Realizing Reconfigurable Mesh Algorithms on Softcore Arrays

Heiner Giefers and Marco Platzner
University of Paderborn
Warburger Str. 100
33098 Paderborn, Germany
{hgiefers, platzner}@upb.de

***Abstract*—The reconfigurable mesh is a very popular model for massively parallel computation for which a large body of algorithms with exceptionally low runtime complexities exists. However, these low complexities can not be exploited due to the unrealistic assumption that communication time is either constant or logarithmic in the number of cores. Nevertheless, studying the reconfigurable mesh model and associated algorithms might lead to new approaches for the design and programming of many-cores with light-weight, circuit-switched interconnects.**

In this paper, we present the mapping of reconfigurable mesh algorithms to softcore arrays. We first discuss our architecture and the corresponding tool flow, and then turn to the most critical issues, minimizing communication delay and establishing synchronization in the single-operand, circuit-switched network. We show experimental results from an FPGA prototype and evaluate our architecture by a sparse matrix multiplication case study.

I. Introduction

A reconfigurable mesh consists of an array of processing elements and an interconnect that can be switched to form different patterns of segmented buses. All processing elements execute cycles of bus configuration, communication, and constant-time computation in a lock-step. In the 90s, the reconfigurable mesh as a model for massively parallel computing has been quite popular and a vast body of algorithms with very low runtime complexities have been developed for it [1]. The practical exploitation of the low runtime complexities, however, has been hindered mainly by the unrealistic assumption that communication time is either constant or logarithmic in the number of processing elements.

In todays computing systems, the integration of several processing cores on a single chip is already standard. Moving to many cores on chip, two key research problems are the development of energy-efficient interconnects and, a tightly coupled issue, useful parallel programming models [2], [3]. Many current proposals advocate packed-switched networks on chip with dynamic routing [4]. While such networks enable the use of well-known message passing techniques, their complex switches, routers and buffers consume substantial amounts of energy [5]. Moreover, some classes of applications do not efficiently map to message passing schemes.

As an alternative, several many-cores with light-weight, circuit-switched interconnects have been proposed. Such architectures connect the cores tightly to their switches and thus avoid the complex network interfaces and several layers of system software typically required by operating system-driven network on chip implementations. Further, the need for dynamic routing and buffering large amounts of data can also be avoided in case the sequence of switch settings is either an integral part of the parallel algorithm (e.g., for reconfigurable mesh algorithms) or determined off-line by design tools (e.g., for time division multiplexed interconnects).

The reconfigurable mesh is one possible theoretical model for such a many-core with a light-weight, circuit-switched network. Consequently, mapping reconfigurable mesh algorithms to such many-cores is of interest despite of the unrealistic assumptions about communication time that mainly served the runtime complexity analysis. Moreover, studying reconfigurable meshes and their algorithms might also lead to new approaches for the design and programming of many-cores. However, as the reconfigurable mesh has mainly be used as a theoretical vehicle, issues of practical architectures and programming tool flows have not been sufficiently considered.

In this paper, we present the practical implementation of reconfigurable mesh algorithms on an array of softcore processors. The paper is structured as follows. Section II reviews some theoretical work as well as the few existing practical implementations of reconfigurable meshes and recent circuit-switched networks on chip. In Section III, we present our architecture including the softcore array and the programming tool flow, followed by a discussion of the interconnect implementation in Section IV. Section V details our prototype system and provides a case study. Finally, Section VI concludes the paper.

II. Related Work

Different reconfigurable mesh models have been proposed in literature. The basic model, which is also used in our work, is the RMESH of Miller et al. [6]. A more general model extending the RMESH by three additional communication patterns is the Processor Array with Reconfigurable Bus System (PARBS) [7]. The Polymorphic Torus model [8] wraps the interconnect around the borders of the mesh. Of special interest are models that pose restrictions on the connectivity, such as the horizontally/vertically (HVRM) or the linear reconfigurable mesh (LRM) [9]. Other researchers deal with

optical models, e.g., the Linear Array with Reconfigurable Pipelined Bus System (LARPS) [10].

There is a wide range of application domains for which reconfigurable mesh algorithms exist, including arithmetic, sorting, selection, graph algorithms, computational geometry and image processing. An extensive overview can be found in [1].

Apart from the large body of theoretical work in the field of reconfigurable meshes, there are also some implementations: YUPPIE [11] is a SIMD computing system based on a polymorphic torus network which is able to simulate several interconnection topologies. A centralized controller generates an instruction stream for the processing elements. The processing element consists of a 1-bit ALU and five 1-bit registers. CAAPP (Content Addressable Array Parallel Processor) [12] uses a mesh of 512×512 1-bit processing elements connected through the so-called coterie network. In contrast to YUPPIE, where the global instruction selects between one of two possible communication patterns, each processing element of CAAPP autonomously establishes the switch setting.

Besides implementations, several researchers developed simulators for the reconfigurable mesh, e.g., RMSIM [13], JRM [14], JRSim [15] and the simulator described in [16]. A language called Polymorphic Parallel C together with a corresponding simulator was presented in [17]. Along the same line, [18] reported on RPC++, a SIMD language and simulator for reconfigurable processor arrays. RPC++ extends the standard C language by several control structures and data types. For instance, variables can be declared to be *poly* which implies that each PE has its own copy of a value.

Reconfigurable meshes rely on light-weight circuit-switched networks, a direction which has also been taken by several recent many-core implementations. For example, Tile64 [19] is a 64 core 32-bit processor that integrates five mesh networks, one of which uses circuit switching.

Ambric also uses a circuit switching approach for their massively parallel processing array architecture [20]. The proposed programming model, which is inspired by Communicating Sequential Processes (CSP) and Kahn Process Networks (KPN), uses processes that communicate over point-to-point links. The globally asynchronous locally synchronous architecture is synchronized by channels, which are pipelines of handshake registers.

The adaptive system-on-a-chip (aSoC) [21] employs a light-weight network to connect heterogeneous elements such as microprocessors, memories, ALUs and reconfigurable logic in a 2D mesh topology. The PicoArray [22] communicates over a network of 32-bit buses and programmable bus switches using a time division multiplex (TDM) scheme. Similar to aSoC, PicoArray schedules data transfers at compile time and requires no run-time arbitration.

III. Architecture & Design Tool Flow

In this section, we first give an overview over our softcore array architecture including the used computing and communication components. Then we discuss the main functions of the corresponding design tool flow that is used to generate both the hardware layout and the binaries for the softcore array.

A. Softcore Array

Reconfigurable meshes consist of an array of nodes, where each node splits into a processing element (PE) and an attached switch element (SE). The SEs are connected to form a two-dimensional mesh. We operate such an array as a coprocessor attached to a host processor system via a dedicated fast data link. On the coprocessor side, a communication controller feeds a number of data channels that can connect to the SEs at the border of node array. Figure 1 shows the block diagram of our system architecture.

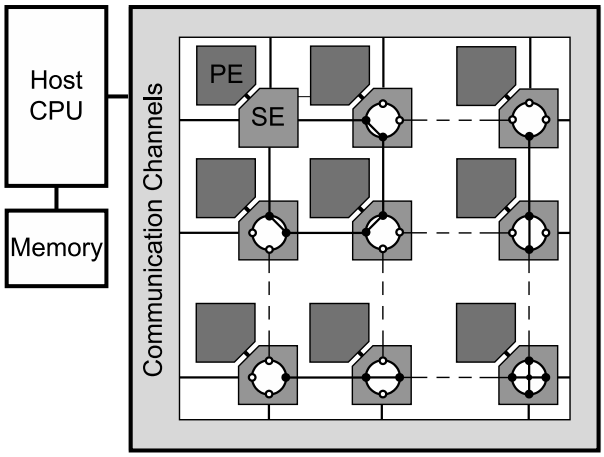

Fig. 1. System architecture for the mesh-based many-core.

An essential component of the RMESH model is the configurable switch that can be set to establish different communication patterns. A switch has four ports, one for each of the four neighbors, and can connect these ports in 12 different ways. The original RMESH communication patterns are shown in the top-most three rows of Figure 2. The pattern shown in the top-left corner denotes no connection between the ports. However, the processing element attached to the switch can always read and write to and from its four neighbor nodes. Hence we have added the four switch patterns in the bottom row of Figure 2 to establish this communication patterns. Overall, the SE can operate in 16 different configurations and requires 4-bit configuration data.

The original RMESH model uses bidirectional communication channels. Driven by FPGA technology, we implement two directed links, one in each direction, to establish a bidirectional channel. This has also been proposed in [23]. Clearly, every algorithm developed for the bidirectional model can be transformed into a corresponding algorithm for our directional model. Moreover, some algorithms even profit from the two independent links per channel as they can overlap several communication steps.

A PE can configure the switch pattern of its attached SE as well as write/read data to/from the switch. Connection

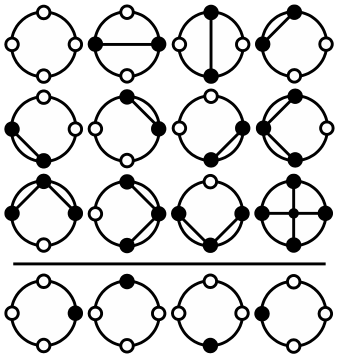

Fig. 2. Switch element connection patterns. The top 12 patterns are defined by the original RMESH model [6].

patterns that join several ports are realized over a wired OR. We have developed SEs which are parameterizable in their bit-width to be able to scale the network bandwidth with different classes of PEs. While in its simplest form, a PE can be implemented as an algorithmic state machine, more powerful meshes will employ processors. To this end, we provide appropriate wrappers that encapsulate the PE and implement registers for buffering data and the switch configuration. Figure 3 presents the node architecture. For our current implementation, we have chosen the Xilinx Picoblaze softcore, an embedded 8-bit RISC microcontroller core, as PE. The Picoblaze core occupies a rather small amount of 104 slices of logic on a Virtex-4 device plus one block RAM that can store up to 1024 instructions. The Picoblaze core provides 16 general-purpose registers, a 64-byte internal scratchpad RAM, and an automatic CALL/RETURN stack. Data can be input and output by special I/O instructions that take the data plus an additional *port_id* as operands. As the Picoblaze core consumes exactly two clock cycles per instruction, its program timing is perfectly predictable.

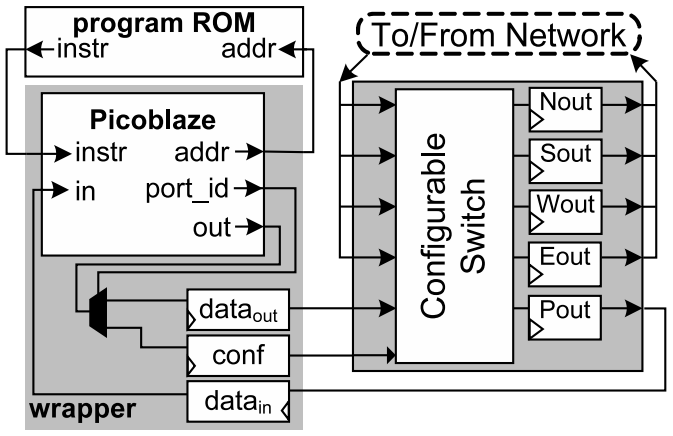

Fig. 3. A network node consisting of a processing element (PE), a switch element (SE), and corresponding wrappers.

As a host processor, we use a Xilinx Microblaze softcore. The fast data link to the coprocessor is implemented with so-called Fast Simplex Links (FSL). FSL channels provide a very low latency interface to the processor pipeline making them ideal for extending the processor's execution unit with a custom hardware accelerator.

B. Design Tool Flow

Figure 4 shows the design tool flow for our softcore array architecture. The modules in the right-hand side of the figure are basically covered by the Xilinx EDK tool flow. The parameters for a mesh instance, e.g., mesh dimensions and the specification for the communication controller, are specified in a configuration file. Alternative versions for PEs and SEs as well as appropriate drivers for data I/O are located in hardware and software repositories, respectively. Taking these data, the EDK tool flow generates the netlist for the overall architecture including the Microblaze and the softcore array. From this netlist, we extract the placement information of the block RAMs that will contain the Picoblaze programs. Alternatively, we could constrain the placement of these block RAMs before synthesis. However, our experiments have shown that we achieve better synthesis results by leaving block RAM placement to the Xilinx tool flow.

On the software side, we use the EDK flow for programming the Microblaze host and a more involved separate flow for creating the binaries for the Picoblaze softcores. Many elementary reconfigurable mesh algorithms, such as counting bits and computing wide logical functions, work in SIMD-like manner and execute the same few instructions synchronously on each processor. In contrast, we are interested in more complex applications that require the processors to compute different and larger codes. Hence, we are relying on a MIMD mode where each Picoblaze executes instructions out of local program memory. While the computation phases of the reconfigurable mesh execution cycle can be rather different, the communication phases have to be synchronized (see Section IV).

In our tool flow, an RMESH algorithm has to be specified in form of a C function, using the node identifier as parameter. The code generator calls this function for each node in the array to create assembly files for the corresponding Picoblaze cores. The resulting source files are then assembled to Picoblaze binaries and merged into an FPGA programming bitfile with the Data2Mem tool.

The code generator supports several higher-level constructs for assisting the programmer with writing reconfigurable mesh applications. These constructs include indexing functions for applying different topologies as well as conditional and loop statements. The code executed on the cores during the computation phase depends on the nodes identifiers or, more precisely, the position of a node under a certain indexing function. This information is available at compile time, allowing the compilation tool to evaluate position-depended conditionals and statically schedule individual instructions to the cores. This results in smaller code sizes for the single cores.

We have further developed a cycle-accurate simulation environment in Java. The environment comprises a Picoblaze ISA simulator for each core and an underlying mesh simulator

that realizes the transportation of data through the network. A simulation run generates logfiles which facilitate debugging and tracing the complete states of all cores and give runtime information for the tested algorithm. The graphical frontend of the simulator allows for visualizing the algorithm steps.

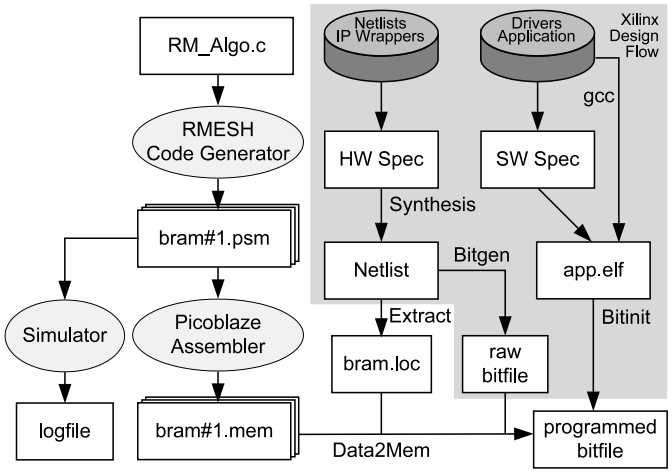

Fig. 4. Design tool flow for generating the softcore array and binaries.

IV. Mesh Interconnect

In the RMESH execution model, all PEs cycle through the three steps bus configuration, communication, and computation in lock-step. For the analysis of runtime complexities it is essential that each of these three steps takes constant time (there are some RMESH algorithms for which runtime complexities have been derived assuming that communication time is logarithmic in the number of PEs). Regarding the bus configuration and computation steps, this model can be perfectly implemented. The constant time communication step, however, is physically impossible. Several researchers deal with this limitation for the RMESH by restricting the maximum bus length to a number of k switches [24], [25]. However, it has been shown that the k-constrained reconfigurable mesh is merely as powerful as ordinary meshes, if k is a constant [26]. Reducing the RMESH model to practice, we have to give up on constant time communication and thus on the intriguing runtime complexities. The practical implementation of the communication step raises two questions: How many clock cycles are required for one communication step (delay), and how to implement the lock-step (synchronization)?

A. Communication Delay

The most straight-forward implementation approach simply sets the clock period such that communication can be done in one cycle, irrespective of the distance between writing and reading nodes. This is only reasonable if the communication patterns and distances are known at compile time. Otherwise, overly pessimistic assumptions had to be made. If writers and readers are unknown a-priori, in the worst case a broadcast signal traversing a mesh of $\sqrt{N} \times \sqrt{N}$ nodes passes $2\sqrt{N}-1$ switches if the $\{\texttt{NSWE}\}$ pattern is applied, and $N-1$ switches if the mesh is configured as a *linear bus*. The resulting signal delay would severely compromise the clock frequency in both cases.

From a practical point of view, we have to resort to implementing communication with an as small as possible delay. The approaches to design switches for fast single operand communication include:

1) Switches implemented in purely combinational logic [27]. In order to avoid extremely low clock rates, the communication infrastructure has to use multi-cycle data paths which are rather inconvenient to synthesize and analyze.
2) Switches that are able to buffer data. Adding registers to the switches simplifies circuit synthesis and allows for pipelining. Such registers are also shown in Figure 3.
3) Switches that are implemented with asynchronous (self-timed) circuit design techniques [28].
4) Photonic switches [29].

In our prototype system, we have experimented with the first two alternatives. In the first case, the switch basically consists of multiplexers, controlled by the PE. A writing node can directly write an operand to any reading node on the same bus. In the second case, the buses are realized as pipelines. A data item travels exactly one hop per clock cycle. In both cases, a communication step takes a certain number of clock cycles.

Generally, the performance for computing an algorithmic problem can be improved if i) the communication distances are short and ii) known at compile time. These features are largely determined by the selected RMESH algorithm. For many problems, several RMESH algorithms are known that differ in their theoretical runtime complexities. However, RMESH algorithms that excel theoretically are not necessarily the best choice when it comes to a practical implementation. A descriptive example discussing three algorithms for the problem of finding the maximum of N numbers shall demonstrate the trade-offs involved:

- Algorithm #1 takes an $N \times N$ mesh to find the maximum of N numbers. Initially, the N numbers are placed along the diagonal of the mesh. In the first step, the network is configured to build column buses, i.e., all switches are set to $\{\texttt{NS, W, E}\}$. The PEs on the diagonal send their numbers onto the bus; all other PEs read the numbers. Then, the network is configured to form row buses, i.e., all switches are set to $\{\texttt{N, S, WE}\}$ and the diagonal PEs again broadcast their data. At this point, each PE holds two numbers, the numbers of the diagonal PEs in the same column and row. Each PE compares the two numbers, and in case the column number is greater than the row number, an invalidation signal is written onto the row bus. Diagonal PEs read from the bus and those PEs which are not invalidated hold the maximum number.

This reconfigurable mesh algorithm takes $\mathcal{O}(1)$ communication steps. In our mesh implementation, we need N clock cycles for a row/column-spanning communication which amounts to overall $3N$ communication cycles.

- Algorithm #2 identifies the maximum among N numbers of K-bit on a mesh of size N. The network has to be configured such that all PEs are connected by a single bus. The algorithm runs in K steps. Figure 5(a) shows an example for $N = 4$ and $K = 3$. In each step $i, 0 \leq i < K$, beginning with the most significant bit, each valid PE writes the bit at position i to the bus if this bit is '1', or reads from the bus if the bit is '0'. Whenever a PE reads a '1', it invalidates itself as apparently another PE holds a greater number. After K iterations and, thus, $\mathcal{O}(K)$ communication steps, the maximum number is found. In our mesh implementation we employ an array of size $\sqrt{N} \times \sqrt{N}$ and the switch settings $\{\texttt{NSWE}\}$ for broadcasting, which results in $2\sqrt{N}K$ clock cycles for communication.
- Algorithm #3 simulates a binary tree on a mesh of dimension $\sqrt{N} \times \sqrt{N}$. Figure 5(b) shows an example for one row of a 4×4 mesh. In the first step, each PE with an odd index receives the number from its right neighbor and stores the maximum of this number and its own number. Then, all PEs with even indices configure their switches as pass-through, i.e., pattern $\{N, S, WE\}$, and the procedure is repeated with all PEs of index i and $i+2$. After $log_2(N)$ steps the left-most PEs in each row holds the row maximum. The procedure is again applied to the first column. Eventually, the top-left PE holds the maximum number. This algorithm requires $2log_2(N)$ communication steps in the reconfigurable mesh model, which maps to $2\sqrt{N}$ clock cycles in our implementation.

Considering runtime complexity, algorithm #1 is certainly optimal. This optimality, however, is achieved at the expense of N^2 PEs. For algorithms #2 and #3, N PEs are sufficient. In turn, their runtime complexities depend either on the bit-width or weakly on the number of PEs. Importantly, in a practical implementation where there is no constant time communication the trade-offs between the different algorithms change. The area-consuming algorithm #1 which theoretically excels in runtime, actually requires the most cycles for communication. Algorithm #2 whose runtime complexity depends only on the bit-width which presumably is rather small, takes K times more cycles for communication than algorithm #3.

Picking an RMESH algorithm with the lowest runtime complexity to solve a given problem on a practical mesh does not automatically lead to the fastest implementation. As a more general observation, we note that those RMESH algorithms are good candidates for practical implementation that tend to rely on multiple, but rather short bus segments.

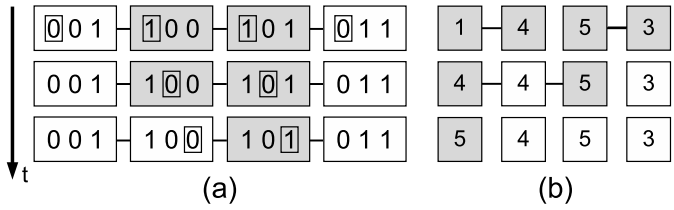

Fig. 5. Two maximum algorithms for N values on N nodes.

B. Synchronization

In an RMESH, all nodes operate in lock-step. While every PE has its own instruction memory and program counter and can thus run independently, the communication step must be synchronized. Currently, we have implemented two different ways of achieving this synchronization.

The first approach schedules PE instructions in such a way that the communication steps on all PEs are executed in the same clock cycle. This requires that we know a-priori the execution times of all computation steps. Since the computation steps of RMESH algorithms must have constant runtime, this is not overly restrictive. The instructions to be executed by a certain PE primarily depend on the PE's identifier and the mesh dimension. If the algorithm statically differentiates between PEs, our code generator statically schedules `nop` instructions to ensure that all branches (all PEs) take the same number of cycles. For dynamic branches, the code generator schedules assembly code that, again, leads to equal execution times irrespective of the branch taken.

A second method for synchronizing the cores is to interrupt them if valid data is detected on the bus. To that end, the data lines are extended with a valid bit. If data arrives, the Picoblaze core receives an interrupt and jumps to the highest address (typically, `3FF`). This interrupt-driven technique couples the cores in a more loosely fashion and does not require the rather tedious synchronization at the level of clock cycles.

```
ENABLE INTERRUPT
...
ADD BARRIER, <#SLEEP+1>
CALL _WAIT
INPUT ACC, pIn
...
_WAIT:
COMPARE BARRIER, 00
RETURN Z
JUMP _WAIT

_ISR:
SUB BARRIER, 01
RETURNI ENABLE

;@3FF
JUMP _ISR
```

Listing 1. Reader

```
...
OUTPUT ACC, pOut
LOAD  RLOOP, 02
CALL _LOOPNOP
...

_LOOPNOP:
SUB RLOOP, 01
COMPARE RLOOP, 00
JUMP NZ, _LOOPNOP
RETURN
```

Listing 2. Writer.

Listings 1 and 2 show assembly code for the communication routines of a reader and a writer node that implements the interrupt-driven barrier mechanism. First, the reader initializes its local `BARRIER` register. If a reader does not participate in the next k communication phases, it simply adds $k+1$ to `BARRIER` and calls the `_WAIT` subroutine. Every time valid

TABLE I
SOFTCORE ARRAY COMPONENTS: RESOURCE UTILIZATION AND CLOCK RATE (IN MHZ) FOR A VIRTEX-4 LX200/SPEEDGRADE -11.

Building block	**slices**	**BRAM**	f_{max}
Picoblaze (PE_1)	114 (0.13%)	1	184
Picoblaze+mult. (PE_2)	131 (0.15%)	1	184
RMESH switch (SE_1)	144 (0.15%)	0	192
HVRM switch (SE_2)	67 (0.08%)	0	237
Node$_1$ ($PE_1 + SE_1$)	265 (0.30%)	1	156
Node$_2$ ($PE_1 + SE_2$)	178 (0.20%)	1	156
Node$_3$ ($PE_2 + SE_1$)	195 (0.22%)	1	156
Node$_4$ ($PE_2 + SE_2$)	282 (0.32%)	1	156
Prototype: μBlaze, FSLs and 256 $\times$ Node$_4$	65091 (73%)	264 (78%)	100

data passes the node, the `BARRIER` register is decremented. When the register reaches 0, the `_WAIT` block is exited and the next valid data can be read.

The writer node outputs its data on a dedicated port (named `pOut` in Listing 2). If required for the overall timing, writers execute a number of `nop` instructions according to a predetermined waiting time. For bounding code size, waiting times are implemented in loops, e.g., `_LOOPNOP` in Listing 2. It is important to note that there is no acknowledgment sent over the bus. Consequently, the code generation tools have to make sure that readers do not lag more than one write phase behind. Further, situations must be avoided where there is no writer for a bus segment. There are several way to further improve the synchronization mechanism. For example, using consecutively numbered communication identifiers instead of a valid bit would allow for more flexibility.

V. PROTOTYPE & CASE STUDY

In this section, we present our FPGA prototype and a matrix multiplication case study in order to demonstrate the practicability of a many-core architecture that follows the reconfigurable mesh model.

A. FPGA Prototype

We have synthesized the different components for our softcore array to an FPGA Xilinx Virtex-4 LX200, speedgrade -11. The synthesis results are shown in Table I. As processing element we employ the Xilinx Picoblaze core in its native version (PE1), as well as extended with a dedicated multiplier (PE2), both versions contained in a wrapper. Further, we have implemented two switch elements, one for the RMESH model (SE1) and one for the more restricted horizontally/vertically switched mesh (SE2). The resource utilization and speed figures for the different node types are also presented in Table I.

As a host, we use the Xilinx Microblaze 32-bit RISC core connected to an $N \times N$ array of PEs via two FSLs. An overall softcore array for running RMESH algorithms is presented in Figure 6. Every first tile of a row contains a special wrapper that connects to the communication controller. This controller fetches 32-bit words out of the incoming FSL, splits the words into four 8-bit words and shifts the data to the indented position of the array. As this conversion takes one clock cycle, a complete column of input data can be provided every $\lceil \frac{N}{4} \rceil$ cycles. For writing data, the communication controller assembles consecutive 8-bit words to 32-bit words and pushes them into the outgoing FSL. The latency for transferring an operand through an FSL link is four clock cycles. Figure 6 displays the 16×16 node prototype system we have used for implementing the matrix multiplication case study. The resource utilization and clock speed for this prototype is also given in Table I.

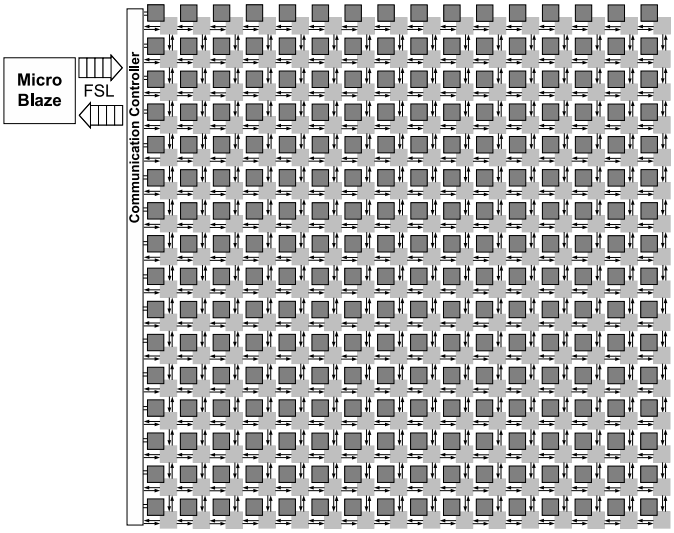

Fig. 6. Prototype comprising a Microblaze system attached to a 16×16 RMESH.

B. Case Study

As an example, we have mapped the reconfigurable mesh algorithm for sparse matrix multiplication presented in [30] to our 16×16 array. The algorithm multiplies two matrices A and B of dimension $N \times N$ on an array of size $N \times N$ and is outlined in Algorithm 1. The elements $a_{ij} \in A$ and $b_{ij} \in B$ have to be preloaded to the corresponding nodes in the array.

The outer `repeat`-loop iterates over the maximum number of nonzero elements in a column, which is less or equal to k, the sparseness value. In the inner-most `forall`-loop, all nonzero products are considered. As there are also at most k of these products per column, the runtime complexity of the algorithm is $O(k^2)$ steps.

The practical realization of this reconfigurable mesh algorithm demonstrates the disparity between the algorithm formulation in pseudo code and the actual physical implementation. For example, the algorithm requires to "route the top-most nonzero p-element of column i to all nodes in row j", where j is the index given by one of the factors of p. Theoretically, this pseudo code statement takes $\mathcal{O}(1)$ steps. In the implementation, we require six column/row broadcasts to realize this step.

We have measured the algorithm runtimes for different sparseness values. The results for $k = 1, \ldots, 10$ are shown in Figure 7. This figure presents the number of clock cycles

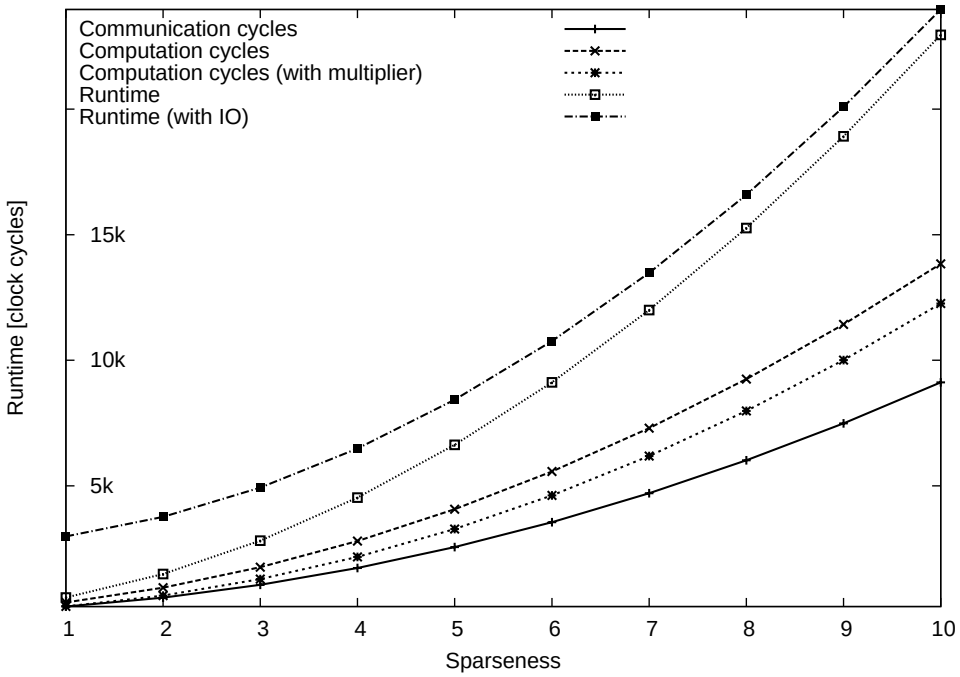

Fig. 7. Runtimes of the matrix multiplication algorithm for different sparseness values.

Data: Matrices A, B with at most k nonzero elements per row
Result: $C = A \times B$
repeat
  forall *columns i* **pardo**
    Broadcast the top-most nonzero A-element of column i together with its row index to all PEs in row i.
  parend
  forall *PEs* **pardo**
    Multiply the received A-element with the local B-element.
  parend
  forall *columns i* **pardo**
    forall *non-zero products p* **do**
      Route the top-most nonzero p-element to the row with the index given by the A-element.
    end
    Discard the top-most nonzero A-element.
  parend
until *All nonzero A-elements have been discarded*

Algorithm 1: Column-sparse matrix multiplication [30].

for communication, computation with and without dedicated multipliers, raw algorithm runtime (with multipliers), and the overall runtime including data communication to the host.

Considering a sparseness of $k = 2$, i.e., a matrix column contains at most two nonzero elements, the overall algorithm runtime is 4590 cycles when using a shift&add routine for multiplication and 3886 cycles when a dedicated multiplier is used. Compared to a straight-forward matrix multiplication routine on the Microblaze which takes 14788 clock cycles, the array is 3.8 times faster. As Figure 7 clearly demonstrates, data I/O leads to a substantial overhead, especially for small sparseness levels where the RMESH algorithm is more efficient. For $k = 2$, as many as 2590 cycles and thus 67% of the overall runtime is spent for transferring data in and out of the array.

Our reconfigurable mesh implementation shares the problem of limited I/O bandwidth with all massively parallel architectures. An approach to increase I/O bandwidth is to feed data from several FSL channels in parallel to our array. The efficiency of this approach depends on the overall system architecture and, eventually, the bandwidth to system memory. Further, the matrix multiplication algorithm might be used as a kernel embedded into a larger application. If the matrices are already in the array as a result of a previous computation, the mesh is 11.4 times faster than the Microblaze for $k = 2$. If we consider a sparseness of $k = 1$, our architecture computes the result in 200 cycles. Compared to the single Microblaze, this yields a speedup of almost 74.

VI. Conclusion

We have presented the design and implementation of a softcore array and a corresponding tool flow, that allow the realization of reconfigurable mesh algorithms. A sparse matrix multiplication case study on an FPGA prototype utilizing 256 softcores has demonstrated the feasibility of our approach. The main result of this work is the insight into the challenges and alternatives of realizing reconfigurable mesh algorithms on many-cores with circuit-switched networks.

The quantitative results we have achieved point to two

important issues that need to be addressed in future work. First, the optimization of the communication phase is of utmost importance to make the reconfigurable mesh approach competetive. One one hand, we need fast implementations of the communication infrastructure in silicon – our implementation in field-programmable logic can only serve as an emulation. On the other hand, we also have to carefully select the reconfigurable mesh algorithm used for a given problem. Here, we have discussed attributes of RMESH algorithms that make them amenable to an efficient many-core implementation. Second, as any massively parallel architecture, the reconfigurable mesh can severely suffer from I/O bandwidth limitations. We need to implement larger applications and analyze the system-level performance for a fair comparison to alternative architectures.

A considerable restriction of almost any reconfigurable mesh algorithm is posed by the dependency between the problem size and the mesh dimension, which severely hampers scalability. For example, the sparse matrix multiplication algorithm used in our case study requires a mesh of the same dimensions as the matrices have. Several techniques have been proposed for reconfigurable meshes in order to achieve scalability with problem size, one being *self-simulation*. For example, Ben-Asher et al. [9] discuss self-simulation techniques and their properties, and present an optimal simulation for the HVR-Mesh model, a strong simulation for the LR-Mesh model and a weak simulation for the basic RMESH model.

A key issue for our future work is to add such scaling strategies to our code generation tools. In our case study, we use a slightly modified version of the original sparse matrix multiplication algorithm of [30]. For this modified algorithm, the HVRM switch patterns are sufficient. Hence we can apply an optimal self simulation that comes with a slowdown of $O(\frac{N^2}{P^2})$, when multiplying matrices of size $N \times N$ on a mesh of size $P \times P$. This particular self simulation technique uses the so-called contraction mapping to assign virtual to physical nodes. Thereby, a local contiguous region of the larger (simulated) mesh is evaluated by a single node on the real (simulating) mesh. Benefits of this technique are shorter broadcast distances and the opportunity to increase the load for single nodes, which improves scalability and efficiency of the proposed architecture.

REFERENCES

[1] R. Vaidyanathan and J. L. Trahan, *Dynamic Reconfiguration*, R. G. Melhem, Ed. Springer, 2004.

[2] L. Benini and G. DeMicheli, "Networks on Chips: A New SoC Paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[3] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of Network-on-chip," *ACM Computing Surveys*, vol. 38, no. 1, 2006.

[4] W. J. Dally and B. Towles, "Route Packets, Not Wires: On-Chip Inteconnection Networks," in *Proc. of the 38th Conference on Design Automation, DAC '01*, 2001.

[5] A. Banerjee, R. Mullins, and S. Moore, "A Power and Energy Exploration of Network-on-Chip Architectures," in *Proc. 1st Int. Symposium on Networks-on-Chip NOCS'07*, 2007, pp. 163–172.

[6] R. Miller, V. Prasanna-Kumar, D. Reisis, and Q. Stout, "Parallel Computations on Reconfigurable Meshes," *IEEE Transactions on Computers*, vol. 42, no. 6, pp. 678–692, 1993.

[7] B.-F. Wang and G.-H. Chen, "Constant Time Algorithms for the Transitive Closure and Some Related Graph Problems on Processor Arrays with Reconfigurable Bus Systems," *Transactions on Parallel and Distributed Systems*, vol. 1, no. 4, pp. 500–507, 1990.

[8] H. Li and M. Maresca, "Polymorphic-Torus Network," *IEEE Transactions on Computers*, vol. 38, no. 9, pp. 1345–1351, 1989.

[9] Y. Ben-Asher, D. Gordon, and A. Schuster, "Efficient Self-Simulation Algorithms for Reconfigurable Arrays," *Journal of Parallel and Distributed Computing*, vol. 30, no. 1, pp. 1–22, 1995.

[10] Y. Pan and K. Li, "Linear Array with a Reconfigurable Pipelined Bus System – Concepts and Applications," *Inf. Sci.*, vol. 106, no. 3-4, pp. 237–258, 1998.

[11] M. Maresca and H. Li, *Reconfigurable Massively Parallel Computers*. Prentice Hall, 1991, ch. Polymorphic VLSI arrays with distributed control, pp. 33–63.

[12] C. Weems, S. Levitan, A. Hanson, E. Riseman, J. Nash, and D. Shu, "The Image Understanding Architecture," *Int. Journal of Computer Vision*, vol. 2, no. 3, pp. 252–282, 1989.

[13] M. M. Murshed and R. P. Brent, "RMSIM: A Serial Simulator for Reconfigurable Mesh Parallel Computers," The Australian National University, Tech. Rep. TR-CS-97-06, 1997.

[14] K. Miyashita and R. Hashimoto, "A Java Applet to Visualize Algorithms on Reconfigurable Mesh," in *Proc. of the 15th Workshops on Parallel and Distributed Processing, IPDPS '00*, 2000, pp. 137–142.

[15] K. Sun, J. Zheng, Y. Li, and X. Pan, "Design of a Simulator for Mesh-Based Reconfigurable Architectures," in *Int. Conf. on Network and Parallel Computing*, 2007.

[16] C. Steckel, M. Middendorf, H. A. ElGindy, and H. Schmeck, "A Simulator for the Reconfigurable Mesh Architecture," in *IPPS/SPDP Workshops*, 1998, pp. 99–104.

[17] M. Maresca and P. Baglietto, "A Programming Model for Reconfigurable Mesh based Parallel Computers," in *Programming Models for Massively Parallel Computers*, 1993, pp. 124–133.

[18] P. Baglietto, M. Maresca, and M. Migliardi, "A Simulator For Reconfigurable Massively Parallel Architectures," in *2nd Euromicro Workshop on Parallel and Distributed Processing*, January 26-28 1994, pp. 185–189.

[19] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal, "On-Chip Interconnection Architecture of the Tile Processor," *IEEE Micro*, vol. 27, no. 5, pp. 15–31, 2007.

[20] M. Butts, "Synchronization through Communication in a Massively Parallel Processor Array," *IEEE Micro*, vol. 27, no. 5, pp. 32–40, 2007.

[21] J. Liang, A. Laffely, S. Srinivasan, and R. Tessier, "An architecture and compiler for scalable on-chip communication," *IEEE Transactions on VLSI Systems*, vol. 12, no. 7, pp. 711–726, 2004.

[22] G. Panesar, D. Towner, A. Duller, A. Gray, and W. Robbins, "Deterministic Parallel Processing," *Int. Journal of Parallel Programming*, vol. 34, no. 4, pp. 323–341, 2006.

[23] Y. Ben-Asher and A. Schuster, "Time-size tradeoffs for reconfigurable meshes," *Parallel Processing Letters*, vol. 6, no. 2, pp. 231–245, 1996.

[24] B. Beresford-Smith, O. Diessel, and H. ElGindy, "Optimal algorithms for constrained reconfigurable meshes," *Journal of Parallel and Distributed Computing*, vol. 39, no. 1, pp. 74–78, 1996.

[25] M. Kunde and K. Gurtzig, "Efficient sorting and routing on reconfigurable meshes using restricted bus length," in *Proc. th International Parallel Processing Symposium*, 1997, pp. 713–720.

[26] M. Murshed and R. P. Brent, "How Promising is the k-Constrained Reconfigurable Mesh?" in *Proc. of the 15th Int. Conf. on Computers and Their Applications*, 2000, pp. 288–291.

[27] H. Giefers and M. Platzner, "A Many-Core Implementation Based on the Reconfigurable Mesh Model," in *Proc. Int. Conf. on Field Programmable Logic and Applications, FPL '07*, 2007, pp. 41–46.

[28] M. Amde, T. Felicijan, A. Efthymiou, D. Edwards, and L. Lavagno, "Asynchronous on-chip networks," *Computers and Digital Techniques, IEE Proceedings -*, vol. 152, no. 2, pp. 273–283, 2005.

[29] A. Shacham, K. Bergman, and L. P. Carloni, "The Case for Low-Power Photonic Networks on Chip," in *Proc. of the 44th Design Automation Conference, DAC '07*, 2007, pp. 132–135.

[30] M. Middendorf, H. Schmeck, H. Schröder, and G. Turner, "Multiplication of Matrices With Different Sparseness Properties on Dynamically Reconfigurable Meshes," *VLSI Design*, vol. 9, no. 1, pp. 69–81, 1999.