCU-s in OpenMP context. The flow of the operations is described in Figure. 4

The first step is to identify the OpenMP sections and to search for functions marked with MOLEN pragma. This is done now within the scope of a compilation unit (a .c file) but this can be extended in the future. After all the pragmas have been identified a list of needed operations is constructed. Multiple invocations of the same operation are ignored, as the code in a section should be sequential, so one hardware instance can be reused multiple times.

Instead of the normal mechanism used by the OpenMP library to determine the number of threads - create as many threads as processors in the system - our framework will provide a special function that will determine that number - *omp_molen_get_thread_num(int *op, int n)*. This function receives as parameter the operation list computed in the previous step, and it's purpose is to determine, using the hardware manager, how many operations can be configured and run in parallel. This function uses *molen_get_max_operations*.

The next step is to add in the parallel sections a local *handle* variable, initialised with the result of calling *int molen_create_operations*. This handle will be used at each call place to identify which instance shall be used by the thread.

The last step is to replace each call of a function with *int molen_execute_operation* with the *handle* created and the operation id.

Using the example given in Section III the code generated will be the one in Figure: 5. We can identify almost all of the steps from Figure. 3 on this code: the second step is contained in *GOMP_loop_runtime_start* - which has the role of assigning work to each of the threads. The third step is represented by *molen_create_operation* which will create the necessary handle. The last step is *molen_execute*.

```
int func(int **in, int **out, int l)
{
  int ops[]={1};
  long i, _s0, _e0;
  if (GOMP_loop_runtime_start (0,
                    n, 1, &_s0, &_e0))
  do {
    h = molen_create_operations(ops);
    for (i = _s0, i < _e0; i++) {
      molen_movtx(h,1,0,in[i]);
      molen_movtx(h,1,1,out[i]);
      molen_execute(h,1);
      molen_break(h,1);
    }
  } while (GOMP_loop_runtime_next (&_s0, _
                     &e0));
  molen_destroy_operations(h);
  GOMP_loop_end ();
}
```

Fig. 5. Example code

V. Profilling Results

We target an **architecture** similar to Xilinx Virtex series. For profilling we used Xilinx Virtex 4 inside the development platform ML410 using a Linux 2.6 kernel. The PowerPC processor runs at 300 Mhz and the hardware designs are clocked at 100 Mhz. We implemented and profilled part of the above framework as work continues to allow parallel execution of CCU-s (Custom Computin Units). To assess the overhead introduced by the framework we measured the time need for the execution of a single *filter* function and compared with the overhead imposed by the operating system calls and thread switching.

TABLE I
VARIOUS OVERHEADS

Overheads	Time (μ sec)
Thread creation	2000
Thread switch	600
Operating system call	20

TABLE II
PROFILE RESULTS FOR ONE ITERATION

	All SW	1 instance	2 instances	3 instances
Execution (msec)	2521	10324	6962	6754
Overheads	-	14%	36%	60%
Speedup vs SW	-	17.21	25.52	26.31

The results were obtained using a beamforming application. The idea is to enhance the capabilities of sensors - in our case microphones - by jointly taking the individual signals of multiple sensors into one computation and thereby modify their spatial directivity. The computation intensive part is composed of a filter operations filter, executed in parallel for all the sources. We present in Table. I the gathered data.

The application process a stream of audio so we will be interested just the overheads occurring at each iteration and not the overheads due to the initialization, as on a long period of time these will become an insignificant part of the total execution time. As *libgomp* (the library implementing OpenMP runtime) creates the thread team at program start we can ignore the *Thread creation overhead*, Of course there must be as many threads as the maximum number of operations that can fit in parallel on the reconfigurable device.

In one iteration of the execution there are 3 parallel regions with 16, 4 and 16 parallel loops, all calling *filter* function. The hardware implementation, generated using DWARV [4] C-to-VHDL compiler is 20 times faster than the software only version (profilled on Virtex4 platform). We give in the Table II summarized estimated results for different number of instances implemented in hardware.

In the table, we consider overhead thread switchting (we need as many thread switches as the number of threads per parallel region) and system calls (for transferring parameter, execution and getting back the result). We can see that, as we create threads, the overheads becomes the dominant part of the time spent in the parallel section.

We can see that the speedup increases significantly as we implement more iterations in hardware and this would scale as the number of microphones will increase - because the number of parallel iterations will also increase.

VI. CONCLUSION

In this paper, we presented a framework that allows executing CCU-s using MOLEN programming paradigm from an OpenMP context taking into account the total area available for configuring the CCU-s. We have made measurements and estimated that our framework can support efficient execution of programs in such a context. Several improvements are possible among which we enumerate: determine the maximum number of parallel iterations taking into account the ratio between the gain and the overhead, extend the framework with a mechanism to allow an efficient partition of the area in the case of multiple applications.

REFERENCES

[1] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, “The molen polymorphic processor,” *IEEE Trans. Comput.*, vol. 53, no. 11, pp. 1363–1375, 2004.
[2] D. A. K. . G. B. W. Martyn A. George, Mathew J. Pink, “Efficient allocation of fpga area to multiple users in an operating system for reconfigurable computing,” in *In Proceedings of ERSA*, 2002, pp. 238–242.
[3] B. Jeong, S. Yoo, S. Lee, and K. Choi, “Hardware-software cosynthesis for run-time incrementally reconfigurable fpgas,” in *ASP-DAC ’00: Proceedings of the 2000 conference on Asia South Pacific design automation*. New York, NY, USA: ACM, 2000, pp. 169–174.
[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.