in IC-SAMOS: Samos, Greece, July 17-20, 2006

FLUX Networks: Interconnects on Demand

Stamatis Vassiliadis and Ioannis Sourdis
Computer Engineering,
TU Delft,
The Netherlands,
{stamatis, sourdis}@ce.et.tudelft.nl
http://ce.et.tudelft.nl/

***Abstract*— In this paper, we introduce the FLUX interconnection networks, a scheme where the interconnections of a parallel system are established on demand before or during program execution. We present a programming paradigm which can be utilized to make the proposed solution feasible. We perform several experiments to show the viability of our approach. We experiment on three case studies, evaluate different algorithms, developed for meshes or binary trees, and map them on "grid"-like physical interconnection networks. Our results clearly show that, based on the underlying network, different mappings are suitable for different algorithms. Even for a single algorithm different mappings are more appropriate, when the processing data size or the number of utilized nodes changes. The implication of the above is that changing interconnection topologies/mappings (dynamically) on demand depending on the program needs can be beneficial.**

I. Introduction

In computer engineering, improvements have been achieved with the technological advances in terms of area (which presumably increases exponentially), delay and chip I/O count (which we postulate increases at best linearly). It has been postulated that, under the conjunctures stated above, microarchitectures provide a substantial increase in performance in uniprocessor systems. Based on experimental evidence, however, it has been indicated that it is doubtful such a claim can be substantiated in the recent past [1]. Given that uniprocessor microarchitectures may experience some difficulties to exploit technological advances, it can be envisioned that multiprocessors could be the answer to the performance quest. In the very near future, it is almost certain that the VLSI technology will allow single chip multicore general purpose processors to become feasible (possibly exceeding the order of 10^x, where $x \geq 2$). Multiprocessor multichip parallel systems are not new (e.g. see ILIAC IV [2]), and it will appear that using past multiprocessor experiences and applying them in single chip VLSI implementations will provide a solution to general purpose uniprocessor performance scalability. While multiprocessors can be implemented on a chip the VLSI design of single chip massive multiprocessors is only one of the challenges and by no means the only one. Simply stated, being able to fit numerous processors in a single chip, does not necessarily imply that the performance increases substantially. It is well known, that in the past only a small fraction of peak performance has been achieved in parallel systems. There are numerous problems that prohibit top performance achievements. For example, assuming shared memory paradigms, scalability is not guaranteed a priori. Clearly, coherence does not scale (not easily) and most definitely creates costs that substantially diminish potential multiprocessor advantages. Additionally, software performance is not "portable". That is, software development for a system at time t may not scale to a system developed at time $t + 1$. One of the fundamental reasons, but by no means not the only one, is that software does not "mutate" to take into account new network topologies, while seldom parallel systems use a single network topology from one design point to the next.

In this paper, we address a single challenge regarding multiprocessor parallel systems. We consider the effects the interconnects have on the portability and scalability of software performance. It is a well known fact that developed algorithms have in mind an interconnection network. Traditionally speaking, interconnection networks are rigid and often (actually usually) the interconnection network changes from one design point to the next. A consequence of the above is that algorithms and software, when ported to a new family of multiprocessor parallel systems, will not scale in terms of performance (at least) and new software development has to be under way if performance is critical.

We introduce a new approach, diametrically opposite to the existing network proposals, for adaptable networks stated by the following: *Interconnection networks are provided (dynamically) on demand to suit the needs of an application/algorithm/program*. We describe some potential implementation and propose a programming paradigm that may allow the interconnects to be fused with traditional models. Finally, we provide experimental evidence suggesting that our proposal is promising.

The paper is organized as follows: In Section II, we present different solutions for FLUX interconnection networks and provide a programming paradigm to change dynamically on demand processing and interconnecting of processors (general purpose or not) allowing them to adapt to the interconnect demands of software. In Section III, we provide initial experimental data supporting our approach. Finally, in Section IV we present our conclusions.

This work was supported by the European Commission in the context of the Scalable computer ARChitectures (SARC) integrated project #27648 (FP6).

1-4244-0155-0/06/$20.00 ©2006 IEEE

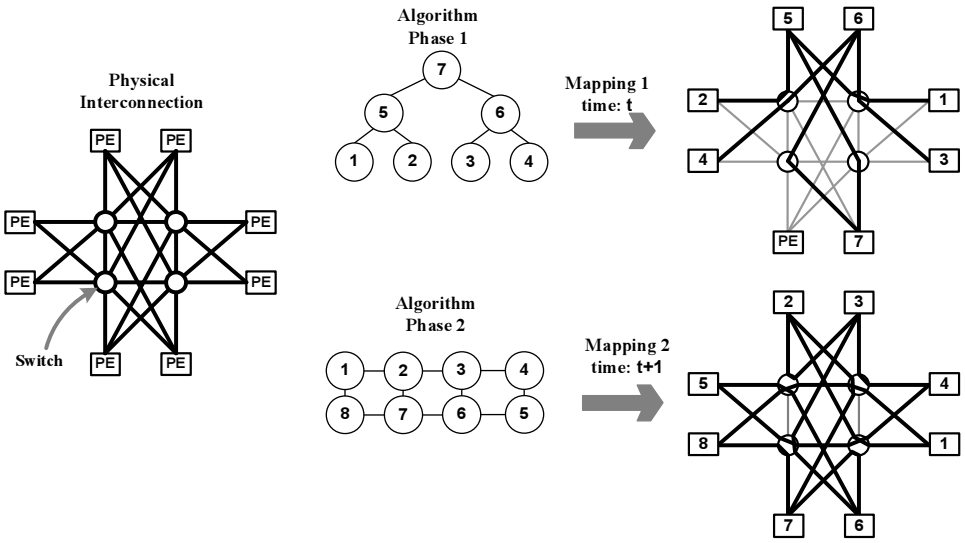

Fig. 1. FLUX Network on Demand.

II. FLUX Interconnects on Demand

Currently, single-chip multiprocessor systems are designed based on a specific hardwired interconnect topology. Algorithms should be created to suit the multiprocessor system topology in order to maximize performance. In the present paper, we propose the opposite: *the physical interconnection network is installed/configured/adapted (dynamically) to fit communication needs*.

Before we introduce the proposed approach in detail, we first describe the concept of logical and physical networks. We denote logical network as the network which the application designer has in mind. For example, the logical network structure of an application developed for binary trees is a binary tree with specific guidelines about the workload distribution and the nodes communication. The physical network is the network available by the designed chip. As described earlier the logical and physical networks do not always match, therefore, the logical structure somehow has to be mapped into the physical network. In this case, no matter what the logical structure is, the physical network constraints the mapping and the logical network connections have to follow the physical paths, usually through intermediate (switching) nodes. Therefore, the link delay of the physical network is the lowest delay that a logical link mapping can achieve. Finally, when mapping is performed, several parameters have to be taken into account such as congestion, dilation and expansion [3].

To exemplify our approach, consider the multiprocessor system of Figure 1 which consists of several Processing Engines (PEs) physically connected on a physical interconnection network. Note that the underlying physical network structure may be highly irregular and chosen by the designer to best "fit in" the technology he/she is considering rather than a predetermined regular structure as proposed by *all* existing network topologies. In the case of an algorithm implemented for binary-trees (BT), this scheme, given a mapping algorithm, can connect the PEs in a BT topology. Similarly, for an algorithm that is suitable for a mesh interconnect, the network topology can be a mesh. Of course, this flexibility is limited by the resources available for the interconnection. This means that the number of the PEs that can be connected in a specific topology depends on the routing resources available (wires and switch boxes). In the proposed FLUX Network on demand, PE interconnection could change during the execution of a single program. If different phases of a program "prefer" different topologies, then the interconnection network could change at run-time. Consequently, at time t the interconnection topology can be a BT and at time $t + 1$ can change to a 2-D mesh. More precisely, in each phase we reassign the nodes and the connections required to match the communication needs of the BT at time t and the mesh at time $t + 1$. Obviously, for a given physical network, logical topologies can be mapped more or less efficiently depending on the logical network and the mapping algorithm to the physical structure.

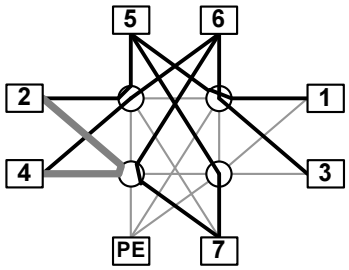

Fig. 2. Direct connections additional to the network topology.

Any network mapping algorithm might leave some of the resources of the underlying network "unused". That means that a network structure per se may not be needed and processors could be connected on demand at point to point networks if there are available connections (unused routing resources). When a BT is mapped into the topology of Figure 1, unused

links can be used to connect two PEs additionally to the utilized interconnection network (Figure 2). In this example, a direct/hot connection between PEs #2 and #4 can be established besides the existing binary tree (BT) interconnect (in dark lines). This connection should be *set* when needed and *released* when the data exchange is finished. That is, if on a specific time "processor 2" needs to communicate with "processor 4" without going via the existing network (BT), which normally would have been following for example nodes: *2-5-7-6-4*, because of a critical event, then a direct connection is established (and afterwards released) on demand.

Programming Paradigm: In order for a network to exhibit the properties described above, explicit network calls should be added to the programming paradigm to support adapting the physical interconnect on demand. In the following, we discuss the way of adapting an interconnection network using ISA extensions similar to the Molen paradigm [4]. Hardware implementations of arbitrary interconnection networks can be instantiated under software or hardware control before program execution or at runtime. They are detected "on-the-fly" or pre-determined "off-line" at hardware/software co-design stage using application partitioning, profiling, monitoring etc.. Generally speaking, the FLUX network mechanism would require additional ISA support to enforce the intended interconnection network. A master-slave parallel processing model could be envisioned with the master processor being responsible for the following:

- Node mapping: distribute the workload to the PEs of the system (possibly generate it as well) and specify an address per node.
- Connection mapping: Specify the communication path between each pair of nodes.
- Run the master/manager process, keeping sequential consistency of the program.
- Control and synchronize the PEs (activate PEs, receive a message when a PE job is finished)
- May perform part of the work itself.

When it is needed to configure the network, then a SET $< parameters >$ instruction is necessary (similar to Molen paradigm [4]). As depicted in Figure 3, the parameters specify the way the logical network (according to the communication needs of the application) maps into the physical network. The parameters are at least the following:

- Node addressing/mapping.
- Workload assignment to nodes (including number of utilized nodes).
- Establish routing paths (mapping of the logical paths to the physical ones)

It should be noted that in difference of existing programming paradigms, our proposal allows usual program structures to co-exist with the direct exposure and controlling of the physical network. In the case of direct point-to-point connections, the communication paths are either scheduled statically at compile time or allocated dynamically. When a request is allocated dynamically, it should be checked first whether the required resources are available (wires and switches), and then that the destination PE(s) is/are available to receive a new connection. The procedure could be based on circuit switching and repeated in a round trip delay request fashion, in case a direct connection is not possible, due to limitations. When all necessary requirements are met the direct connection(s) can be configured using a partial SET $< parameters >$ instruction. Lastly, when the necessary data is exchanged the connection should be released, meaning that the utilized resources should be again available for other possible use.

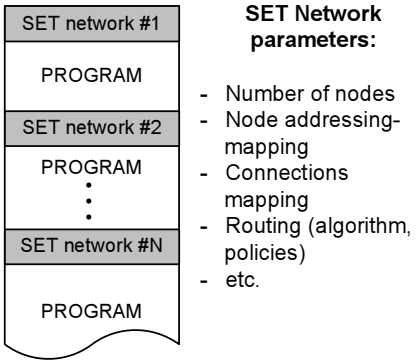

Fig. 3. Execution of the SET instruction before or during different phases of an application.

FLUX Networks on Reconfigurable Hardware: Reconfigurable technologies have an underlying network that can be (dynamically) "modified", thus they are excellent potential for FLUX implementation platforms. In this paragraph, we consider using reconfigurable hardware as the underlying network of the FLUX interconnects. Current FPGA physical interconnects can approximate the logical network of an application (i.e. one-to-one mapping), since they use different types of wires to traverse short, medium or long distances. This way, distant logic blocks can be connected avoiding most of the in between switch boxes and the delay they introduce. The entire interconnection network, or part of it can be reconfigured on demand using the programming paradigm described above and the MOLEN ISA extensions [4]. Reconfigurable FLUX networks can be implemented using numerous schemes including (but not limited by) the following:

- **FLUX networks with dynamic PE placement:** The interconnection networks and the PEs are reconfigured (dynamically). In this case, the PEs are soft-cores in order to alleviate the routing of the network (place & route).
- **FLUX networks with static PE placement:** Only the network is (dynamically) reconfigurable, while the PEs are statically placed (hard-cores), consequently, the reconfiguration overhead is minimized.
- **Direct "point-to-point" & Chaotic Interconnects:** The FPGA routing structures provide an underlying "unused" reconfigurable network. Consequently, a network structure per se may not be needed and processors could be connected on demand at direct point to point connections if there are available connections (unused

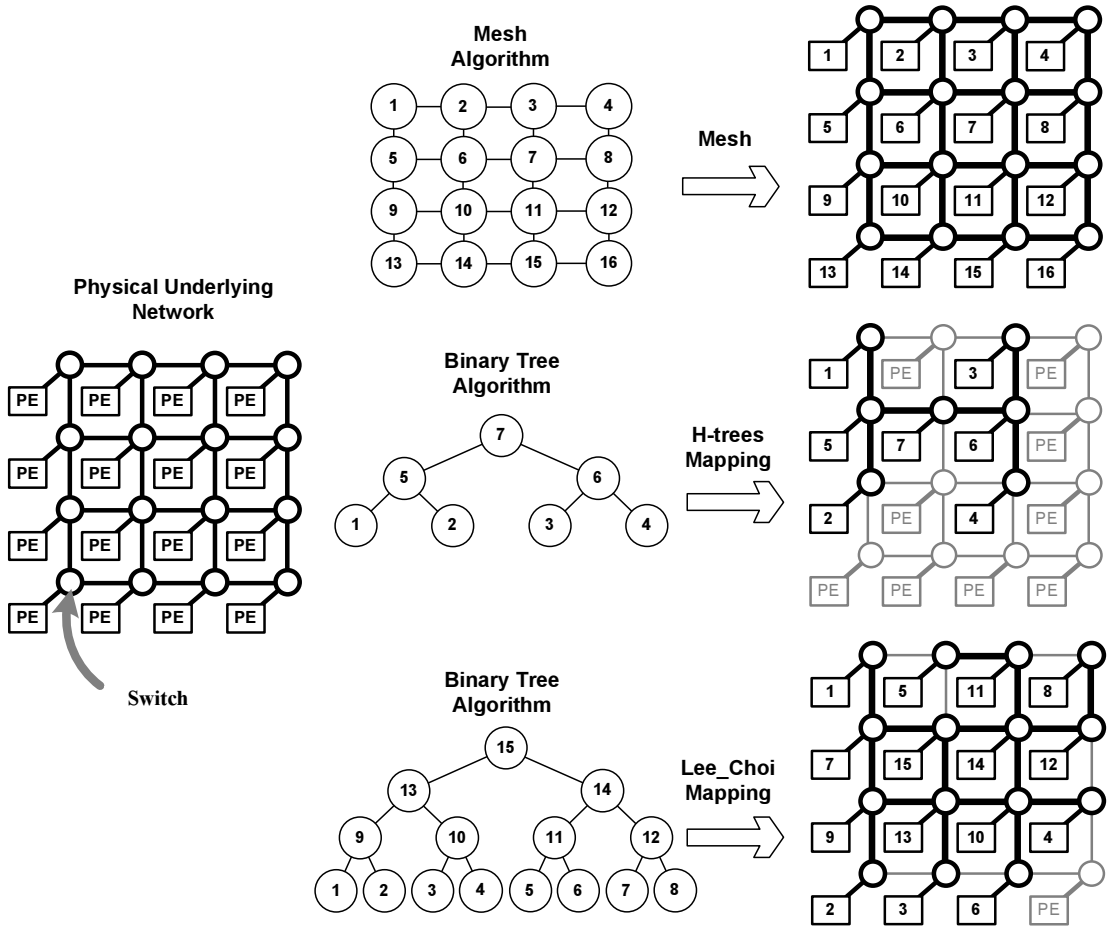

Fig. 4. Binary tree and mesh logical structures mapped on a 2-D mesh physical underlying network.

routing resources). Furthermore, The PE interconnections can be build on dynamically established connections (*chaotic network*) if some specific conditions are satisfied. This approach discards the notion of fixed network topologies and allows to directly interconnect PEs based on the communication requests of the application and the available connections (resources).

- **Multi-chip multiprocessor systems:** Finally, the above schemes can be applied in multi-chip multiprocessor systems.

III. Experimental Results

In this section, we provide evidence suggesting the viability of our proposal when the underlying network is fixed. We evaluate three sample parallel problems using logical interconnects that are binary trees (BT) or 2-D meshes. The physical interconnections are assumed to be a 2-D mesh. That is, for specific mesh logical topologies the links are *physical* = *logical*, while for the BT logical topologies usually *physical* $\neq$ *logical*. We use a regular physical structure rather than irregular only as an example and for simplicity of discussion (most readers are familiar with such structures and there is plenty of literature for mapping a regular network structure into another also regular structure). Figure 4 illustrates the above, where on the left-hand side column is the parallel system composed of its processors and the physical interconnection network, the middle column is the logical BT and mesh structures, and on the right-hand column are mappings of these structures into the physical network.

A. Embedding a Binary-Tree into a 2-D Mesh

Efficient strategies and algorithms can be developed to map algorithms in multiprocessor systems and several researchers discuss embedding one interconnection network into another [3], [5]–[7]. In order to evaluate the performance of an algorithm developed for BTs into a 2-D mesh interconnection, we first need to use an algorithm that maps the BT into the 2-D mesh. Next, we describe two different ways of embedding a BT topology into a mesh and analyze their advantages and disadvantages.

Lee and Choi mapping: The first mapping algorithm, proposed by Lee and Choi [8], results on a maximum congestion[1] of 2 when a BT with $2^p - 1$ nodes is mapped into a $\sqrt{2^p} \times \sqrt{2^p}$

[1]When embedding topology A into topology B, edge congestion is the maximum number of A edges, mapped onto any B edge.

mesh (optimum expansion[2]). The dilation[3] of this mapping is $\frac{D}{2}+1$ for the edges between the 2^{nd} and 3^{rd} level of the tree, where D is the dimension of a $D \times D$ mesh. In many cases however, BT networks suffer from a communication bottleneck at higher levels of the tree [9], [10] and, when mapped into a mesh with such a dilation, the communication bottleneck becomes even greater.

H-trees: Another way of mapping a BT into a mesh is the well known H-trees described in [11]. H-trees result on edge congestion one and a smaller dilation ($\frac{D+1}{4}$) compared to the previous algorithm. On the other hand, the expansion of the mapping is asymptotically twice the optimum, since a $(2^{\frac{p+1}{2}}-1) \times (2^{\frac{p+1}{2}}-1)$ mesh is required to map a BT of (2^p-1) nodes.

B. Evaluation of Three Case Studies

In this section, we evaluate the performance of the 2-D mesh on three parallel problems (case studies), more suitable when solved in a BT topology. For each case study, we utilize a mesh algorithm and one or more BT algorithms. In order to run the BT algorithms, we map the BT topologies into the mesh ones using the mappings described above.

The Maximum: Given a set of n numbers (in our experiments n: 2^{13}, 2^{16} or 2^{20}), the goal in this case study is to find the greatest number in the set. Three algorithms are used, two for BTs and one for meshes:

- **MaxBT1:** Each one of the $\frac{p}{2}$ leaf BT nodes is loaded with a smaller subset of $\frac{n*2}{p}$ numbers. Each cycle, one element of each subset is compared with the results of the other nodes ($\frac{p}{2}$ elements in total). The root node keeps the partial maximum and compares it with the partial results coming next in a pipelined fashion. The maximum number of the set is found when all the elements of the subsets have been compared through the tree.
- **MaxBT2:** The set is divided into $(p-1)$ smaller subsets and loaded onto the $(p-1)$ processors. Each processor finds sequentially the maximum on its data subset which consists of $\frac{n}{p-1}$ numbers. This maximum is compared to the results of other nodes. The tree structure is used to obtain the maximum number of the set by passing only the maximum number from each subtree.
- **MaxME:** Similar to the above algorithm, the set is divided into p smaller subsets and loaded onto the p processors. We merge the partial results first row by row and then column by column, until we obtain the maximum number of the entire set.

We evaluate the first two algorithms in BTs (BT_MaxBT1 and BT_MaxBT2) and in meshes using Lee_Choi and H-tree mappings (ME_LeeChoi_MaxBT1, ME_LeeChoi_MaxBT2, ME_Htree_MaxBT1, and ME_Htree_MaxBT2), and the third algorithm in meshes (ME_MaxME). Figure 5 depicts the

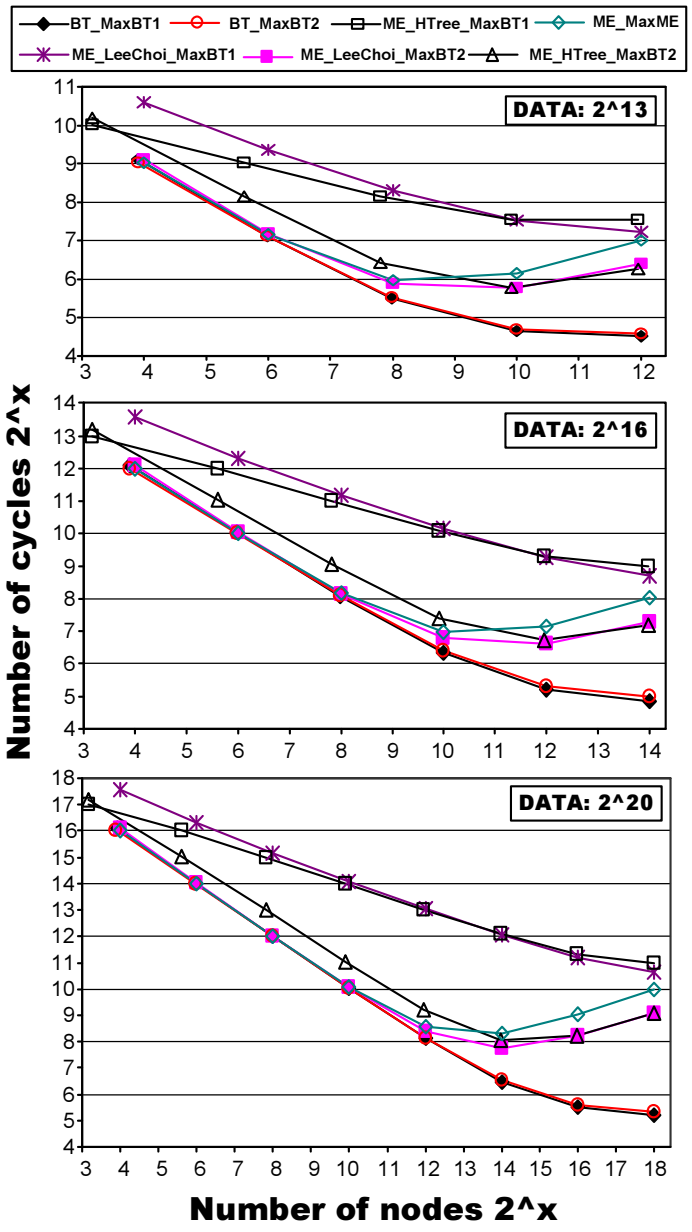

Fig. 5. Performance of the maximum case study for different algorithms, data sizes and number of nodes. (For this graph and the graphs below, we use ^ to denote "the power of"; i.e. 2^x = 2^x)

total number of cycles required to execute the algorithms for different sizes of data sets and number of nodes. The MaxBT1 requires more communication than the other algorithms, since the maximum is calculated throughout the tree instead of having each node processing a subset sequentially. Therefore the MaxBT1 algorithm when running on a mesh has up to 4-32 times higher latency than a BT. On the contrary, the MaxBT2 adapts better into the mesh mappings. For the MaxBT1 algorithm H-trees are better (up to 2×) than Lee_Choi mapping (for small and medium systems), since H-trees have lower dilation. However, when the total number of nodes increases and the processing data remain constant then the Lee_Choi mapping

[2]When embedding topology A into topology B, expansion of the mapping is the ratio of number of the B nodes to the number of A nodes. For the above mapping, that is $\frac{2^p}{2^p-1}$

[3]When embedding topology A into topology B, dilation is the maximum number of links in B that any edge of A is mapped onto.

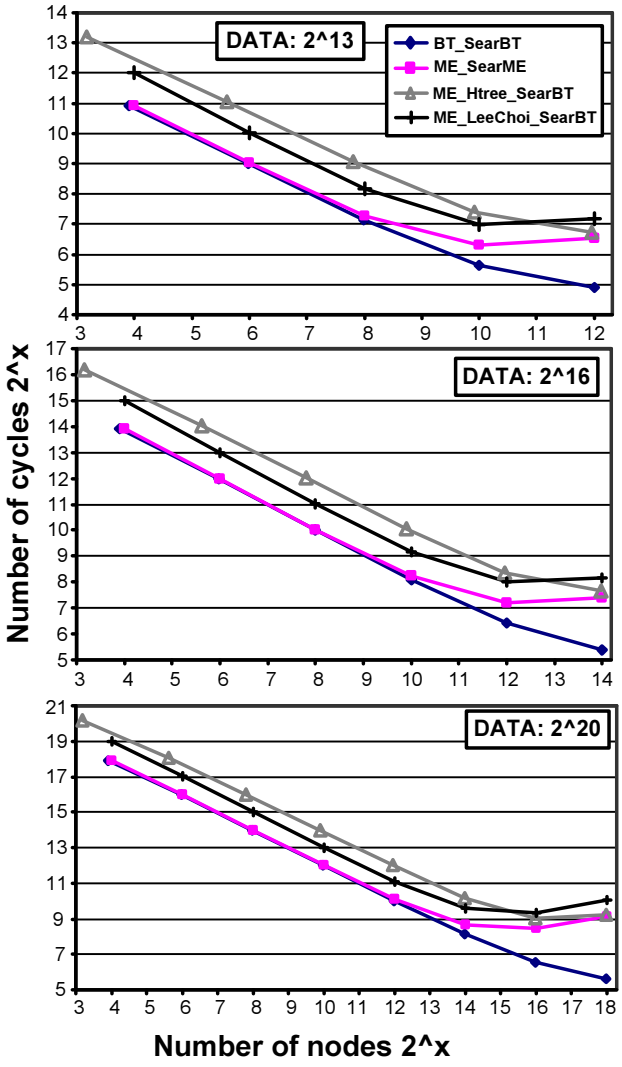

Fig. 6. Performance of the searching case study for different algorithms, data sizes and number of nodes.

is better (up to 50%) since the total number of utilized nodes (expansion) is more important. When running the MaxBT2 algorithm, the Lee_Choi mapping becomes better because the diameter of this mapping is smaller. That is because although Lee_Choi mapping has higher dilation, the average number of mesh edges required per BT edge is lower. Additionally, Lee_Choi mapping exploits almost all mesh nodes, while H-trees have worse expansion. The MaxME is almost as good as the BTs for small number of nodes, but when the system gets larger has up to $2\times$ worse performance even compared to any mapping of the MaxBT2 algorithm into the 2-D mesh. Finally, the size of the processing data affects performance. For example, for smaller data sets the ME_LeeChoi_MaxBT1 gets more efficient than the ME_Htree_MaxBT1 for large systems, while the point (#nodes) where it starts being better differs for different data sets.

Searching: The purpose of this case study is to search for m specific numbers on an unsorted sequence S of n numbers

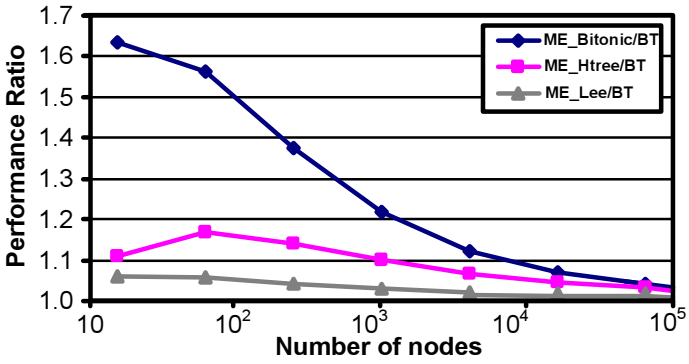

Fig. 7. Performance ratio between sorting on a 2-D mesh (using different mappings and algorithms) and on a binary-tree.

(n= 2^{13}, 2^{16} or 2^{20}, m = 8). If such a number is in S, the searching algorithm outputs the position of the matched number, otherwise, the output is zero. The implementation of this case study is similar for the BT and mesh topologies, **SearBT** and **SearME** respectively. The set is divided into small subsets of $\frac{n}{p}$ numbers. Each of these subsets is processed by a single node and the partial results are sent towards the root node.

Figure 6 depicts again the number of cycles spent for the execution of the searching algorithm. In this case, the gap between the binary-tree (BT) and the meshes is smaller because the searching algorithm requires more processing, $O(nm)$ instead of $O(n)$, while the percentage of the total time spent for communication is smaller compared to the MaxBT2 algorithm. The SearME is up to $4\times$ better than the SearBT algorihm mapped into a mesh. For the SearBT algorithm, the Lee_Choi mapping is generally better than the H-trees (about 50%), however, for large systems the H-trees achieve similar or better performance. Again the size of the data set affects performance. For example, the SearME algorithm (running on a mesh) for medium systems ($2^8 - 2^{12}$ nodes) follows the BT_SearBT performance when processing large data sets, while for smaller data sets has higher latency.

Sorting: Given a sequence of n numbers, a sorting algorithm will produce a sorted sequence of the same input set S. For sorting in BTs we use the algorithm described in [12], [13] (denoted here as **SortBT**) and is performed as follows: the set of numbers S is divided into smaller subsets and loaded onto the leaf processors. Each processor executes a sequential quick sort algorithm on its data subset; parallelism is achieved by having all leaf processors work on their portion of data at the same time. The smallest element of each sub-sequence is sent towards the root node. The set S is sorted when all the elements are sent out through the root node. For sorting in meshes we implemented the bitonic sort as described in [14]. This mesh algorithm (**Bitonic**) sorts n^2 numbers on a $n \times n$ mesh. Therefore, in order to have a fair comparison between the two algorithms, the set contains as many numbers as the number of mesh nodes.

We evaluate the bitonic sort in meshes and the SortBT

algorithm in BTs mapped into meshes (using Lee or H-tree mapping). Figure 7 illustrates the performance ratio between the latency of the above cases and the SortBT when running in the original BT. In each case, the time spent to load and unload data into/from the system is included in the overall latency. Clearly, the bitonic sort in meshes is less efficient than the SortBT in Lee and H-tree mappings. However, when the size of the processing data increases (along with the number of nodes) the performance difference between the three cases and the BT topology becomes insignificant, since the load/unload latency is the dominant factor. *Contrary to the MaxBT1 algorithm, for sorting the Lee mapping is better than the H-trees*. That is because in this case the expansion of the mapping is more significant than the dilation.

Some of the reasons why FLUX Networks are beneficial: We discuss, next, some advantages of the proposed FLUX networks:

- Definitely, when a single algorithm is ported into a physical network (designed to match the algorithm) then it will be faster. That is an algorithm communication needs might match the physical interconnect. Generally speaking, this is a difficult task since the algorithm developer has to have in mind the technology details of the physical network. Furthermore, multiple algorithms should be able to efficiently run on a single multiprocessor system, and if there is an one-to-one mapping for one network, this will not be the case for others. Therefore (as shown by the example mappings) the interconnection network should be adaptable to achieve more benefits.
- Software portability: for a given technology an algorithm may match the physical network. However, for the next device family (new technology) the algorithm won't match the new physical structure. In this case the algorithm communication needs become the "logical" network that has to be efficiently ported into the new physical structure, implying that generally speaking the FLUX networks are the most beneficial solution.
- When the logical and the physical networks do not match, the algorithm usually cannot exploit all the physical network resources. That is, because of lack of technology knowledge, an algorithm developer has difficulties in achieving optimum mappings. Therefore, using directly the physical structure may not improve performance and will possibly increase complexity. In FLUX networks, both users and developers of technologies are involved improving the networking.
- The FLUX networks offer the ability to adapt the physical underlying network to the application needs. More precisely, based on the parameters that affect the application performance (underlying network, number of nodes, data size, etc.), it chooses the best mapping (pre-selection) of the logical network to the physical one. Our experiments in a rigid physical underlying network (2-D mesh) show that Lee mapping is better for the MaxBT2, while the H-trees is more efficient for MaxBT1 algorithm. Actually, the performance can be 1.5-2$\times$ higher, when the best mapping is followed.
- In FLUX networks, direct "point-to point" connections can be utilized to detect and change any wrong decisions of the application developer regarding the communication needs of the application. Hot spot connections of the network can also be added (see also Figure 2).
- We propose that designer, system programmer and application developer should be involved in a $complimentary\ \ fashion$. The hardware designer maximizes physical network flexibility to accommodate mapping arbitrary "spaghetti" logical networks. The system programmer finds the most suitable mapping/utilization of the network, exploiting the flexibility of the provided FLUX network and gives feedback to the algorithm developer regarding the performance tradeoffs of different network decisions. The application developer utilizes several techniques (profiling, monitoring etc.) to find the most suitable interconnect for the targeting problem.
- The FLUX networks allow physical network descriptions to coexist with common programming constructs. For a single application running on a single physical network the best mapping can vary. FLUX networks provide the ability to detect and change the mapping of the application into the physical network *on-the-fly* (multiple mappings), and therefore, can exploit in each case the best network configuration. The above is not supported by previous work [7], [15].
- Contrary to others [15], FLUX Networks on reconfigurable fabric can reconfigure the PE routers, changing the routing algorithm, the number and width of the links, add buffering etc. on demand instead of being prefixed.
- Our approach can $dynamically$ adapt to $arbitrary$ topologies, while other solutions can only support $several$ topologies and regular predefined structures [15].

C. A Programming Example

In this section, we present a programming example, showing the way to port an application/algorithm in different underlying networks. In our example, the underlying network is either a $n \times n$ 2-D mesh, an FPGA or a BT interconnection network. The utilized application is the MaxBT1 algorithm described in the previous Section III-B. The program decides which mapping to use according to the following parameters: *the underlying physical network, the processing data size and the number of nodes*. For different applications or physical networks, other parameters might also be considered (node size, network area cost etc.) Assuming that a 2-D mesh, a BT, or an FPGA is the underlying physical network, Figure 8 illustrates the programming function that decides the interconnection setup for the MaxBT1 algorithm.

The above function is based on the results of Figure 5. When the underlying network is a binary tree (BT) then obviously it is more efficient to use the physical topology itself. In case of the FPGA interconnection, our experiments show that when a BT is implemented, it has a similar cycle time with the

```
SetNet_MAXBT1:

    CASE (PHY Net) { // what is the physical network?
    BT: // if the Physical Network is a binary tree
        SET BT; // then map a binary tree

    2-D Mesh: // if the Physical Network is a 2-D Mesh
        CASE (#nodes){
                // if the system has up to 2^10 nodes
            (#nodes <= 2^10):
            // then map a binary tree using H-trees
                SET H-trees mapping;
              // if the system has more than 2^10 and up to 2^12 nodes
              (2^10 < #nodes <= 2^12):
            // and the processing data size is upto 2^10
                IF(Data <= 2^10)THEN
            // then map a binary tree using H-trees
                    SET H-trees mapping;
                ELSE
            //else map a binary tree using Lee_Choi mapping
                    SET Lee_Choi mapping;
              // if the system has more than 2^12 and upto 2^14 nodes
              (2^12 < #nodes <= 2^14):
                    // and the processing data size is upto 2^16
                IF(Data <= 2^16)THEN
                  // then map a binary tree using H-trees
                    SET H-trees mapping;
                ELSE
            //else map a binary tree using Lee_Choi mapping
                    SET Lee_Choi mapping;
            // if the system has more than 2^14 nodes
            (#nodes > 2^14):
            // then map a binary tree using Lee_Choi  mapping
                SET Lee mapping;
        }

    FPGA: // if the Physical Network is Reconfigurable
                SET BT; // then map a binary tree
    }
```

Fig. 8. A programming example for the MaxBT1 algorithm.

2-D mesh and requires about 70% less resources.Therefore, based again on the performance results of Figure 5, it is more efficient to utilize the BT topology. For the 2-D mesh physical network, the most efficient mapping depends on the number of nodes and the processing data size. More precisely, for small number of nodes or medium systems and small data sets the H-trees are better, while for large number of nodes or medium systems and large data sets the Lee_Choi mapping is more beneficial. Finally, when a specific topology/mapping is decided (e.g. *SET* H-trees mapping into a 2-D mesh), at least the following parameters should be explicitly specified:

- Node Addressing: assign each physical node with an address and a workload.
- Establish Routing paths: specify the communication path between every pair of (utilized) nodes.
- Routing algorithms/policies: specify routing algorithms and policies (i.e priorities of connections), if can be supported by the physical network.

IV. Conclusions

In this paper, we introduced the concept of the FLUX networks and have discussed some performance potential for parallel applications suitable for different interconnection topologies/mappings. We studied different types of physical interconnections and presented a programming paradigm as a way to accomplish the configuration (mapping) of an interconnection network on demand. In addition, we presented some experimental results to show that, when running a parallel algorithm in a multiprocessor system interconnected in a fixed topology, performance is affected. More precisely, we showed that the performance of a parallel algorithm drops when using other mapping than the appropriate one. We also pointed out that, besides the implemented algorithm, other parameters such as the data size, the underlying technology and the number of nodes should be taken into account in order to decide which topology is most suitable for an application. The implication of the above is that by determining the network in advance and by exploiting network instalments (statically or dynamically) substantial gain can be expected.

References

[1] S. Vassiliadis, L. A. Sousa, and G. N. Gaydadjiev, "The Midlifekicker Microarchitecture Evaluation Metric," in *Proceedings of the IEEE Int. Conf. ASAP05*, July 2005, pp. 92–97.

[2] W. Bouknight, S. Desenberg, D. McIntyre, J. Randall, A. Sameh, and D. Slotnick, "The Illiac IV system," *Proc. IEEE*, vol. 60, April 1972.

[3] S. Ranka and S. Sahni, *Hypercube Algorithms for Image Processing and Pattern Recognition*. New York City, NY: Springer-Verlag, 1990.

[4] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The Molen Polymorphic Processor," *IEEE Transactions on Computers*, pp. 1363– 1375, November 2004.

[5] F. T. Leighton, *Introduction to parallel algorithms and architectures: array, trees, hypercubes*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[6] M. Reingold, J. Nievergelt, and N. Deo, *Combinational Algorithms: Theory and Practice*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1977.

[7] B. Monien and I. Sudborough, "Embedding one interconnection network in another," *In Computational Graph Theory, G. Tinhofer et al. Eds., Computing Supplementa, vol. 7*, pp. 257–282, 1990.

[8] S.-K. Lee and H.-A. Choi, "Embedding of Complete Binary Trees into Meshes with Row-Column Routing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 5, pp. 493–497, 1996.

[9] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*. USA: Pearson Educ. Lim., 2003.

[10] C. E. Leiserson, "Fat-Trees: universal networks for hardware-efficient supercomputing," *IEEE Trans. Comput.*, vol. 34, no. 10, pp. 892–901, 1985.

[11] S. A. Browning, "The Tree Machine: A Highly Concurrent Computing Environment," Ph.D. dissertation, Dept. of Computer Science, CalTech, 1980.

[12] T. H. Cormen, E. Leiserson, Charles, and R. L. Rivest, *Introduction to Algorithms*, ser. Cambridge. Massachusetts: The MIT Press, 1990.

[13] J. G. Delgado-Frias, S. Vassiliadis, C.-L. Chu, and A. de Luca, "DT: A Binary Tree Parallel Computer with Distributed I/Os," *Journal of the Mexican Society of Instrumentation 3 (4)*, pp. 33–42, January 1994.

[14] C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected parallel computer," *Commun. ACM*, vol. 20, no. 4, pp. 263–271, 1977.

[15] L. Snyder, "Introduction to the Configurable, Highly Parallel Computer." *IEEE Computer*, vol. 15, no. 1, pp. 47–56, 1982.