Message Passing Interface Support for the Runtime Adaptive Multi-Processor System-on-Chip RAMPSoC

Diana Göhringer[1], Michael Hübner[2], Laure Hugot-Derville[1], Jürgen Becker[2]
Fraunhofer IOSB, Germany[1]
ITIV, Karlsruhe Institute of Technology (KIT), Germany[2]
{diana.goehringer, laure.hugot-derville}@iosb.fraunhofer.de[1]
{michael.huebner, becker}@kit.edu[2]

Abstract— **Parallel processor architectures are a promising solution to provide the required computing performance for current and future high performance applications. Certainly, the impact on the computational power of such a parallel computer system is related to the inherent parallelism of the algorithm to be implemented. The implementation of an algorithm onto a parallel computer architecture, requires from the developers a good knowledge of the underlying hardware in order to exploit the effect of the parallelization most beneficial. In order to hide as good as possible the complexity of the hardware from the developers, novel programming languages for parallel computers were developed. For example the programming models CUDA, OpenMP, OpenCL, Open GL and MPI are targeting novel multiprocessor system-on-chip architectures like the Intel Single Chip Cloud Computer with 48 cores or the Nvidia Tesla processors with hundreds of processor cores. If a new hardware architecture is invented and developed, it is always beneficial to follow standards in programming models in order to keep a compatibility to already developed programs. A novel runtime adaptive multiprocessor system-on-chip is the RAMPSoC. RAMPSoC combines the benefits of multiprocessors and reconfigurable hardware in one system and is therefore of high importance for future system design. In order to align the RAMPSoC approach to current standards, a support for Message Passing Interface (MPI) was included recently. This important step allows now to re-use already existing source code written with MPI extensions on a runtime adaptive platform.**

Keywords- Message Passing Interface (MPI); MPSoC; FPGA; Reconfigurable Computing; Runtime Reconfiguration; Network-on-Chip (NoC)

I. INTRODUCTION

Efficient process intercommunication on Multi-Processor System-on-Chip (MPSoC) is a crucial requirement, if algorithms were implemented on numerous processor cores in a parallel computer system. Since multiprocessors, like e.g. the Intel Single Chip Cloud (Intel SCC) Computer [1] with its 48 x86 compatible processors or the Nvidia Fermi processor with 512 CUDA cores, became state-of-the-art, researchers from industry and academic investigate on novel programming models and languages. The general goal is to hide the hardware complexity from the user by simultaneously keeping traditional standards like C or C++. Certainly these programming standards need to be extended by methods, which allow to exploit the parallelism on the underlying hardware. However, the communication and task distribution is hidden by a library of functions which are standardized in order to enable a re-use of software on different multiprocessors. Another important requirement of the programming model is to enable and support scalability of the underlying hardware. No developer will accept the burden to re-design the proven software in case that the number of processors for this application will vary from one series of a multiprocessor chip to the next one. Exactly this argument encouraged the developer of the RAMPSoC approach to support also a well established programming language. RAMPSoC, which especially benefits through a runtime scalable architecture, definitely requires such programming models. Since the RAMPSoC approach is based on a distributed memory model, the choice was to support MPI (Message Passing Interface) [2]. As RAMPSoC provides a specialized, highly flexible and runtime adaptive Network-on-Chip, the integration suitable to the hardware was created and developed fully new, but provides the identical methods on the programming layer, which allows to start immediately any MPI compatible software. The following sections in this paper describe the process of integrating MPI on RAMPSoC and show the benefit with an application example from bioinformatics. The paper is organized in the following manner: In Section II related work is presented. Section III presents the hardware architecture and the design methodology of RAMPSoC. The design and implementation of the MPI support for RAMPSoC is described in Section IV. The application integration and the results of RAMPSoC-MPI using a well-known bioinformatics algorithm programmed with the MPI standards are presented in Section V. Finally, the paper is closed by presenting the conclusions and future work in Section VI.

II. RELATED WORK

MPI is the programming standard used for describing a parallel program for a multiprocessor system with a distributed memory, e.g. computer clusters, supercomputers and parallel computers.

TABLE I. COMPARISON OF DIFFERENT MPI IMPLEMENTATIONS AGAINST RAMPSOC-MPI

	OpenMPI [3][4]	MPICH [5]	TMD-MPI [6]	SoC-MPI [7]	RAMPSoC-MPI
Availability	Open-Source	Open-Source	Proprietary	Proprietary	Proprietary
Code size MPI layer	25 MB	7 MB	9 KB	13 KB	37 KB
Code-size all layers	40 MB	47 MB	--	--	43 KB
Number of supported MPI standard commands	300	300	11	6	18

There exist several different implementations for MPI. TABLE I. shows the most well known implementations, like OpenMP [3][4] and MPICH [5], and two MPI implementations for embedded systems and compares them against the in this paper described RAMPSoC-MPI implementation.

OpenMPI supports around 300 MPI standard commands and is available as open source. The drawback is the huge code size of 47 MB, which is not feasible for an embedded system like RAMPSoC.

MPICH is also available as open source and it supports as well 300 commands. Also here the drawback is the huge code size of 40 MB, which is required for this implementation.

TMD-MPI [6], on the other hand, was especially designed for embedded systems. Therefore, it only requires 9KB of memory. The drawbacks are that it is a proprietary implementation and that it only supports 11 MPI commands.

Finally, SoC-MPI [7] is another example for a lightweight MPI implementation for embedded systems. It is proprietary and requires 13 KB of memory. It supports only 6 MPI commands, which are fewer functions than TMD-MPI.

In summary, none of these implementations fulfill all the requirements of the RAMPSoC system, which are a lightweight implementation, the support of sufficient MPI standard functions to port existing MPI applications onto RAMPSoC and finally the support of the runtime adaptive Network-on-Chip called Star-Wheels. Therefore, a custom MPI implementation called RAMPSoC-MPI was developed. It supports the most frequently used 18 MPI standard commands and requires only 43 KB of memory. The implementation is divided into separate layers. This way it can be easily ported to other MPSoCs and other communication infrastructures.

III. RAMPSOC

RAMPSoC [8] was designed to provide the flexibility and performance needed for embedded high performance computing applications, such as image processing in surveillance systems. As shown in Figure 1. , RAMPSoC is an MPSoC with a distributed memory approach consisting of a combination of heterogeneous processors and finite state machines (FSM). The processors as well as the FSMs can be closely coupled with one or several hardware accelerators. Different communication infrastructures like point-to-point, buses, network-on-chips (NoC) or a hybrid of these are supported. It further provides a scalable, heterogeneous and runtime adaptive NoC called Star-Wheels Network-on-Chip [9], which is described in detail in the next subsection. Figure 1. shows how the processing elements of RAMPSoC are connected over an incomplete version of the Star-Wheels NoC.

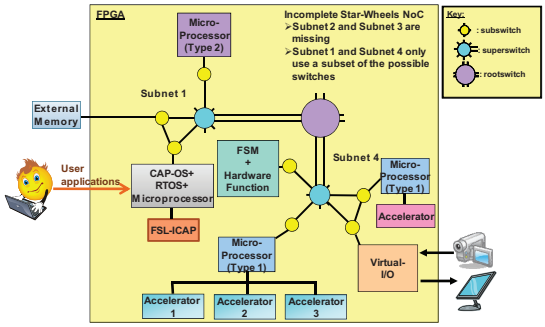

Figure 1. RAMPSoC architecture at one point in time with a incomplete Star-Wheels Network-on-Chip [9].

The processors, the accelerators and the communication infrastructure can be adapted at runtime to the requirements of the application. For controlling and supervising the runtime adaptation a special purpose operating system called CAP-OS [10] (Configuration Access Port-Operating System) has been developed. As shown in Figure 1. CAP-OS receives task graphs of the partitioned application from the user together with the partial bitstreams of the processors and the accelerators and the compiled software programs for the processors. All data (task graphs, partial bitstreams and the compiled software programs) has been generated using the novel semi-automatic design methodology of the RAMPSoC approach [11]. CAP-OS is responsible for the runtime scheduling of the tasks to the available processors and for managing the available hardware resources in such a way that the real-time requirements are fulfilled, while the used hardware resources and therefore the power consumption are minimized.

A. *Star-Wheels Network-on-Chip*

To provide an efficient and flexible communication structure for runtime adaptive MPSoCs for dataflow intensive applications such as image processing the Star-Wheels Network-Chip was developed. It has to fulfill several requirements. First, it needs to support the runtime adaptation of the PEs by recognizing, if a PE has been exchanged, added or removed at runtime. Second, it needs also to support the runtime adaptation of the network. This means the topology and the number of switches need to be adaptive. Therefore, the network needs to be modular and scalable. It should be organized in a decentralized fashion. This means, there is no centralized arbiter. Each switch decides on its own over the

routing strategy. Furthermore, each switch can check the existence and the addresses (ID number) of its direct neighbors. The size of the switches has to be reasonable in order to be area efficient for FPGAs. As the network will be used for runtime adaptive MPSoCs, it should support different clock domains. Moreover a low latency and a high data throughput are demanded by the target applications. To get a good tradeoff between area and performance constraints a novel heterogeneous topology as shown in Figure 2. was developed. All shown connections are bidirectional. It uses the novel Wheel topology to provide many parallel communication channels within each subnet. For communication between different subnets the Star topology is used. The Wheel topology combines the benefits of the ST Spidergon [12] and the star topology. This means, that the routing algorithm is much simpler compared to the Spidergon and the central switch is not as complex and high in area utilization as it would be for a simple Star topology. Furthermore, the number of switches between a sender and a receiver varies between two for neighboring and three for not neighboring switches. This results in a reduced latency compared to the Star topology, which always has three switches between a sender and a receiver. The benefits compared with a full connection topology are fewer connections and less area consumption. The Star-Wheels NoC is scalable and runtime adaptive by exploiting the dynamic and partial reconfiguration feature of Xilinx FPGAs. Therefore, if the full featured Wheel topology is not needed, each subnet could also be implemented or modified at runtime into a line, ring or star topology.

Due to this heterogeneous topology three different types of switches exist. The subswitch connects the different processing elements (PEs) to the network. Seven subswitches form the peripheral ring of one subnet, but not all of them have to be present at one point of time. The subswitches have the simplest structure as they have a maximum of four bidirectional connections. One to the PE, one to the left and one to the right neighbor subswitch in the peripheral ring and one to the superswitch, which is the central switch of each subnet. The superswitch allows the communication between subswitches, which are not direct neighbors. Neighboring subswitches can communicate directly with each other using the peripheral ring. This way the complexity and therefore also the required area of the superswitch can be reduced. It has a maximum of nine connections and is therefore the most complex one. Seven bidirectional connections are for the seven subswitches and two additional connections are used to communicate with other subnets over the central rootswitch. In the current implementation four subnets are connected via two bidirectional connections to the rootswitch. The number of subnets and connections between the subnets and the rootswitch can be increased at the cost of higher resource requirements for the rootswitch as well as for the superswitch. To support different clock domains the buffers within each switch are asynchronous.

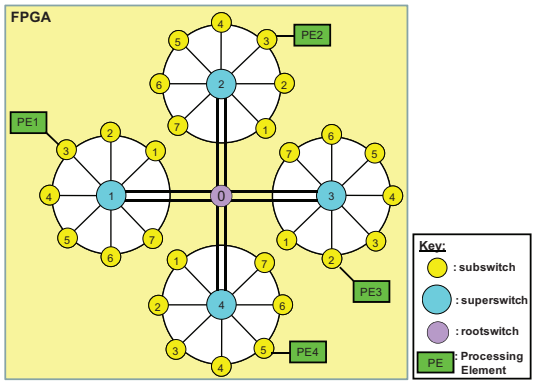

Figure 2. Heterogenous topology of the Star-Wheels Network-on-Chip

To achieve a high data throughput combined with a low latency, a heterogeneous communication protocol combining the benefits of circuit- and packet-switching was chosen. For control purposes, such as establishing and freeing a communication channel a packet-based communication protocol is used. To exchange data between to processing elements over the communication channels a circuit-switching communication protocol is used. The packet-switching and the circuit-switching are physically separated. This means for each type of communication protocol different communication ports are used, so that they will not interfere with each other. This synergy of two communication principles is beneficially for image processing applications, as circuit-switching offers the required performance for transferring large amounts of data, such as images or tiles of images, between two processing partners. Furthermore, no additional data buffers are required to reorder the incoming packages of a PE. This way the size of the switches can be kept small.

The control-packets support also the runtime adaptation of the network and the multiprocessor system. They are used to recognize, if the network topology or the number or addresses of the PEs has been changed at runtime. This runtime adaptation is feasible in e.g. Xilinx FPGAs, which provide a feature called dynamic and partial reconfiguration. With this feature a part of the configuration memory of the FPGA can be exchanged, while the rest continues to process undisturbed. Each switch has an internal timer. If within the user-specified time interval no communication has occurred between a switch and one of its neighbors or the PE, the switch sends a control-packet to them to check if they still exist. If he does not receive an answer within a second user-specified time interval, it assumes that the neighbor has been removed and it updates its internal routing table. If on the other side the switch receives an answer, it checks if the address of this communication partner is still the same. If not the routing table is updated accordingly. The control-packets are processed within each switch using a round robin scheme.

B. Design Methodology

To program such a complex hardware structure a design methodology was developed to hide the complexity of the hardware architecture from the user. Figure 3. shows this

design methodology, which consists of a combination of commercial and custom tools. In this first version still some manual steps are required. The design methodology requires an application written in C, C++ as an input. It is separated in 3 phases. Phase 1 partitions the application on a functional basis using a hierarchical clustering algorithm. Therefore, the timing of the different functions is profiled using a commercial profiling tool like e.g. the AMD CodeAnalyst. After that, the call graph is generated using a own developed tracing library. The communication analysis still had to be done manually in this version. As an alternative a novel neighborhood relationship was developed, which is explained in detail in [11]. The results from the application analysis step are used in the closeness function of the hierarchical clustering algorithm. The results of phase 1 are a suggested partitioning of the application as well as a suggested MPSoC architecture. This suggestion includes a definition of the number of processors and their required communication infrastructure. The MPSoC architecture then can be designed with the GUI of the Xilinx Platform Studio (XPS).

In Phase 2 a line by line profiling, with e.g. the AMD CodeAnalyst, has to be done for the code fragment of each processor separately. The output of the profiling is then used by the custom HW/SW partitioning tool called ProfileAnalyzer. This tool calculates the execution times within each function and each loop and also illustrates these results and their relation to each other graphically to the user. Finally, it generates a list of possible hotspots, which would be good candidates for one or several accelerators.

Phase 3 is the implementation phase. Here, the code of the application has to be manually partitioned. Inter-processor communication has to be inserted manually. The C-code, which shall be outsourced in a hardware accelerator, needs to be adapted depending on the requirements of the commercial C-to-FPGA compiler, e.g. ImpulseC [13]. Then the Xilinx tools are used for hardware synthesis and the GCC compiler is used for generating the binaries for the processors. Finally, a custom tool called GenerateRCS is used for generating the full and partial bitstreams.

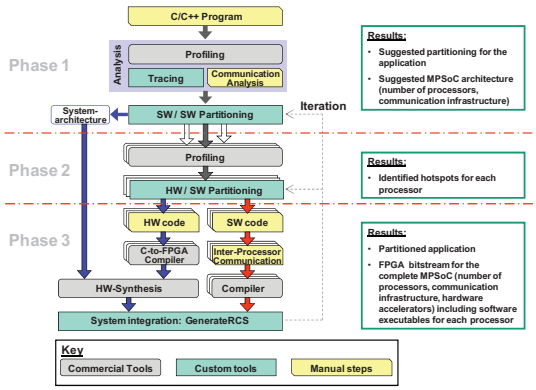

Figure 3. Design methodology of RAMPSoC without MPI support

Even though in this first version several steps are manually, no knowledge of hardware description languages is required.

IV. RAMPSoC-MPI

As many high performance computing (HPC) applications are written using the MPI standard, it was desired to support such a standard in RAMPSoC as well. This way, such applications can be transferred fast and without knowledge of the underlying hardware. Furthermore, applications written with MPI provide a possibility for the user to suggest an efficient partitioning and to insert user knowledge about the application behavior into the C-code. Moreover, these applications are very scalable, which is a desired feature for such an adaptive MPSoC like RAMPSoC. Therefore, the design methodology of RAMPSoC was extended to support also C, C++ applications with MPI, as described in Subsection IV.A. As the available open source MPI implementations are too huge in terms of memory allocation, they cannot be used by an embedded system. Therefore, an own modular MPI implementation was developed, which is described in detail in Subsection IV.B. This MPI implementation was developed to support the communication mechanisms of the Star-Wheels Network-on-Chip, but due to the usage of different implementation layers it can be easily ported to other communication infrastructures. Furthermore, the application programmer does not need to know the specific protocols of the underlying communication infrastructure, as these are hidden.

A. *Integration into the Design Methodology*

The design methodology of RAMPSoC has been extended to support also C, C++ applications using the MPI standard. Two custom tools have been developed, which resulted in a stronger automation of the design methodology as can be seen in Figure 4.

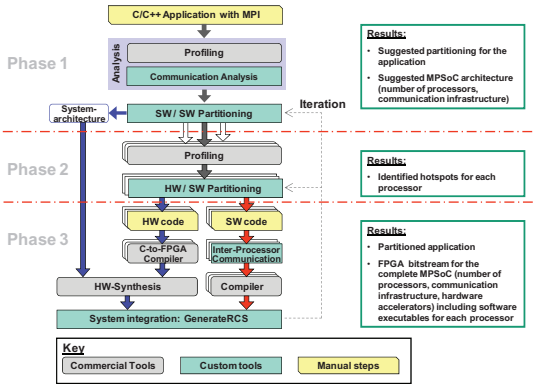

Figure 4. RAMPSoC design methodology for MPI-based C/C++ applications

1) *Communication Analysis*

The communication analysis has been automated in such a way that it uses static code analysis for generating the call graph and for extracting the communication costs between the different functions. For MPI applications it further extracts the communication costs of the MPI commands. The flow diagram of the communication analysis tool is given in Figure 5. In this first version of the tool only applications written in C are

supported, but the support for C++ is currently under development.

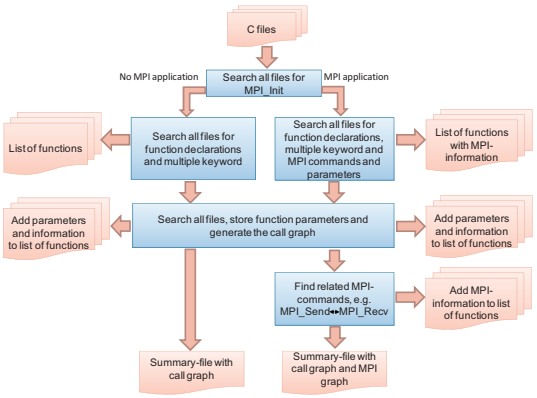

Figure 5. Flow diagram of the communication analysis tool

First, the tool searches all C input files for the MPI_Init command, which is used in all standard MPI applications for initialization purposes. If this command is not found, it is assumed, that the given application is not a MPI application. In this case the tool follows the left path in the flow diagram. If MPI_Init was found, then the tool follows the right path in the flow diagram.

In the second step the tool searches in both paths for the function declarations in all input files and generates a list with a structure for each found function. In order to find the function declaration, the user needs to declare the functions used within a file at the beginning of this file and marking the functions with a specific begin and end comment, as shown in Figure 6. In addition, the tool searches for the multiple keyword, which is specified by the following user comment "*//multiple call possible*". With this parameter the user can specify, if a specific function can be multiplied and mapped onto several processors. One example are the slave functions of the bioinformatics algorithm, as this is only one function, which can be multiplied, depending on the available number of processors, in order to speed up the overall execution. This way, the knowledge of the user about the application can be exploited by the tool. For MPI applications the tool searches further for all MPI commands and parameters used within each function and store this information also in the structure of the each function.

```
//functions declarations
void execute_master();
static void gather_results();
static void send_exit(int rank);
static void send_job(int rank, int x, int y);
static int wait_job_request();
void format_alignment(char *alignment_stack_1, char *alignment_stack_2, int alignment_stack_size);
//end of declarations
```

Figure 6. Programming rules for function declarations

In the third step, the tool searches within the different C input files for the parameters of each function and stores them also in the structure of each function. This is done for both types of applications. The function parameters are used to extract the communication costs for each function. If a function has pointers as parameters, the communication analysis tool will ask the user to specify a typical size for this pointer. With this information the tool generates the call graph for the application. For non MPI applications the communication analysis tool finishes after this step and generates a summary file with the call graph of the application.

For MPI applications a fourth step is required in order to find related MPI commands and to add this information as well as the MPI communication costs into the structure of each function. This means, to find for example for each MPI_Send the corresponding MPI_Recv command. It also considers that one MPI_Recv command can have several MPI_Send commands. For each command it analyzes the communication costs and stores them into the list of functions. Finally, a summary file is generated, which is used by the SW/SW partitioning tool, which executes the hierarchical clustering algorithm. The original closeness function of the hierarchical clustering algorithm was extended to support also MPI applications, as can be seen in equation (1).

Closeness Function:

$$C(x,y)=\begin{cases}\omega_{MPI}\dfrac{MPI\_COM(x,y)}{T(x,y)}+\omega_{Call}\dfrac{Call\_COM(x,y)}{T(x,y)}\\ \dfrac{NH(x,y)}{T(x,y)}, \text{else if } MPI\_COM, Call\_COM \text{ unknown}\end{cases} \quad (1)$$

T(x, y): Sum of the profiled runtimes of the two tasks to be clustered

MPI_COM (x, y): Communication costs between two tasks communicating via MPI

Call_COM(x, y): Communication costs between two tasks in the call graph

NH(x, y): Proximity of two tasks based on the call Graph

ω_{MPI}: Weighting factor for MPI communication

ω_{Call}: Weighting factor for call graph communication

The closeness function differentiates between communication costs resulting from the call graph (*Call_COM*) and communication costs resulting from MPI commands (*MPI_COM*). Two weights were introduced ω_{MPI} and ω_{Call}. The communication costs for the call graph receive a higher weight (ω_{Call}) to assure, that they will be more likely clustered. The communication costs for the MPI commands receive a lower weight (ω_{MPI}), because the usage of MPI between two functions is a signal given by the user, that these two functions should be placed on two different processors.

Depending on the application programmer the values for the weights can be adapted. Here for ω_{MPI} 0.2 and for ω_{Call} 0.8 have been used. If the application does not use MPI, then ω_{MPI} will be set to 0 and for ω_{Call} will be set to 1.

If MPI_COM and Call_COM are unknown, the clustering can still be done by using the own developed proximity heuristic as mentioned in Subsection III.B.

2) Inter-Processor Communication

The inter-processor communication has been automated for MPI applications through the development of an own MPI implementation for RAMPSoC called RAMPSoC-MPI. It currently supports the 18 most frequently used MPI standard commands and translates them at runtime into the corresponding commands required by the communication

protocol of the Star-Wheels NoC as described in the next subsection.

B. *OSI (Open System Interconnection) Model for RAMPSoC-MPI*

To abstract from the complexity of the underlying hardware and to make the MPI implementation easily portable for other communication infrastructures the layered approach of the OSI standard model has been exploited, resulting in the RAMPSoC model. As shown in Figure 7. six layers have been used.

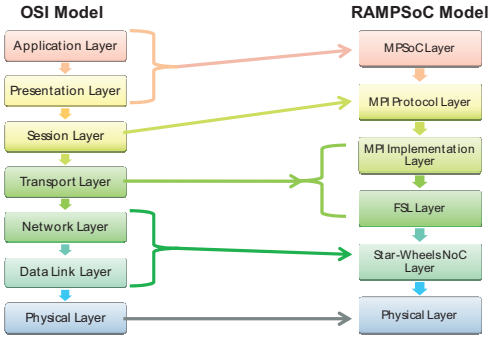

Figure 7. Relation between the OSI model and the RAMPSoC model.

The layer with highest abstraction is the MPSoC Layer, which corresponds to the combination of the Application- and the Presentation Layer of the OSI model. The MPI Protocol Layer corresponds to the Session Layer and the Physical Layer is identical in both models. The Transportation Layer of the OSI model is split into two layers in the RAMPSoC model: MPI Implementation Layer and FSL Layer, which are explained in detail in the next two subsections. Here, two layers were used for the RAMPSoC model in order to adapt the MPI implementation in the future easily to other communication infrastructures. Finally, the Network- and the Data Link Layer of the OSI model correspond to the Star-Wheels NoC Layer of the RAMPSoC model.

1) *MPI Implementation Layer*

This layer implements the 18 different MPI commands for the RAMPSoC. MPI_Broadcast calls for example MPI_Comm_size, MPI_Comm_rank and several MPI_Send commands. Within this layer, only virtual addresses are used for the different application functions. These virtual addresses are equal to the global rank, which represents the ID of each function / task within the application.

2) *FSL Layer*

In this layer the virtual addresses and therefore the global rank of the tasks / functions are transferred into the physical addresses of the executing processor within the Star-Wheels NoC. In the Star-Wheels NoC each processor has a specific address consisting of 6 Bit, 3Bit specifying the ID of the subnet and the other 3 Bit specifying the ID of the subswitch within the subnet to which the processor is connected. Furthermore, in this layer the establishment and freeing of a communication channel using the in Subsection III.A described control packets are done. In addition, incoming packets are analyzed and processed accordingly.

V. Application Integration and Results

To evaluate the functionality of RAMPSoC-MPI, a bioinformatics HPC application for DNA sequence alignment was used. This application, called Z-align [14], is from the research group of Prof. Alba de Melo. Z-align is a parallel variant of the Smith-Waterman algorithm that runs in user-restricted memory space and uses affine gap penalties. It is programmed with MPI and was evaluated by the research group of Prof. Alba de Melo on a cluster computer. The algorithm was partitioned for the RAMPSoC using the new version of the design methodology. Figure 8. shows the resulting call graph consisting of 20 functions.

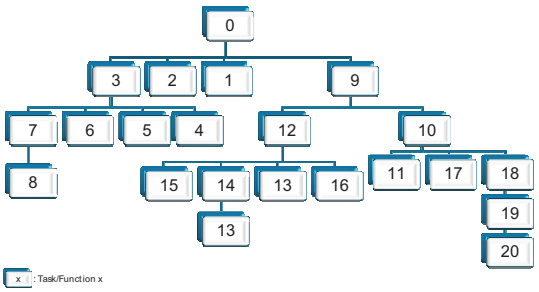

Figure 8. Call graph for the z-Align algorithm

The extracted MPI communications are shown in Figure 9. On the top right are two MPI_Bcast functions and the others are MPI_Send and MPI_Recv functions.

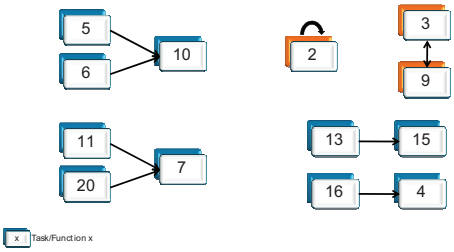

Figure 9. MPI communications between the functions / tasks of the z-Align algorithm. On the top right are two MPI_Bcast functions, all others are MPI_Send and MPI_Recv functions.

Figure 10. shows the summary, which is generated by the communication analysis tool for the z-Align algorithm. There are four columns. The first one shows the name of each function, the second one shows the ID, the third one shows the MPI communications (MPI_Com) of each functions and finally the fourth column shows the call graph communication (Call_Com) for each function. For MPI_Com and Call_Com the sender, the receiver and the length of the message are given. In Figure 10. two examples are given to show the relation between the summary file and the MPI_Com graph (see Figure 9.) and the call graph (see Figure 8.). This summary is then used as an input for the hierarchical clustering algorithm, which partitions the applications for the processors of RAMPSoC.

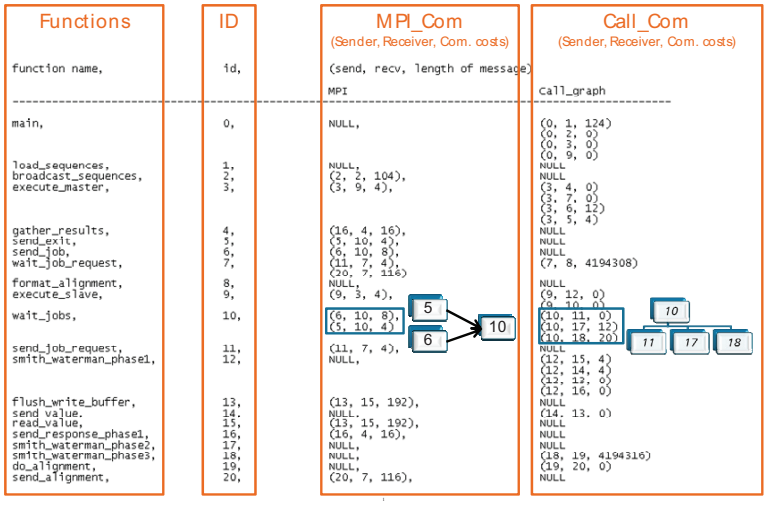

Figure 10. Summary of the communication analysis tool

The z-Align application was then partitioned for three processors: One master and two slaves, as shown in Figure 11. The processors were placed in different subnets to explore also the behaviour of the Star-Wheels NoC. Before executing the z-Align algorithm, the master processor sends to each slave its global rank and its subnet – and subwitch ID within the Star-Wheels NoC. Normally, this functionality would be done by the CAP-OS processor together with the resource management, scheduling and configuration management. As here the full functionality of CAP-OS was not required, no additional CAP-OS processor was added. The application was easily integrated into RAMPSoC without modification and it was evaluated with two sequences on a ML507 evaluation board from Xilinx with a Virtex-5FX70T FPGA.

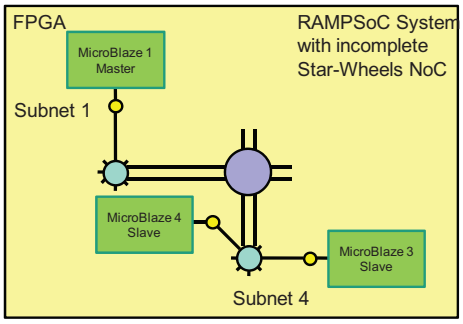

Figure 11. Implemented RAMPSoC system consisting of 3 Xilinx MicroBlaze processors connected over the Star-Wheels NoC on a ML507 evaluation board from Xilinx with a Virtex-5FX70T FPGA.

For each MPI command within RAMPSoC, the execution time was measured on the MicroBlaze processors. It was differentiated between the execution time of the MPI Implementation Layer and the FSL Layer, because the FSL Layer depends on the communication infrastructure. TABLE II. shows the supported MPI commands and their execution times for a system clock of 125 MHz. MPI_Comm_size, MPI_Comm_rank and MPI_Bcast call other MPI commands. MPI_Send and MPI_Recv are the two commands, which use the FSL Layer to communicate between two processors. MPI_Init and MPI_Finalize have the highest execution times, because they initialize / free the data structures, which are used within the RAMPSoC-MPI to store important information for each function / task, e.g. the global rank.

TABLE II. IMPLEMENTED MPI COMMANDS AND THEIR EXECUTION TIME ON A XILINX MICROBLAZE AT 125 MHZ

	MPI command	Execution Time @ 125 MHz(µs)
1	MPI_Init	47,04
2	MPI_Finalize	52,43
3	MPI_Initialized	0,43
4	MPI_Finalized	0,43
5	MPI_Comm_group	3,21
6	MPI_Group_size	0,52
7	MPI_Group_rank	1,02
8	MPI_Group_free	1,19
9	MPI_Comm_size (calls 5,6,8)	5,9
10	MPI_Comm_rank (calls 5, 7, 8)	6,23
11	MPI_Group_excl	11,37
12	MPI_Comm_create	5,65
13	MPI_Comm_free	1,17
14	MPI_Status_set_elements	0,54
15	MPI_Get_count	2,16
16	MPI_Send	0,79 (+ 2,59 for FSL Layer)
17	MPI_Recv	1,47 (+3 for FSL Layer)
18	MPI_Bcast (calls 9, 10, 16/17)	18,33 /16,09

VI. Conclusions and Future Work

This paper describes the integration of the MPI programming standard to the RAMPSoC approach. It enables to re-use existing applications on the novel runtime adaptive hardware. The approach provides a further abstraction layer over the complex hardware through the realization of the traditional OSI / ISO layer model. The MPI standard is supported by the RAMPSoC toolflow, which automatically partitions the MPI application onto the RAMPSoC resources consisting of multiple processor cores and an adaptive Network-on-Chip. This feature is achieved through an extension of the cost function in the hierarchical clustering algorithm used within the toolflow. The approach is evaluated with the z-Align algorithm from the bioinformatics domain.

For future work, it is envisioned to extend the MPI commands and user defined data types in order to increase compatibility to other application source codes. Furthermore, multitasking will be supported in order to enable intra-processor communication between tasks. Along with this, the Star-Wheels NoC will be improved to support more communication features of MPI.

The realization of standard programming models on novel multiprocessor systems is a mandatory step to receive acceptance from developers and researchers. However, it is still a hot topic in research to find a proper way to program parallel hardware efficient and from a high abstraction layer. Programming models like e.g. MPI, OpenCL, OpenMP etc. are a step forward in this direction and need to be evaluated on several platforms like e.g. the RAMSoC system.

Acknowledgment

The authors would like to thank Prof. Alba Christina M. A. de Melo, Rodolfo Batista and Felipe Scarel for providing us with the source code of the z-Align algorithm.

References

[1] J. Howard, S. Dighe, Y. Hoskote et al.: “A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS”; In Proc. ofIEEE International Solid-State Circuits Conference (ISSCC 2010), San Francisco, CA, USA, Feb. 2010.

[2] MPI: A Message-Passing Interface Standard, Version 2.2, Message Passing Interface Forum, Sept. 4, 2009. Available at: www.mpi-forum.org

[3] E. Gabriel, G.E. Fagg, G. Bosilca, T. Angskun, J.J. Dongarra, J.M.Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R.H. Castain, D.J. Daniel, R.L. Graham, T.S. Woodall: “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”; In Proc. of 11th European PVM/MPI Users’ Group Meeting, Budapest, Hungary, pp. 97-104, Sept. 2004.

[4] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, A. Lumsdaine: “Open MPI: A High Performance, Heterogenous MPI”; In Proc. of Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, Barcelona, Spain, September 2006.

[5] W. Gropp, E. Lusk, A. Skjellum: “Using MPI: Portable Parallel Programming with the Message-Passing Interface”; MIT Press, 1999.

[6] M. Saldana, P. Chow: “TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs”; In Proc. of the 16th International Conference on Field-Programmable Logic and Applications (FPL 2006), Madrid, Spain, 2006.

[7] P. Mahr, C. Lörchner, H. Ishebabi, C. Bobda: “SoC-MPI: A flexible Message Passing Library for Multiprocessor Systems-on-Chips”; In Proc. of IEEE International Conference on ReConFigurable Computing and FPGAs (ReConFig'08), Cancun, Mexico, December 2008.

[8] D. Göhringer, J. Becker: “High Performance Reconfigurable Multi-Processor-Based Computing on FPGAs”; In Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010), Atlanta, USA, April, 2010.

[9] D. Göhringer, B. Liu, M. Hübner, J. Becker: “STAR-WHEELS NETWORK-ON-CHIP FEATURING A SELF-ADAPTIVE MIXED TOPOLOGY AND A SYNERGY OF A CIRCUIT- AND A PACKET-SWITCHING COMMUNICATION PROTOCOL”; International Conference on Field Programmable Logic and Applications (FPL 2009), Prague, Czech Republic, September 2009.

[10] D. Göhringer, M. Hübner, E. Nguepi Zeutebouo, J. Becker: “CAP-OS: Operating System for Runtime Scheduling, Task Mapping and Resource Management on Reconfigurable Multiprocessor Architectures”; In Proc. of Reconfigurable Architectures Workshop (RAW 2010) der IPDPS Konferenz, Atlanta, USA, April, 2010.

[11] D. Göhringer, M. Hübner, M. Benz, J. Becker: “A Design Methodology for Application Partitioning and Architecture Development of Reconfigurable Multiprocessor Systems-on-Chip”; Annual International IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2010),Charlotte, USA, May, 2010.

[12] M. Coppola, R. Locatelli, G. Maruccia, L. Pieralisi, A. Scandurra: “Spidergon: a novel on-chip communication network”; In Proc. of Intern. Symposium on SoC, Nov. 2004.

[13] D. Pellerin, S. Thibault: “Practical FPGA Programming in C“; Prentice Hall Professional Technical Reference, 2005.

[14] R. B. Batista, A. Boukerche, A. C. M. A. de Melo: “A parallel strategy for biological sequence alignment in restricted memory space”; Journal of Parallel and Distributed Processing, vol. 68, no. 4, pages 548-561, 2008..