A Note on Implementing Combining Networks[*]

Jörg Keller Thomas Walle

FB 14 Informatik, Universität des Saarlandes

Postfach 15 11 50, 66041 Saarbrücken, Germany

Phone: +49-681-302-$\{2576, 3036\}$

Fax: +49-681-302-4290

Email: {jkeller,twalle}@cs.uni-sb.de

May 30, 1995

Abstract

In shared-memory multiprocessors, combining networks serve to eliminate hot spots due to concurrent access to the same memory location. Examples are the NYU Ultracomputer, the IBM RP3 and the Fluent Machine. We present a problem that occurs when trying to implement the Fluent Machine's network nodes with network chips that do not know their position within the network. We formulate the problem mathematically and present two solutions. The first solution requires some additional hardware around nodes that can be put outside network chips. The second solution requires a minor modification of the routing algorithm but no additional hardware is needed.

Keywords: Computer architecture, combining networks, butterfly networks, shared memory

1 Introduction

In machines with emulated shared memory, combining networks serve two purposes. First, they route memory access requests and their answers between processors and memory modules. Second, they merge concurrent accesses of several processors to one memory cell into one request and thus reduce hot spots. This kind of access cannot be neglected because it will occur in

[*]This work was supported by the German Science Foundation (DFG) in SFB 124.

system parts like synchronization and resource management. Also concurrent access is often used in parallel algorithms for the PRAM model. Combining networks have been used in several architectures, e.g. the NYU Ultracomputer [3], the IBM RP3 [5], and the Fluent Machine [6].

The Fluent Machine differs from previous approaches. Its routing algorithm guarantees that requests for the same cell are merged into one request. However, it is not obvious how to implement the Fluent Machine's network nodes with *universal network chips*, i.e. chips that do not have encoded their position within the network. The use of universal network chips leads to scalability because only one type of node is necessary for several machine sizes.

We will formulate the problem mathematically and present two solutions. The first does not change the routing algorithm used in the Fluent Machine but requires additional hardware around network nodes. The second solution requires a minor change of the routing algorithm. We prove that the algorithm is still correct and the performance is not affected by this modification.

The remainder of the article is organized as follows. In Section 2 we review Ranade's routing algorithm for the Fluent Machine and give some re-engineering improvements. In Section 3 we work out the problem that occurs when implementing this algorithm in universal network chips. In Section 4 we present two solutions to that problem. Section 5 contains a discussion.

2 Ranade's Routing Algorithm

Ranade's routing algorithm uses six phases, i.e. six traversals of butterfly networks to route and combine requests from processors to memory modules and to re-duplicate and route answers back to processors. Routing only occurs in phases 2 and 5, the other phases can be implemented by dedicated hardware [1]. In Ranade's scheme, each butterfly node contains a processor and a memory module. This can be changed such that processors (together with dedicated hardware for phases 1 and 6) are only placed at the inputs of phase 2 and the outputs of phase 5. Memory modules with multiple banks (implementing phases 3 and 4) are only placed at the outputs of phase 2 and the inputs of phase 5. One physical processor simulates a number of Ranade's processors. We call the execution of one instruction of each simulated processor a *processor round*. For details of the processor architecture see [1, 4].

We will focus on phase 2 because combining happens here. Phase 2 is implemented on a butterfly network as given by Def. 1.

Definition 1 *A* butterfly network *with $N = 2^n$ inputs and outputs is a graph G_n that consists*

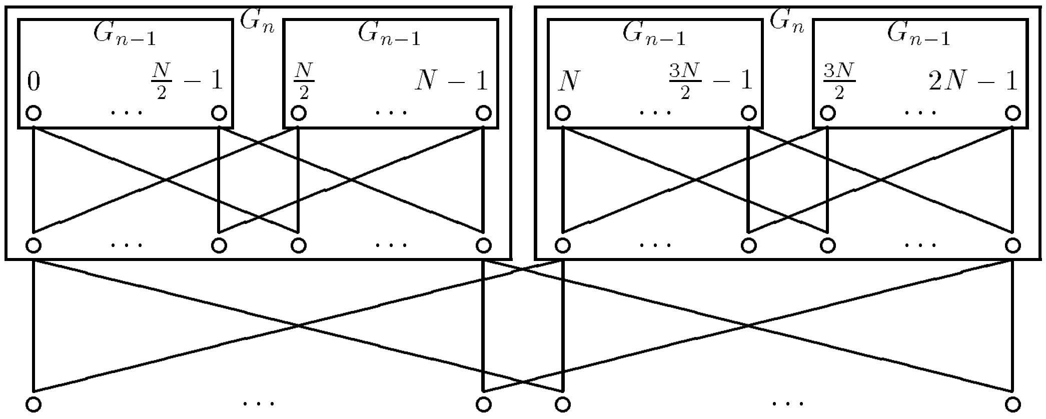

Figure 1: Construction of G_{n+1}

of $n+1$ stages, numbered from 0 to n, with N nodes per stage, numbered[1] *from 0 to $N-1$. G_0 consists of a single node, G_{n+1} can be constructed by taking two copies of G_n and $2N$ additional nodes that form the last stage of G_{n+1}. Nodes i, where $0 \leq i < N$, in stage n of the two smaller butterflies are connected to nodes i and $i+N$ in stage $n+1$. The construction is shown in Fig. 1. The left output of a network node is denoted by 0, the right one by 1.*

Requests by processors are put into packets, injected at level 0, and delivered to memory modules at level n. Packets consist of a *mode* (READ or WRITE), an *address* and, in the case of a WRITE, also of a *data word.* To obtain a unique packet length a dummy value is inserted for READ packets. The address encodes both the module number and the memory address within that module. Packets of one processor round are injected sorted by their addresses. At the end of the round a packet with the special mode *End of Round* (EOR) and address ∞ is injected.

Each network node selects from the two input buffers the packet with the smaller address and thus maintains the sorted order of packets, which can be easily proven by induction. If two packets with identical addresses and modes meet, one is selected. The other is deleted and in the case of a READ some information is stored to guarantee re-duplication of the answer packets on the way back. The sorting guarantees that all packets of one round with identical addresses meet and get combined.

The packet selected by a node is transmitted to the next level of the network via the appropriate output link of the node (for path selection see Section 3). Only EOR packets are transmitted via both outputs to ensure separation of rounds (address ∞ ensures that an EOR

[1]In the sequel we will use binary representations instead of the numbers itself.

is only selected if both input buffers contain EOR packets).

An empty input buffer prevents a node from sending a packet that waits at the other input buffer. If it would be sent, the sorting could be destroyed by a packet with smaller address arriving later at the empty input buffer. To avoid unnecessary waiting, GHOST packets are introduced. If a selected packet is transmitted via an output link a, where $a \in \{0,1\}$, then a GHOST packet carrying the same address is sent via output link $1-a$. GHOSTs serve as lower bounds of future packet addresses along this link. GHOSTs that must wait because they are not selected or blocked by full buffers are destroyed because a new GHOST or a packet will follow the next cycle, so no information is lost.

3 Implementation

The n most significant bits of a packet's address specify the destination module of this packet. The remaining bits specify the local address within that module. Path selection is given by the following Lemma 1.

Lemma 1 *A packet with destination module $j_{n-1} \ldots j_0$, that is injected at level 0 of a butterfly network G_n, must be transmitted in level i, where $0 \leq i < n$, along output j_{n-i-1}.*

Proof (by Induction on n): The case $n = 0$ is obvious. To prove the claim for a butterfly network G_n, where $n \geq 1$, we consider the recursive construction from networks G_{n-1} as given in Fig. 1. The packet will be routed to node $x = j_{n-1} \ldots j_1$ in level $n-1$ in one of the networks G_{n-1}. By the definition of G_n, it will reach node $j_{n-1} \ldots j_0$ in level n from both positions by taking output j_0. ■

Note that in our implementation the order of destination bits is not of particular importance. A modification of this order only leads to a permutation of memory modules which does not affect correctness.

In a direct implementation of the path selection scheme from Lemma 1 each network node must know its level number to select the routing bit j_{n-i-1}. To implement the algorithm with universal network chips, this must be avoided. One could try to place the desired routing bit always at the same position in every level. This is possible by the following Lemma 2.

Lemma 2 *If two packets meet in a network node in level i, where $0 \leq i \leq n$, then the i most significant bits of both addresses are identical. We will call these bits* address prefix.

Proof: Consider the subgraph of G_n that contains the two nodes where the packets were injected and the node where they meet. The subgraph is a butterfly network G_i. We apply

Lemma 1 with $n = i$, then the two packets are destined for the same output node of a butterfly network G_i and hence their i most significant address bits are identical. ∎

Lemma 2 seems to induce the following implementation: Because the prefixes of two meeting packets are identical, only the *remaining addresses*, which consist of address bits $n - i - 1$ to 0, are needed to compare addresses in level i. If the address is shifted left by one position after each level, then the desired routing bit is always bit $n - 1$ which leads to universal network chips.

However, this implementation leads to errors as the following Lemma 3 shows.

Lemma 3 *If a GHOST and a packet meet in node $j_{n-1} \ldots j_0$ in level i, then their address prefixes are different, i.e. comparison of the remaining addresses is not sufficient.*

Proof: For a packet, the prefix is the sequence of routing decisions so far. However, when a GHOST is generated, it is not transmitted via the output that the address would force (see end of Section 2). Hence, the GHOST's address prefix differs in that position from the sequence of routing decisions. If a GHOST and a packet meet, their sequences of routing decisions are identical, and hence their address prefixes must be different. In this case it can happen that the packet is selected before the GHOST, because the packet's remaining address is smaller, although the packet's address is larger than the GHOST's address. ∎

4 Two Solutions

4.1 Minor Hardware Modification

One can avoid the above error by providing complete addresses to comparator units (see Fig. 2). To achieve this, we must compensate the left shift applied in every level. Furthermore, the desired routing bit must still be in position $n - 1$. Both demands together can be fulfilled by inserting the 'Rightshift and Copying' circuit before the routing and address shifting unit in level i (denoted by a box above the dash line in Fig. 2). The address is shifted right by one position, then the desired routing bit, i.e. bit $n - i - 2$, is copied to position $n - 1$.

Now the desired routing bit always is in position $n - 1$ and after the regular left shift we have the complete unshifted address. We only have to ensure that we have one spare bit in the address part of the packets so that no address information is lost during the right shift. This should normally be possible as address parts typically have a fixed size (32 or 64 bit) and real address spaces are smaller.

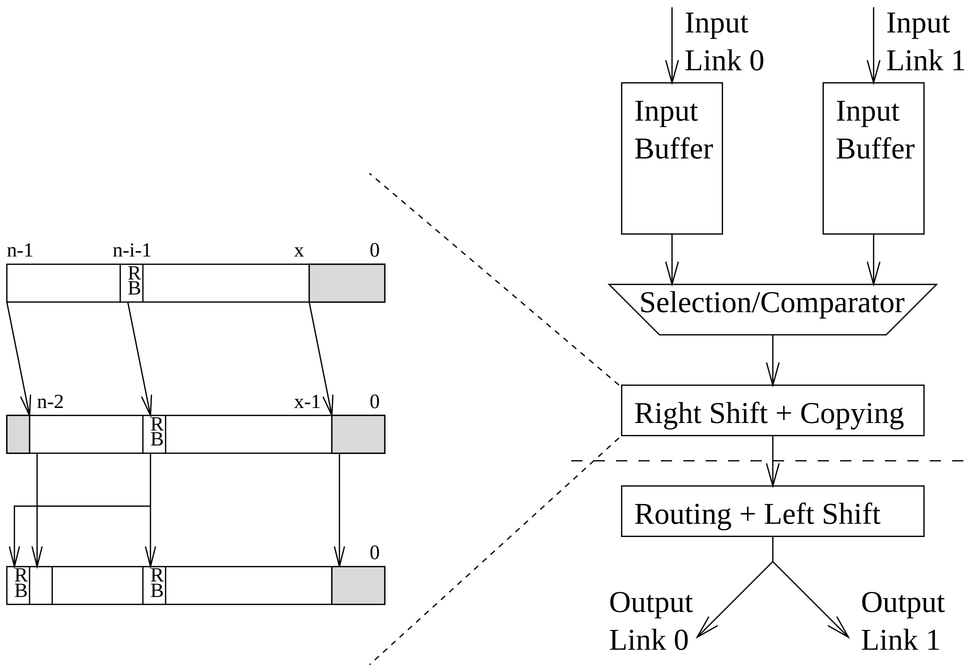

Figure 2: Schematic Design of a Network Node

The copying of address bit $n - i - 2$ is an implicit encoding of the level number i. Hence, we have to take care that this copying is done outside the network chips. Then we can use universal network chips, encoding of levels on boards can be done by jumpers.

To see how the copying unit can be placed outside a chip we consider the design of a network node as shown in Fig. 2. An obvious mapping of one network node to a chip would place the copying unit within the chip. However, if we consider the dashed line in Fig. 2, the number of wires crossing it is not more than the number of wires in an output link. Hence, we can use a mapping from [2] as shown in Fig. 3. The resulting chips do not use more pins than chips

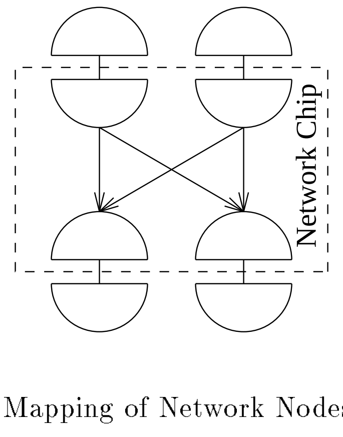

Figure 3: Mapping of Network Nodes to Chips

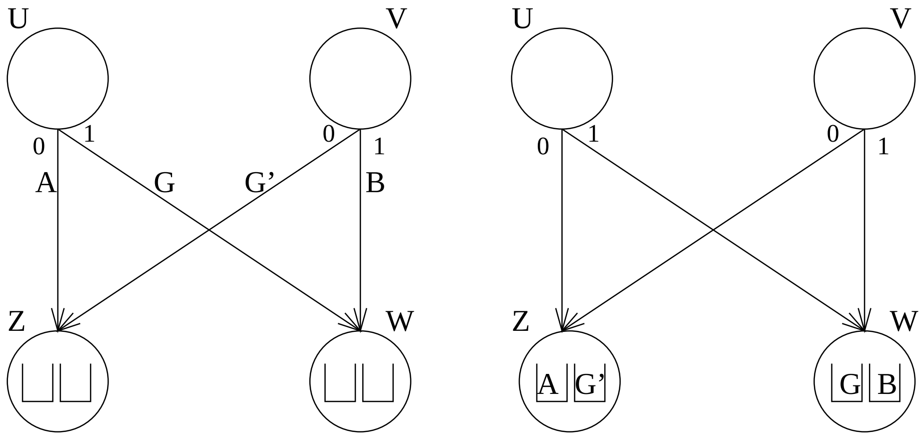

Figure 4: Generation of GHOSTs

that implement one network node, and the copying unit can be put between two chips. One can prove that the network of chips obtained by this mapping still is a butterfly network [2]. Note that this mapping doubles the gate utilization in network chips. As network chips are pinlimited, this does not impose a problem and even reduces the number of chips by a factor of two [2].

4.2 Minor Algorithm Modification

Consider the situation shown in Fig. 4. Node U sends a packet A to node Z along output link 0. The packet's address then must have the form $a0b$, where $a0$ is the prefix and b is the remaining address. The packet generates a GHOST G with address $a0b$ that is sent along output link 1 to node W. Some packet B that meets this GHOST in node W must have entered W along the other input and hence must have address $a1c$. It follows that GHOST G must be selected in node W. Since GHOSTs serve to avoid unnecessary waiting, GHOST G is of no use in node W, because the packet in W must wait, no matter whether the GHOST was there or whether the buffer was empty.

Now consider the situation for packet B which is routed along output link 1 from V to W. Packet B generates a GHOST G' with address $a1c$ that is transmitted along output link 0 to node Z where it meets packet A with address $a0b$. It follows that in node Z packet A must always be selected. Thus, if one sends GHOSTs only along output link 0, then the comparison between a packet and a GHOST is independent of the GHOST's address, the packet will always win.

It is obvious that the modified algorithm is correct as long as an empty input buffer prevents nodes from sending a packet that is waiting in the other input buffer. It is also easy to see that performance will not change as GHOSTs generated along an output link 1 have a smaller address than the packets that they meet, even if these GHOSTs are further forwarded. As the GHOST's address are not used anymore, Lemma 3 does not apply. Therefore, it is possible to employ the implementation from section 2, where addresses are shifted.

5 Discussion

We presented two solutions to an implementation problem of Ranade's routing algorithm. Both solutions do not affect the correctness of the routing algorithm, Ranade's proof of the algorithm's performance still works. Our simulations with random requests show better performance than Ranade's time bound.

The first solution has the minor disadvantage that it only allows usage of universal network chips. On the boards, the levels can be encoded by jumpers. Also if packets are transmitted between chips in several pieces called *flits*, the flits must be treated differently depending on whether they carry an address part or a data part of a packet. This requires additional hardware on boards and enlarges propagation delay between two network chips. The second solution allows to use universal network chips and boards and requires no additional hardware between network chips.

The second solution suffers from the fact that it can only be applied if routing bits are taken as given by Lemma 1. If routing bits are used in this order, then in each processor round each network node will first send packets along output link 0, then along output link 1. If routing bits are used in any other order, then routing decision and sorting are de-coupled. The sorting depends on the most significant bit whereas the routing decision does not. This better distribution leads to a better utilization of buffers. In simulations, improvements have been between 5 and 10 %. Note that the first solution allows any order of routing bits.

Thus one has a kind of trade-off between performance and universality of design.

Acknowledgements

The authors would like to thank Andreas Paul for bringing up the problem of disturbed sorted order in the network.

References

[1] F. Abolhassan, J. Keller and W. J. Paul, On the cost–effectiveness of PRAMs, in: *Proc. 3rd IEEE Symp. on Parallel and Distributed Processing* (1991) 2–9.

[2] D. Cross, R. Drefenstedt and J. Keller, Reduction of network cost and wiring in Ranade's butterfly routing, *Inform. Process. Lett.* **45** (1993) 63–67.

[3] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, The NYU ultracomputer — designing an MIMD shared memory parallel computer, *IEEE Trans. Comput.* **C–32** (1983) 175–189.

[4] J. Keller, W. J. Paul and D. Scheerer, Realization of PRAMs: Processor design, in: *Proc. WDAG '94, 8th Internat. Workshop on Distributed Algorithms* (1994) 17–27.

[5] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton and J. Weiss, The IBM research parallel processor prototype (RP3): Introduction and architecture, in: *Proc. 1985 Internat. Conf. on Parallel Processing* (1985) 764–771.

[6] A. G. Ranade, How to emulate shared memory, *J. Comput. System Sci.* **42** (1991) 307–326.