JOURNAL OF SYSTEMS ARCHITECTURE
http://ees.elsevier.com/jsa
ISSN 1383-7621
Volume 53, Number 10, October 2007, pp. 677–794
JSA
SPECIAL ISSUE:
EMBEDDED COMPUTER SYSTEMS: ARCHITECTURES, MODELING, AND SIMULATION
GUEST EDITORS:
T.D. HÄMÄLÄINEN, S. WONG, J. GLOSSNER and S. VASSILIADIS
The EUROMICRO Journal

Available online at www.sciencedirect.com
ScienceDirect
Journal of Systems Architecture 53 (2007) 777–793
JOURNAL OF SYSTEMS ARCHITECTURE
www.elsevier.com/locate/sysarc

FLUX interconnection networks on demand [☆]

Stamatis Vassiliadis, Ioannis Sourdis [*]

Computer Engineering, TU Delft, The Netherlands

Received 31 December 2006; accepted 17 January 2007
Available online 25 January 2007

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 and the potential performance gain of using the most suitable network configuration for a given parallel program. We experiment on several case studies, evaluate different algorithms, developed for meshes or trees, and map them on "grid"-like or reconfigurable 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, the number of utilized nodes or the hardware cost of the processing elements changes. The implication of the above is that changing interconnection topologies/mappings (dynamically) on demand depending on the program needs can be beneficial.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Interconnection networks; Multiprocessor parallel systems

1. 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 conjectures 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 [2]. 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 $x \geqslant 2$). Multiprocessor multichip parallel systems are not new (e.g. see ILIAC IV [3]), and it will appear that using

[☆] This work was supported by the European Commission in the context of the Scalable computer ARChitectures (SARC) integrated project #27648 (FP6). This paper is an extended version of [1].

[*] Corresponding author.
E-mail addresses: stamatis@ce.et.tudelft.nl (S. Vassiliadis), sourdis@ce.et.tudelft.nl (I. Sourdis).
URL: http://ce.et.tudelft.nl/~sourdis/ (I. Sourdis).

1383-7621/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2007.01.006

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 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 2 we discuss previous solutions in interconnects of multiprocessor parallel systems and point out their performance drawbacks. In Section 3, we introduce the FLUX networks, present several implementation schemes 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 4, we provide initial experimental data supporting our approach. Finally, in Section 5 we present our conclusions.

2. Background

Currently, multiprocessor systems are designed based on a specific hardwired interconnect topology. That is, the designer provides the physical structure of the interconnects having in mind a regular network topology such as crossbar, cube, fat-tree, etc. Furthermore, the network structure is fixed and rigid. For example, once the designer fixes the link width, it will remain the same for the entire life time of a parallel system. Additionally, since the physical structure of the network is rigid, the way communications occur may be restricted. Even when it is found that different communication/network schemes will be more beneficial to achieve better performance because of the rigid network restrictions (e.g. fixed buffer space, bus width, etc.) in most circumstances the benefits can not be achieved. Clearly for these circumstances a different physical organization is required and such an organization can not be accommodated by fixed networks. For example, as depicted in Fig. 1, an application at time "t" requires a 2D mesh topology, while at time "$t+1$" the lower processing elements (PEs) need to transfer large amount of data

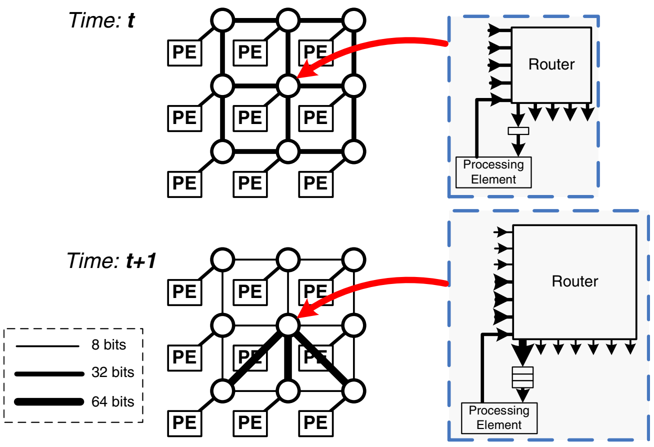

Fig. 1. Adapting interconnects on demand.

to the middle PE. A fixed/rigid network would not be able to alleviate these communication requirements, resulting in substantial performance drawbacks. On the contrary, in reconfigurable fabrics the interconnects can be reconfigured, e.g. changing the PE router, link width and buffering size, (and possibly the communication scheme and algorithm) to accommodate the communication traffic. In the example of Fig. 1, extra links and buffering in the middle PE are added and several PE routers change. In addition, the width of several links changes, that is, critical links become wider (64 instead of 32 bits), while links which are not often used become narrower (8 instead of 32 bits). This way, in FLUX networks the hardware resources are better utilized to facilitate the communication requirements of the current parallel application/program phase and maximize performance.

Obviously, some classes of applications benefit from a specific physical structure. A general purpose parallel system is build however to accommodate a multiplicity of application classes. Given that a provided interconnection network and communication scheme does not fit a pre-specified interconnection mechanism, not all applications can substantially benefit from parallel processing.

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. When mapping is performed, several parameters have to be taken into account such as congestion, dilation and expansion [4], while how successful this mapping is determines how efficient the communication will be and therefore the performance of the entire system.

To alleviate performance penalties, numerous researchers have provided algorithms for mapping communication networks needed for an application on to different interconnections [5,4,6,7]. Considering VLSI chip structures, the current designer practices may not be the most appropriate. Currently, algorithms should be created to suit the multiprocessor system topology in order to maximize performance. Alternatively, we propose *the interconnection network to be provided (dynamically) on demand to fit an algorithm's/program's communication needs*. In order to allow for on demand interconnection networks, connections have to be "adapted". This is possible because reconfigurable technologies have an underlying network that can be "modified". Consequently, it may be of benefit for multiprocessors using reconfigurable fabric, to not commit in advance the underlying network structure into specific interconnects.

3. FLUX interconnects on demand

In FLUX Networks, the network is the one to be adapted instead of the parallel programs. To do so, the underlying physical network requires to provide higher flexibility than the current fixed networks. Obviously, this flexibility comes at the expense of delay and possibly area overhead, which is a fair price to pay, just like in previous experience in general purpose computers. Concerning the delay overhead, it is a fact that an application of a logical network "*A*" when ported in a fixed network "*A*" will execute substantially faster than in the FLUX Networks. However, when other parallel applications of different logical networks (which is the general case) are ported into a fixed network "*A*" and the FLUX Networks, then the latter may be faster since it can adapt to any new communication needs. The FLUX Networks require increased hardware resources to provide flexibility and accommodate arbitrary network installments. However, technological improvements provide more metal layers which may be used for the FLUX network underlying physical network. In addition, fixed networks also require significant amount of hardware resources. Each fixed network switch may often be as large as the 32-bit RISC processor it serves. A fixed network often uses complicated (adaptive) routing algorithms and large buffers (i.e. packet/reorder buffers) to overcome the fact that it is fixed (cannot match multiple logical networks) and to bypass congested communication hot spots. Consequently, fixed networks also require significant area resources, may increase network latency, and possibly need techniques such as packet reordering to guarantee correct communication. On the contrary, in FLUX Networks *simple network structures* can be efficient to achieve high performance for a given traffic pattern, since they can be reinstalled and adapted again to any new communication pattern.

To exemplify our approach, consider the multiprocessor system of Fig. 2 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 pre-determined 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 FLUX Networks, PEs interconnects can change during the execution of a single program. In case 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 2D 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.

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 Fig. 2, unused links can be used to connect two PEs additionally to the utilized interconnection network (Fig. 3). 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 fin-

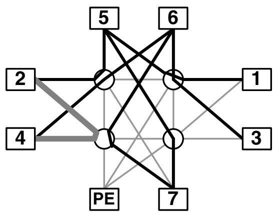

Fig. 3. Direct connections additional to the network topology.

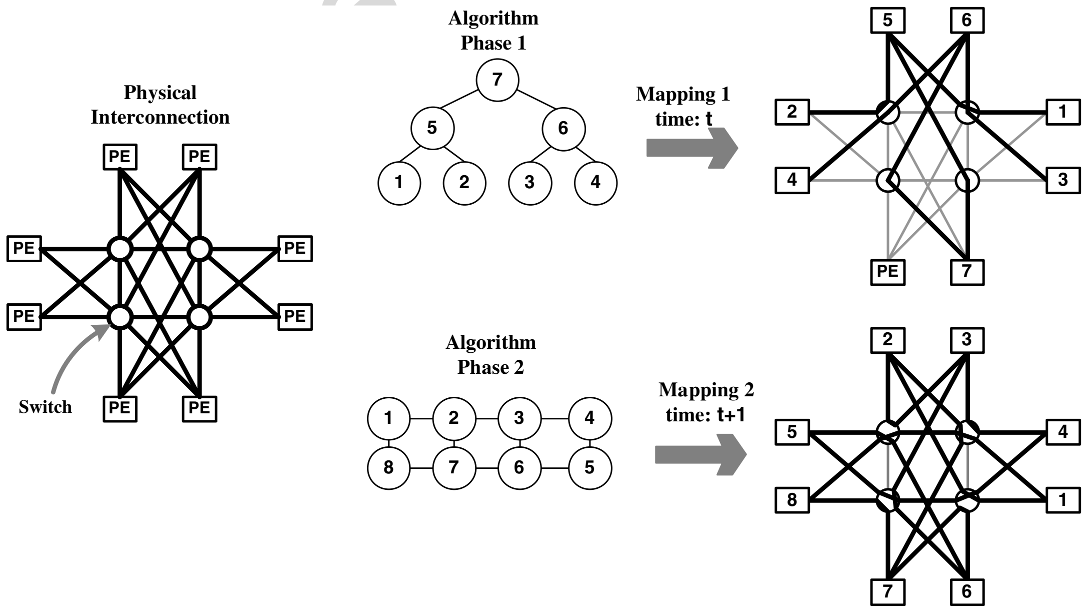

Fig. 2. FLUX network on demand.

ished. That is, if on a specific time "processor 2" needs to communicate with "processor 4" without going via the existing BT network (that is for the mapped BT the path through processors: 2–5–7–6–4), because of a critical event, then a direct connection is established (and afterwards released) on demand.

3.1. *FLUX networks in reconfigurable hardware*

Reconfigurable technologies have an underlying network that can be (dynamically) "modified", thus they are excellent potential for FLUX implementation platforms. Next, 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 [8]. 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 in the next section and the MOLEN ISA extensions [9]. Reconfigurable FLUX networks can be implemented using numerous schemes including (but not limited by) the ones described next.

Reconfigurable Interconnects with static/dynamic PE placement: Fig. 4 depicts a multiprocessor system that consists of several PEs and a reconfigurable part that can interconnect them in different arbitrary topologies. For instance, in the case of an algorithm implemented for binary-trees (BT logical network), this scheme can connect the PEs in a BT topology. For an algorithm that is suitable for a mesh interconnect, the interconnection can be a mesh. Clearly, the topologies will follow different physical links to match the logical structure of each algorithm or phase of a program. The reconfigurable FLUX networks can also be adjusted at run-time during the execution of a single program. The run-time reconfiguration overhead of the network is technology dependent. *When run-time reconfiguration is decided, it should be clear that the performance gain, which results from the network switching, is greater than the reconfiguration overhead, otherwise adapting the interconnects will be proven inefficient.* In this first scheme, each PE consists of two parts, the first part is fixed and executes part of the program, while the second part involves the PE interface with the interconnection network. Since the network is reconfigurable, the interface between the PE and the interconnection network should also be reconfigurable in order to apply different routing algorithms for different topologies. Thus, this latter part of the PE should include a routing module and an interface between the variable number of network links and the processor core.

The above scheme implies that the processing engines are fixed (statically placed, hardcores). Static PE placement may restrict the network routing. To overcome this restriction, an alternative solution is that PEs are softcores (Fig. 5). In this case, the

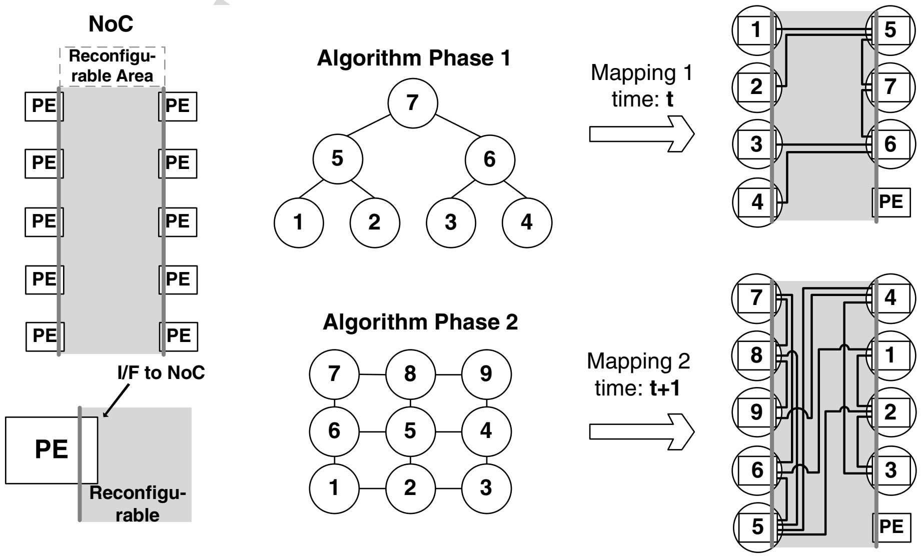

Fig. 4. Reconfigurable FLUX networks with fixed PE placement.

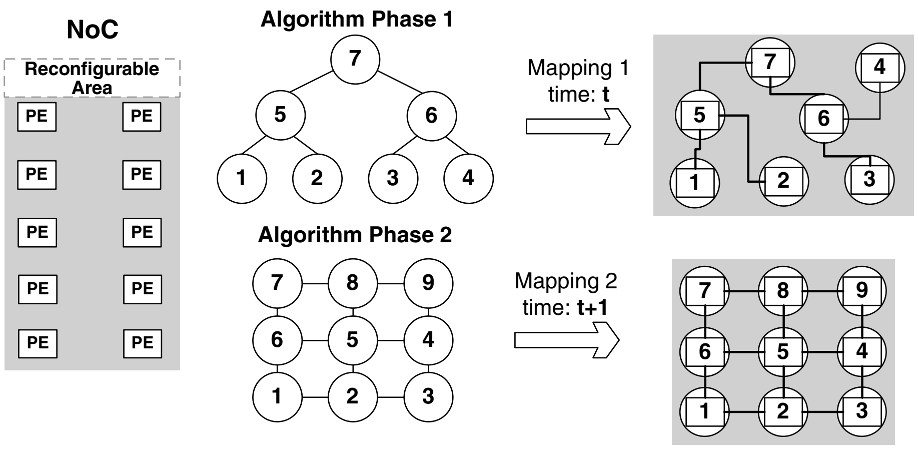

Fig. 5. Reconfigurable FLUX networks without fixed PE placement.

interconnection topology and the PE placement can change over time (in different phases of an application). Consequently, when a network topology is decided, both the network and the PEs are reconfigured. However, this approach introduces different performance and reconfiguration overheads. In order to compare the above two alternatives, we first assume a hybrid technology where the PEs would be implemented in ASIC and between PEs reconfigurable hardware would be available for the interconnect. In this case, the PEs could operate faster than if they were implemented in reconfigurable hardware. Even if both approaches were implemented in reconfigurable hardware, then in the first case the network reconfiguration process could be substantially faster. That is because in the first case the reconfigured area (only the network) is much smaller and can be chosen to be partially reconfigured. Furthermore, the PEs may continue running their part of the algorithm (without communicating with each other), while this does not hold true for the second case.

Direct "point-to-point" & chaotic interconnects: The FPGA routing architectures provide an underlying "unused" reconfigurable network. Consequently, the general *direct point-to point connections* scheme can be applied in reconfigurable hardware. Fig. 6 illustrates a direct point-to-point connection in reconfigurable hardware. The PEs #1 and #5 are directly connected besides the existing Ring topology. This way for example, PE #1 can send data to PE #5 spending a single hop instead of using the ring network and spending four hops. In addition, depending on the amount of available wires/resources the width of the direct "point-to-point" connection *may* be increased so

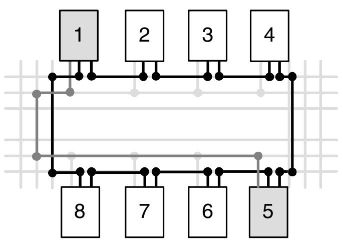

Fig. 6. Direct "point-to-point" connection.

that higher communication throughput can be achieved. The PE interconnections can be build on dynamically established connections (*chaotic network*) if some specific conditions are satisfied. This approach discards any fixed network topology to directly interconnect PEs based on the communication requests of the application and the available connections. Apart from the complex routing of the wires that this solution requires, a second issue is *timing*. Not knowing in advance the wire length of each connection implies that proper mechanisms are required to guarantee correct communication between the PEs (GALS, fully asynchronous connections, etc.). Furthermore, a priori analysis of the routing resources is required to determine the maximum communication load that the interconnection network can handle. For each connection request, a specific methodology should be followed: a routing path should be established; then the data should be sent and last, the connection should be released. Having said the above, in case the underlying structure is partially reconfigurable dynamically in acceptable speeds, the point-to-point and the chaotic networks could be of interest.

Technology considerations: An interesting question regarding what has been presented is which of the proposed mechanisms can be implemented by currently available technologies and which are the directions for making the remaining mechanism a reality. Current technology allows for reconfiguration to be done before program execution. Thus loading an interconnection network before program execution is readily available. Regarding dynamic reconfiguration, we first note that a network is used for substantially long time (e.g. scientific applications) in parallel systems that perform massive data operations with the same network requirements. In [10], we showed that FLUX networks as the traffic load increases can asymptotically approach the theoretical latency of a network that always provides the most suitable topology without any reconfiguration overhead. The suggestion is that, *the reconfiguration overhead is negligible, when a traffic load of specific characteristics runs for sufficient time.*[1] Consequently, it can be suggested that interconnects can be dynamically changed with current technology. Direct point-to-point and chaotic interconnects could be difficult to implement in current technologies because they require fine-grain and fast reconfigurability. However, current technologies such as Xilinx allow partial reconfiguration of relatively large areas, which may span the entire height of a device and a fraction of one column and require few milliseconds [12]. This restriction can provide substantial difficulties for point-to-point and chaotic interconnects. Numerous approaches can be envisioned, however, outside of the scope of the paper, to change current commercial chips and incorporate smaller dynamic reconfigurability slides to achieve point-to-point and chaotic interconnects in the near future.

3.2. Programming paradigm

In our proposal we do not consider using a fixed network for all the parallel applications ported on the system. Instead, we let the program decide at run-time how to more efficiently use the physical connections and which network configuration to install. In order for a network to exhibit these properties, 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 [9]. 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. A master-slave parallel processing model is considered, where the program running on the master processor being responsible for (at least) 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 [9]). As depicted in Fig. 7, 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)

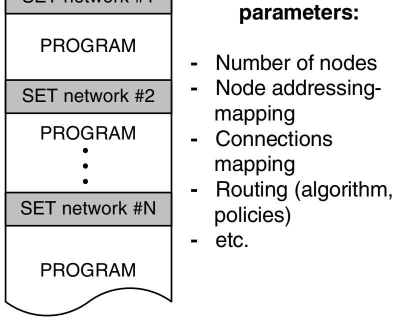

Fig. 7. Execution of the SET instruction before or during different phases of an application.

[1] This time is technology dependent, relative to the reconfiguration time of the device (which in current FPGAs is a few tens of milliseconds [11]) and finally depends on the size of the reconfiguration area.

It should be noted that in difference with 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.

The Molen organization [9] (Fig. 8), can be considered for implementing the FLUX Networks in reconfigurable hardware. When it is needed to configure the network, then a SET ⟨*address*⟩ instruction is necessary. The *SET* instruction utilizes an address to a memory location where the first element of the configuration bitstream is to be loaded from. This way, numerous different network configurations are allowed to be available in the configuration memory. The bitstream may include the configuration of the entire interconnection network or part of it (the partial set P_SET ⟨*address*⟩ Molen instruction). Furthermore, the PEs configuration (including routing information) and possibly the initial data of each local PE memory (instructions and data, etc.) may also be part of the bitstream. Fig. 8 illustrates a possible reconfigurable FLUX network organization using a control processor. The control processor manages the reconfiguration of the reconfigurable multiprocessor system. An arbiter detects the SET instructions and subsequently activates the reconfiguration process utilizing the microcode unit. The bitstream is downloaded from the memory to the reconfigurable unit through the data load/store unit and the data memory multiplexer. When the reconfiguration is accomplished the microcode unit sends a signal to the arbiter and the following instructions are sent to the control processor in order to continue the execution of the remaining program. Finally, the synchronization of the PEs can be accomplished through the exchange registers bank and the MOVTX and MOVFX Molen instructions.

4. Experimental results

In this section, we provide evidence suggesting the viability of our proposal when the underlying network is either fixed or reconfigurable. First, we evaluate several sample parallel problems using logical interconnects that are binary-trees (BT) or 2D meshes. The physical interconnections are assumed to be a 2D mesh. That is, for specific mesh logical topologies the links are *physical* = *logical*, while for the BT logical topologies usually *physical* ≠ *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 struc-

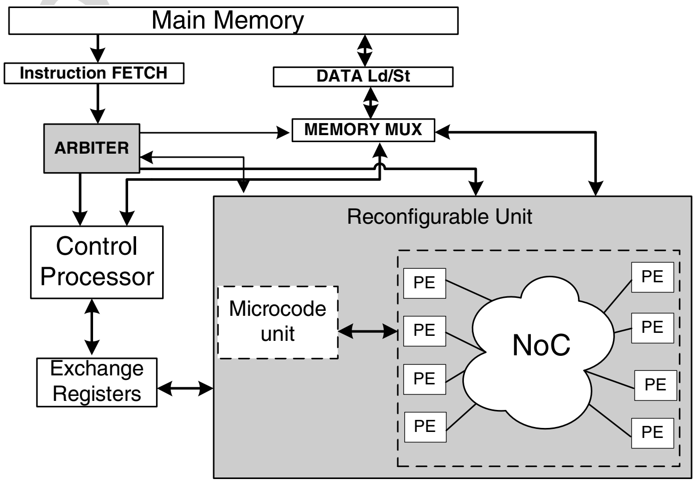

Fig. 8. A FLUX parallel system scheme with reconfigurable FLUX networks.

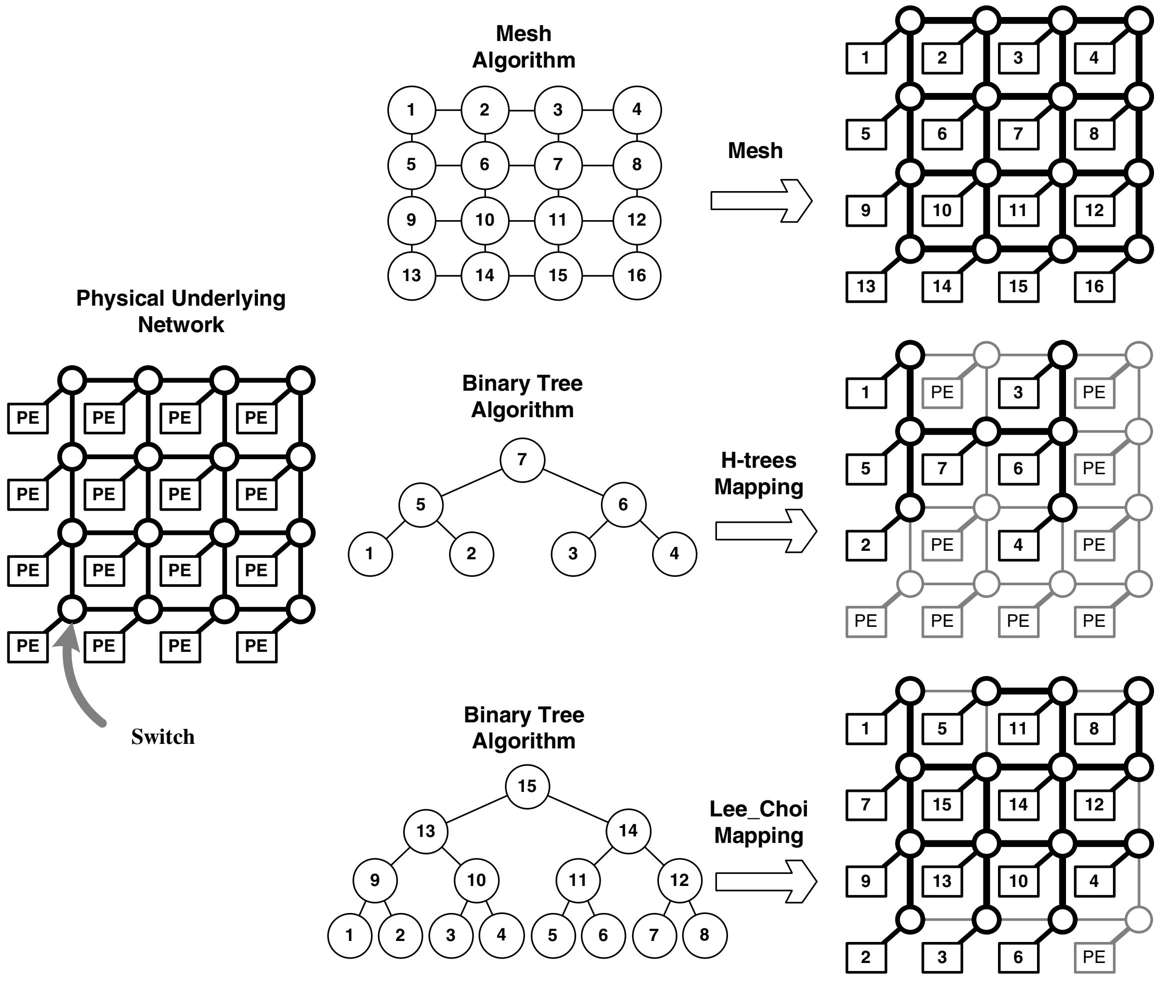

Fig. 9. Binary-tree and mesh logical structures mapped on a 2D mesh physical underlying network.

ture). Fig. 9 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. Second, we use reconfigurable hardware, set a network area constraint, and evaluate a single parallel algorithm when changing parameters such as the processing data size and the PE hardware cost.

4.1. Embedding a binary-tree into a 2D mesh

Efficient strategies and algorithms can be developed to map algorithms in multiprocessor systems and several researchers discuss embedding one interconnection network into another [5,4,6,7]. In order to evaluate the performance of an algorithm developed for BTs into a 2D mesh interconnection, we first need to use an algorithm that maps the BT into the 2D 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 [13], results on a maximum congestion[2] of 2 when a BT with $2^p - 1$ nodes is mapped into a $\sqrt{2^p} \times \sqrt{2^p}$ mesh (optimum expansion[3]). The dilation[4] of this mapping is $\frac{\sqrt{2^p}}{2} + 1$ for the edges between the second and third level of the tree. In many cases however, BT networks suffer from a communication bottleneck at higher levels of the tree [14,15] and, when mapped

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

[3] 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 $\frac{2^p}{2^p-1}$.

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

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 [16]. H-trees result on edge congestion one and a smaller dilation $\left(\frac{\sqrt{2^p}+1}{4}\right)$ compared to the previous algorithm. On the other hand, the expansion of the mapping is asymptotically twice the optimum, since a $\left(2^{\frac{p+1}{2}}-1\right)\times\left(2^{\frac{p+1}{2}}-1\right)$ mesh is required to map a BT of (2^p-1) nodes.

4.2. Evaluation of several case studies

In this section, we evaluate the performance of several parallel problems (case studies), more suitable when solved in a specific topology. For each case study, we utilize either a fixed or reconfigurable physical network. In order to run BT algorithms a 2D mesh physical network, we map the BTs into the mesh networks using the mappings described above. All network performance results are simulated, considering that all networks use packet switching and wormhole routing, routers have single flit buffers, while the minimum latency per router is one clock cycle. Finally, in all cases single flit packets are assumed sufficient for each communication transaction between two nodes.

Find the maximum in BTs and meshes: 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_Htree_MaxBT1, ME_Htree-_MaxBT2, ME_LeeChoi_MaxBT1, and ME_LeeChoi_MaxBT2), and the third algorithm in meshes (ME_MaxME). In this and the next case studies, we consider that the comparison between two elements takes a single clock cycle, while a single communication transaction is a single flit packet. Fig. 10 depicts the 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 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× worse performance even compared to any mapping of the MaxBT2 algorithm into the 2D 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.

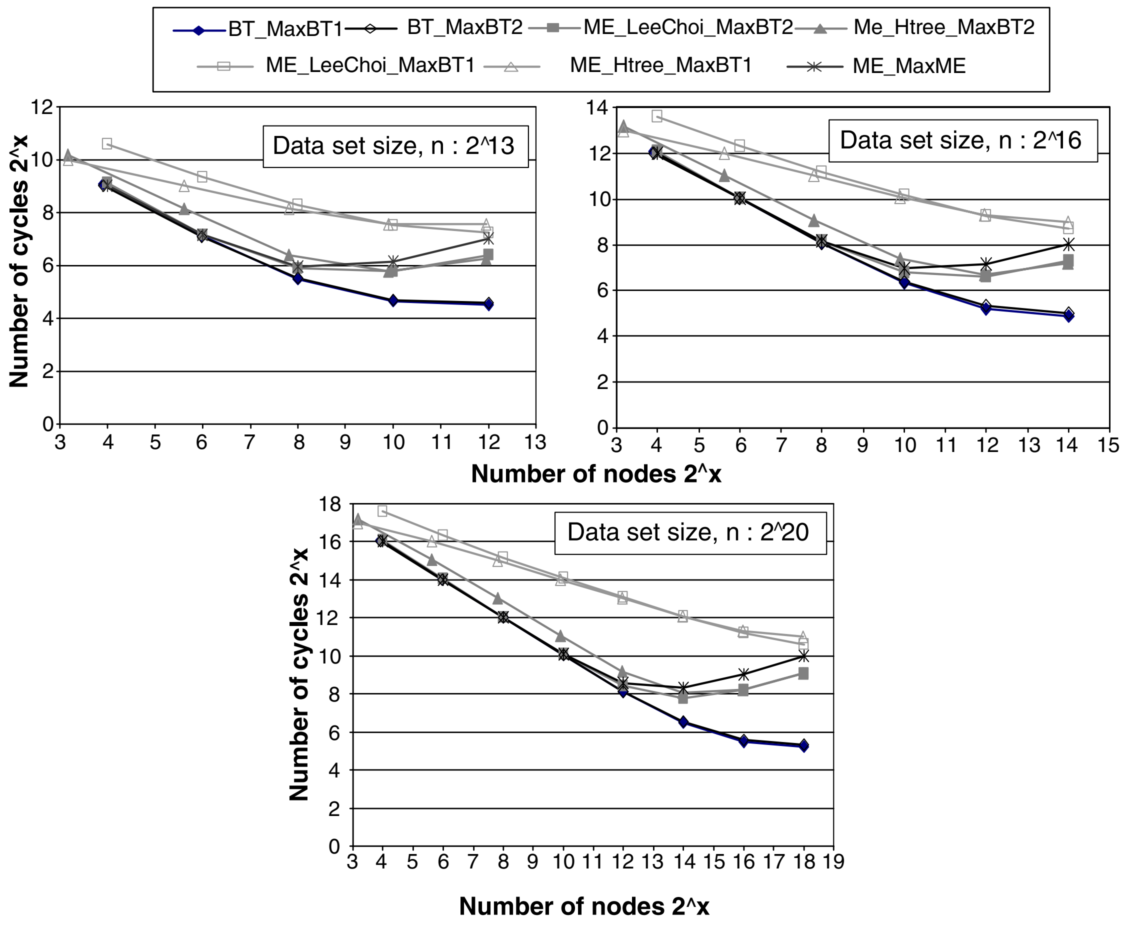

Fig. 10. Performance of the maximum case study for different algorithms, data sizes and number of nodes. (For this graph and the graphs in Figs. 11 and 13, we use ^ to denote "the power of"; i.e. $2^\wedge x = 2^x$.)

Searching algorithms in BTs and meshes: The purpose of this case study is to search for m specific numbers on an unsorted sequence S of n numbers ($n = 2^{13}$, 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.

Fig. 11 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× 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 execution time.

Sorting algorithms in BTs and meshes: 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 [17,18] (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.

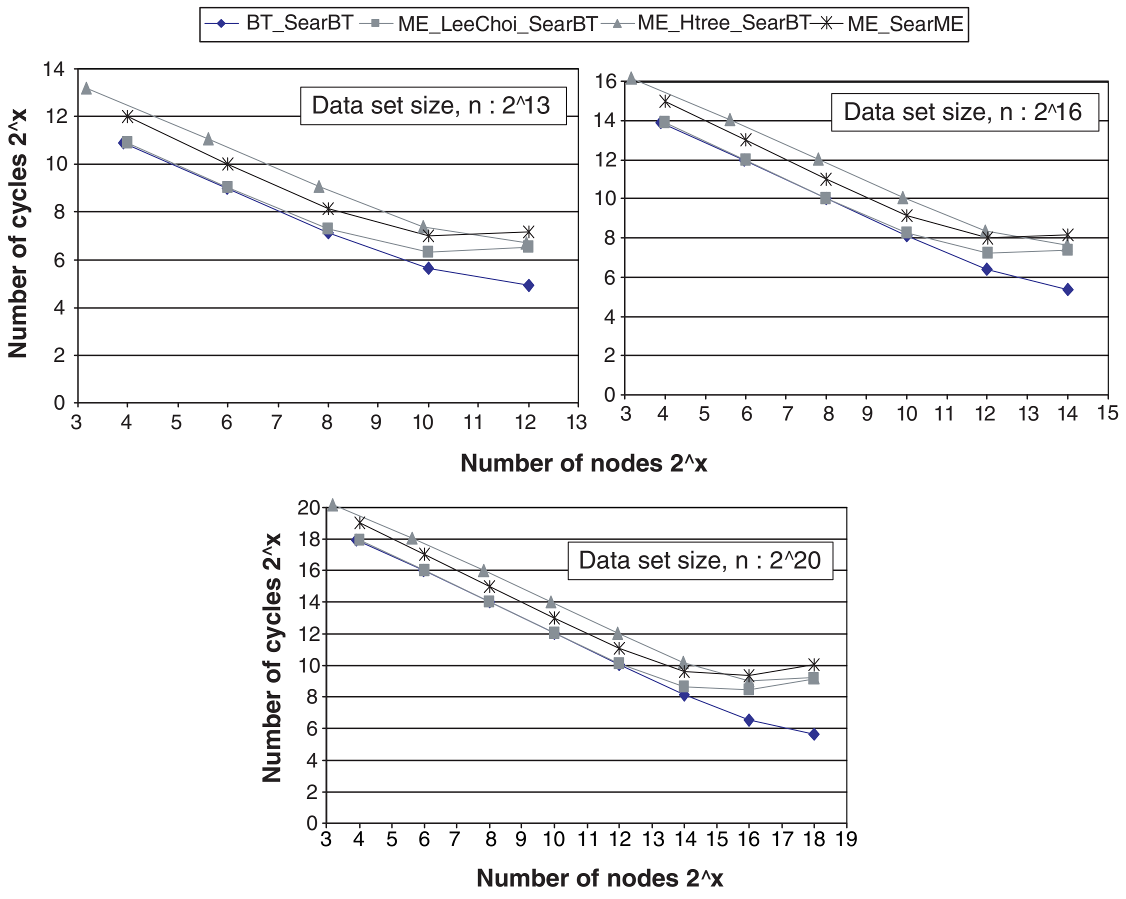

Fig. 11. Performance of the searching case study for different algorithms, data sizes and number of nodes.

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 [19]. 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). Fig. 12 illustrates the ratio between the execution 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 runtime ratio between the three cases and the BT topology decreases. *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.

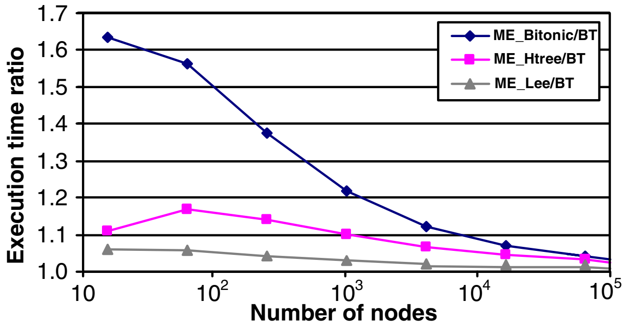

Fig. 12. Execution time ratio between sorting on a 2D mesh (using different mappings and algorithms) and on a binary-tree.

The SortBT algorithm in reconfigurable hardware: In this case study, we utilized the SortBT algorithm described above and reconfigurable hardware as the FLUX implementation platform. We implemented the routing structures of an 8-level binary-tree (BT) and 7 and 6-level fat-trees (FT) using 32-bit words. Table 1 depicts the characteristics of these

Table 1
Implementation results: 8-L binary-tree & 6,7-L fat-trees

	Link width	Routing area	Total area,	Total area	Frequency	Diameter # hops	#Nodes	#Links	Max. links
		logic cells	PE = 29LC,	PE = 357LC					per node
8L-BT	32-bit	22,008	**29,403**	**113,043**	254	14	255	254	3
7L-FT	32-bit	67,592	71,275	**112,931**	238	12	127	384	128
6L-FT	32-bit	27,432	**29,259**	49,923	257	10	63	160	64

three interconnection networks. We chose to implement BTs and FTs of different levels in order to create structures of similar area. However, since the trees do not have the same depth, the number of PEs in each case differs. Therefore, in order to fairly compare their area cost we need to take into account the area of the PEs. Consequently, as Table 1 illustrates, the BT requires more area than the 6-level FT when using PEs larger than 29 logic cells, and more area than the 7-level FT when each PE occupies more than 357 logic cells. We evaluate the performance of the following sorting algorithm including the latency of loading and unloading data for these three interconnection networks for data sizes 2^{10} up to 2^{26}. Fat trees cannot exploit during the execution phase the fact that they have more links per node in the higher levels. That is because only one element can move from a child node to a parent node at a time. Therefore, the execution latency is identical in BTs and FTs *of the same tree depth*. However, the difference between the FTs and the BTs occurs in the load/unload phase, where the FTs are significantly better.

Fig. 13 illustrates the performance ratio between the 7 and 6-level FT and the BT. The 7-level FT is up to 1.7 times better than the 8-level BT, however for large data sets (2^{26}) is less efficient. For small data sizes the 6-level FT achieves higher performance than the BT up to 1.3×, while for larger data sets ($>2^{14}$) it is less efficient requiring up to 2× the latency of the 8-L BT. In general, as the data set gets larger the performance ratio decreases. That is because the initial sorting in the leaf nodes becomes the dominant factor compared to the load/unload communication delay.

One would assume that the FT topology is more suitable for sorting than the BT, since FT I/O bandwidth is substantially higher. However, this case study clearly shows that we cannot choose in all cases the most suitable topology according only to the application. There are other parameters that should be taken into account such as the data size, the underlying technology and the architecture of the PEs. This makes our argument stronger, meaning that reconfigurable interconnects can be proved beneficial even when its not clear in advance which topology is suitable. The above case study indicates that the overall performance depends on the data size and PEs area requirements. *Assuming a certain area constrain on a chip, the same program may demand a different interconnection network, depending on the amount of data it has to operate upon,*

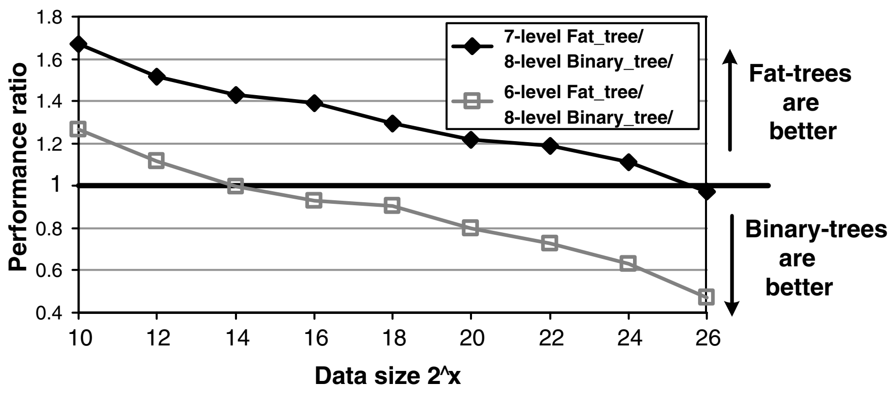

Fig. 13. Performance ratio between a 7-L or 6-L FT and a 8-L BT in sorting.

implying that the networks on demand will have been the correct choice.

4.3. 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 $n \times n$ 2D mesh, an FPGA or a BT interconnection network. The utilized application is the MaxBT1 algorithm described in the previous Section 4.2. 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 2D mesh, a BT, or an FPGA is the underlying physical network, Fig. 14 illustrates the programming function that decides the interconnection setup for the MaxBT1 algorithm.

The above function is based on the results of Fig. 10. 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 2D mesh and requires about 70% less resources. Therefore, based again on the performance results of Fig. 10, it is more efficient to utilize the BT topology. For the 2D 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

```
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. 14. A programming example for the MaxBT1 algorithm.

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 2D 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.

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 will not 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 (2D 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× 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 (e.g. via monitoring mechanisms). Hot spot connections of the network can also be added (see also Figs. 1 and 6).
- 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 logical networks. The system programmer finds the most suitable mapping/utilization of the network, exploiting the flexibility of the 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 co-exist 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 works [20,7].
- Contrary to others [20], 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.
- Reconfigurable hardware has a *unique* characteristic. Physical connections can match the logical connections of an application and support addi-

tional direct point-to point connections not foreseen by the algorithm developer.

- Our approach can *dynamically* adapt to *arbitrary* topologies, while other solutions can only support *several* topologies and regular predefined structures [20].
- In reconfigurable hardware, we set *raw* connections and the configuration time can be relatively small if a fine-grain configuration can be supported by the technology. Furthermore, there is no local memory under each switch, to store the possible configurations for *every* supported topology. Other solutions employ routing elements to "reprogramme" the local memory introducing delay [20]. In essence, such networks are *programmable* rather than *reconfigurable*, adding extra interconnection overhead and delays.

5. Conclusions

In this paper, we introduced 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 or reconfigurable 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, I. Sourdis, FLUX networks: interconnects on demand, in: International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS), Samos, Greece, July 2006, pp. 160–167.

[2] S. Vassiliadis, L.A. Sousa, G.N. Gaydadjiev, The Midlifekicker Microarchitecture Evaluation Metric, in: Proceedings of the IEEE International Conference ASAP05, July 2005, pp. 92–97.

[3] S. Vassiliadis, L.A. Sousa, G.N. Gaydadjiev, The Midlifekicker Microarchitecture Evaluation Metric, in: Proceedings of the IEEE International Conference ASAP05, July 2005, pp. 92–97.

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

[5] F.T. Leighton, Introduction to Parallel Algorithms and Architectures: Array, Trees, Hypercubes, Morgan Kaufmann Inc., CA, USA, 1992.

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

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

[8] S. Vassiliadis, I. Sourdis, Reconfigurable fabric interconnects, in: International Symposium on System-on-Chip (SoC), Tampere, Finland, November 2006, pp. 41–44.

[9] S. Vassiliadis, S. Wong, G.N. Gaydadjiev, K. Bertels, G. Kuzmanov, E.M. Panainte, The molen polymorphic processor, IEEE Transactions on Computers (November) (2004) 1363–1375.

[10] S. Vassiliadis, I. Sourdis, Reconfigurable FLUX Networks, in: IEEE International Conference on Field Programmable Technology (FPT), Bangkok, Thailand, December 2006, pp. 81–88.

[11] Xilinx, Virtex-II pro Platform FPGA Handbook.

[12] P. Sedcole, B. Blodget, J. Anderson, P. Lysaght, T. Becker, Modular partial reconfiguration in Virtex FPGAs, in: Proceedings of 15th International Conference on Field Programmable Logic and Applications, 2005.

[13] S.-K. Lee, H.-A. Choi, Embedding of complete binary trees into meshes with row-column routing, IEEE Trans. Parallel Distributed Systems 7 (5) (1996) 493–497.

[14] A. Grama, A. Gupta, G. Karypis, V. Kumar, Introduction to Parallel Computing, Pearson Education Limited, USA, 2003.

[15] C.E. Leiserson, Fat-Trees: universal networks for hardware-efficient supercomputing, IEEE Trans. Comput. 34 (10) (1985) 892–901.

[16] S.A. Browning, The Tree Machine: A Highly Concurrent Computing Environment, Ph.D. dissertation, CS Dept., CalTech, 1980.

[17] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, The MIT Press, Cambridge, MA, 1990.

[18] J.G. Delgado-Frias, S. Vassiliadis, C.-L. Chu, A. de Luca, DT: a binary tree parallel computer with distributed I/Os, Journal of the Mexican Society of Instrumentation 3 (4) (1994) 33–42.

[19] C.D. Thompson, H.T. Kung, Sorting on a mesh-connected parallel computer, Commun. ACM 20 (4) (1977) 263–271.

[20] L. Snyder, Introduction to the configurable, highly parallel computer, IEEE Computer 15 (1) (1982) 47–56.

Stamatis Vassiliadis (M'86-SM'92-F'97) was born in Manolates, Samos, Greece, in 1951. He is currently a Chair Professor in the Electrical Engineering, Mathematics, and Computer Science (EEMCS) department of Delft University of Technology (TU Delft), The Netherlands. He previously served in the Electrical and Computer Engineering faculties of Cornell University, Ithaca, NY and the State University of New York (S.U.N.Y.), Binghamton, NY. For a decade, he worked with IBM, where he was involved in a number of advanced research and development projects. He received numerous awards for his work, including 24 publication awards, 15 invention awards, and an outstanding innovation award for engineering/scientific hardware design. His 72 USA patents rank him as the top all time IBM inventor. Dr. Vassiliadis received an honorable mention Best Paper award at the ACM/IEEE MICRO25 in 1992 and Best Paper awards in the IEEE CAS (1998, 2001), IEEE ICCD (2001), PDCS (2002) and the best poster award in the IEEE NANO (2005). He is an IEEE and ACM fellow and a member of the Dutch Academy of Science.

Ioannis Sourdis was born in Corfu, Greece, in 1979. He received his Diploma degree in 2002 and his Masters Degree in 2004 in Electronic and Computer Engineering from Technical University of Crete, Greece. He is currently working towards the Ph.D. in Computer Engineering in the Delft University of Technology, The Netherlands. His research interests include the architecture and design of computer systems, multiprocesor parallel systems, interconnection networks, reconfigurable hardware, and networking systems.