On the Physical Design of PRAMs

Ferri Abolhassan
Reinhard Drefenstedt
Jörg Keller
Wolfgang J. Paul
Dieter Scheerer

Computer Science Department
Universität des Saarlandes
6600 Saarbrücken
Germany

Abstract

We sketch the physical design of a prototype of a PRAM architecture based on RANADE's Fluent Machine. We describe a specially developed processor chip with several instruction streams and a fast butterfly connection network. For the realization of the network we consider alternatively optoelectronic and electric transmission. We also discuss some basic software issues.

1 Introduction

Today all parallel machines with large numbers of processors also have many memory modules as well as a network or a bus between the processors and the memory modules. The machines however come with two radically different programming models.

The user of multicomputers is given the impression, that he is programming an ensemble of computers which exchange messages via the network. The user has to partition the data, and exchange of data between computers is done by explicit message passing. A very crude model of the run time of programs on such machines is: as long as no messages are passed, things are obviously no worse than on serial machines. As soon as messages are passed, things can become bad, because of the network.

The user of shared memory machines is given the impression, that he is programming an ensemble of CPUs which simultaneously access a common memory. This is much more comfortable for the user but there is a catch. Because the underlying machine has several memory modules (and/or several large caches) there is of course message passing going on (e.g. by transporting cache lines). Again this message passing can cause serious deterioration of performance, but because the message passing is hidden from the user it is very difficult for the user to figure out, under which circumstances this effect can be avoided.

In spite of this drawback the ease of programing provided by the shared memory model is considered such an advantage, that one tries to provide this view even for machines, which were originally designed as multicomputers.

The best of both worlds would obviously be provided by a shared memory machine whose performance is highly independent of the access pattern into the shared memory. In the theoretical literature such machines are called PRAMs [9]. An impressive number of ingenuous algorithms for these machines has been developed by theoreticians, and simulations of PRAMs by multicomputers were extensively studied. Among these simulations [16] was generally considered the most realistic one.

In [13] a measure of cost–effectiveness of architectures was established, where hardware cost is measured in gate equivalents and time in gate delays. In [1, 2] the simulation from [16, 17] was developed into an architecture which according to this measure is surprisingly cost–effective even if compared with multicomputers under a numerical workload.

This paper describes a possible physical realization of a 128 processor prototype of the machine described in [1, 2]. Roughly speaking the paper deals with those aspects of the hardware, which are not captured by the model from [13]: pins, boards, connectors, cables etc. We also treat two basic software issues: synchronization and memory allocation.

2 The Fluent Machine

The Fluent Abstract Machine [17] simulates a CRCW priority PRAM with $n \log n$ processors. The processors are interconnected by a butterfly network with n input

nodes. Each network node contains a processor, a memory module of the shared memory and the routing switch. If a processor $\langle col, row\rangle$ wants to access a variable stored at address x it generates a packet of the form (destination,type,data) where destination is the tuple $(node(x), local(x))$ and type is READ or WRITE . This packet is injected into the network, sent to node $node(x) = \langle row', col'\rangle$ and sent back (if its type is READ) with the following deterministic packet routing algorithm.

1. The packet is sent to node $\langle \log n, row\rangle$. On the way to column $\log n$ all packets injected into a row are sorted by their destinations. The reason for the sorting is the fact that two packets with the same destination have to be combined.
2. The message is routed along the unique path from $\langle \log n, row\rangle$ to $\langle 0, row'\rangle$. The routing algorithm used is given in [16].
3. The packet is directed to node $\langle col', row'\rangle$ where memory access is handled.
4. – 6. The packet is sent the same way back to $\langle col, row\rangle$.

RANADE proposes to realize the six phases with two butterfly networks where column i of the first network corresponds to column $\log n - i$ of the second one. Phases 1,3,5 use the first network, phases 2,4,6 use the second network. Thus the Fluent Machine consists of $n \log n$ nodes each containing one processor, one memory module and 2 butterfly networks.

3 Improved Machine

In RANADE's algorithm the next round can only be started when the actual round is completely finished, i.e. when all packets have returned to their processor. This means that overlapping of several rounds (*pipelining*) is not possible in the Fluent Machine. This disadvantage could be eliminated by using 6 physical butterfly networks. Furthermore the networks for phases 1 and phase 6 can be realized by n sorting arrays of length $\log n$ as described in [2]. The networks for phases 3 and 4 can be realized by driver trees and OR trees, respectively. Both solutions have smaller costs than butterfly networks and have the same depth.

The processors spend most of the time waiting for returning packets. This cannot be avoided. But we can reduce the cost of the idle hardware by replacing the $\log n$ processors of a row by only one physical processor (pP) which simulates the original $\log n$ processors as virtual processors (vP). Another advantage of this concept is that we can increase the total number of PRAM processors by simulating $X = c \log n$ (with $c > 1$) vP's in a single pP. VALIANT discusses this as *parallel slackness* in [19]. The simulation of the virtual processors by the physical processor is done by the principle of *pipelining*. A closely related concept is *Bulk Synchronous Parallelism* in [19].

In vector processors the execution of several instructions is overlapped by sharing the ALU. If a single instruction needs x cycles, pipelined execution of t instructions needs $t + x - 1$ cycles. Without pipelining they need tx cycles.

Instead of accelerating several instructions of a vector processor with a pipeline, we use pipelining for overlapped execution of one instruction for all X vP's that are simulated in one physical processor. To simulate X vP's we increase the depth of our ALU artificially. The virtual processors are represented in the physical processor simply by their own register sets. We save the costs of $X - 1$ ALU's.

The depth δ of this pipeline serves to hide network latency. This latency is proved to be $c \log n$ for some c with high probability [16]. If $\delta = c \log n$ then normally no vP has to wait for a returned packet. This c increases the number of vP's and the network congestion. But network latency only grows slowly with increasing c. Thus there exists an optimal c.

When the last of all vP's has injected its packet into the network, there are on the one hand still packets of this round in the network, on the other hand the processors have to proceed (and thus must start executing the next instruction) to return these packets. CHANG and SIMON prove in [7] that this works and that the latency still is $O(\log n)$. The remaining problem how to separate these different "rounds" can easily be solved. After the last vP has injected its packet into the network, an *End of Round Packet (EOR)* with a destination larger than memory size m is inserted. Because the packets leave each node sorted by destinations, it has to wait in a network switch until another EOR enters this switch along its other input. It can be proved easily that this is sufficient.

One problem to be solved is that virtual processors executing a LOAD instruction have to wait until the network returns the answer to their READ packets. Simulations indicate, that for $c = 6$ this works most of the time (see [2]). But this is quite large in comparison to $\log n$. We partially overcome this by using delayed LOAD instructions as in [15]. We require an answer to a READ packet being available not in the next instruction but in the next but one. Investigations show that insertion of additional 'dummy' instructions happens very rarely [15]. But if a program needs any dummy instructions, they can easily be inserted by the compiler. This reduces c to 3 without significantly slowing down the machine.

Our machine will consist of 128 physical processors (pP) with 32 virtual processors (vP) each. The vP's correspond to the different pipeline steps of a pP.

4 The Processor Chip

The instruction set of our processor is based on the Berkeley Risc processor [15]. The basic machine commands are quite similar to this processor except the special commands for handling the several instruction streams. Instead of register windows we have the register sets of the virtual processors. The processor has a LOAD–STORE architecture, i.e. COMPUTE instructions (adding, multiplying, shifts, logarithmical and bit oriented operations) work only on registers and immediate constants. Memory access only happens on LOAD and STORE instructions. All instructions need the same amount of time (one cycle). We do not support floating point arithmetic but the

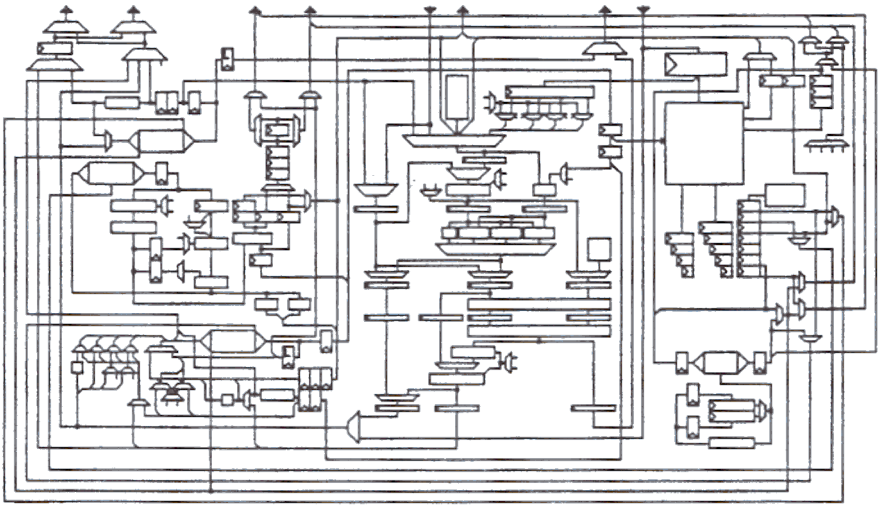

Figure 1: Data paths of the processor chip

addition of a commercial coprocessor is possible.

Because of the LOAD-STORE architecture one multiplier can be used for multiplications in COMPUTE instructions and for hashing global addresses with a linear hash function in LOAD and STORE instructions. This means that hashing does not require much special hardware.

The processor will be located in a 299 PGA and will consist of about 50,000 gate equivalents. Figure 1 shows the data paths of the processor.

Each virtual processor is represented by its own register set consisting of 32 registers $R_0 - R_{31}$ each 32-bit wide. R_1 of each register set is the program counter, R_2 the local stack pointer and R_3 the global stack pointer. The register sets are held in a static RAM outside the chip. The vP's are handled in pipeline in a round robbin manner. Each cycle of a vP corresponds to a step in the pipeline. The cycle time of the pipeline will be $120ns$ in $2\mu m$ CMOS technology. One step of all 32 vP's takes $32 \cdot 120ns = 3840ns$ time. Additionally each vP can support up to 32 contexts which we will also call logical processors (lP) later on. Therefore the programmer can handle $32 \cdot 32 \cdot 128 = 131,072$ contexts without any software overhead.

5 Several Processes

Each vP is able to simulate 32 lP's without any software overhead. In the following we describe the hardware support of this simulation. In this section we call the work of a logical processor a process. The vP's needs machine commands to "create", "switch" and "terminate" processes. A process (lP) is represented by the values of

its register set including the program counter, stack pointers and status register. We call these values the "context" of a process. If a vP switches from one process to another it has to switch the context, i.e. the current lP has to save the value of its register set somewhere and has to load the value of the register set of the next lP from somewhere. This is a complex operation and a fast mechanism to realize that is needed. However the execution time of the commands to switch, terminate and create contextes should be as fast as the other machine instructions. Because it is impossible to hold $32 \cdot 32$ register sets on chip, the register sets are located in a $32K \times 32$ static RAM outside of the processor. Access time to the large static RAM is not the critical part of computation and therefore does not slow down processor speed. To switch from one lP to another one has only to compute the base address of the new register set.

An arbitrary number of processes has to be emulated in software. This could be done e.g. by using a FIFO queue of process descriptions that is located in global memory. Parallel management of that queue needs constructs similar to parallel storage management as given in section 8.1.

A new process can only be created by a process on the same vP. A process can only terminate itself. A switch of processes can only activate the next inactive process (lP). The control of the different lP's is handled for each vP by a 32 bit wide mask b (the reason for the upper bound of lP's per vP). The 32 masks are held on chip. The value of b_i indicates, whether the i–th register set contains a process ($b_i = 1$) or not ($b_i = 0$). At the beginning $b = (0, \ldots, 0, 1)$, i.e. only the first process (lP_0) of every vP is active.

If a process lP_i wants to create a new process one looks for the smallest j with $b_j = 0$ and $i < j < 32$, if this exists. If that does not exist, one looks for the smallest j with $0 \leq j < i$. One changes the bit ($b_j = 1$) and sets the program counter of the j–th register set. The status register has an additional bit which indicates whether further process can be created ($b = (1, \ldots, 1)$). If there is no free register set nothing can be done.

If lP_i switches the process, one is looking for the smallest j with $i < j < 32$ and $b_j = 1$. If that does not exist, one looks for the smallest j with $0 \leq j \leq i$. This exists (e.g. $j = i$). The "actual" process is now lP_j. If a process lP_j is terminated, one sets the corresponding bit b_j to 0 and switches the process. The last process of a vP can not be terminated. The status register contains a flag, that is set if and only if b contains exactly one 1, i.e. if only one process is active.

The additional commands for the support of the different processes are the following: `CREATE` R_x, R_y, R_z creates a new process (if possible). The program counter of the new process is loaded with the value R_x of the current process, register R_y of the new process is loaded with the value of R_z of the current process, `SWITCH` switches a process, `KILL` terminates a process.

6 Network Design

As already mentioned, the prototype uses a butterfly network for processor–memory communication. It consists of 8 stages with 128 network nodes per stage. Packets from processors to memory modules consist of a 32 bit address, 32 bit data and 6 control bits specifying modus and operation. Packets on the way back consist of 32 bit data and 1 control bit. In each direction of a link there exists a bit specifying whether the input buffer of the node at the end of the link is already filled up or not. One link between two network nodes has to be $32 + 32 + 6 + 32 + 1 + 2 = 105$ bits wide (71 forward, 34 backward).

We have to decide how to partition network nodes on VLSI chips, how to partition these chips on printed cicuit boards (PCB's) and how to arrange the boards in racks. Clearly these decisions are not independent of each other. A chip is restricted by maximum numbers of gates and pins available. A PCB is restricted by its area and by the number of connections that can leave it. An arrangement of boards is restricted by the form of the available racks. The wires should not be too long because length of a wire restricts transmission speed and increases delay. The wiring should allow removal of boards.

6.1 Mapping Network Nodes to Chips

A network node that realizes RANADE'S routing algorithm and is able to perform multiprefix operations [17] needs the data paths shown in figure 2. It needs about 15,000 gate equivalents and a total of 420 pins plus power supply. The largest commercially available ASIC VLSI chips have about 70,000 gates and 300 pins (HDC105) or 48,000 gates and 240 pins (HDC064) [12]. This means that we have enough gates to implement several network nodes on one chip but not enough pins to realize the links for only one network node. Distributing a network node on several chips does not solve the problem because all parts of the node are connected by wide busses which lead to a lot of additional pins.

If we half the width of the links and send packets in two parts, we loose a factor of 2 in speed of the network but can implement one network node on one chip HDC064 — but we waste two third of the chip area. Further reduction of the links' widths is not useful because it would slow down the network too much. Thus the links have width $w = 53$ bits, $w_1 = 36$ in forward and $w_2 = 17$ in backward direction.

Fortunately RANADE'S routing algorithm allows to increase the gate/pin ratio by a factor 2 without increasing the number of links. One network node can be cut in two halves such that only $w+2$ bits cross the cut if w denotes the width of a link. The cut can be seen in figure 2. We implement in a chip a 2×2 butterfly but take only the last part of the nodes in one stage and take only the first part of the nodes in the following stage. Figure 3(a) shows the partitioning and 3(b) shows the implementation with 4 chips. The resulting butterfly network contains 7 stages with 64 chips per stage. One chip now contains 4 half network nodes or 2 nodes and 4 links.

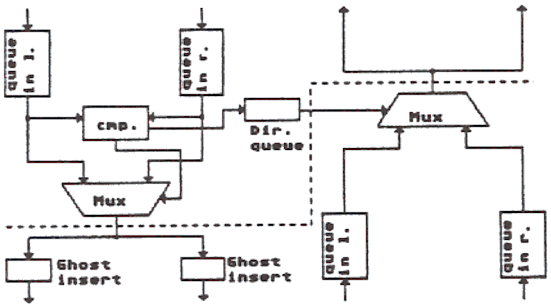

Figure 2: Cut of network nodes

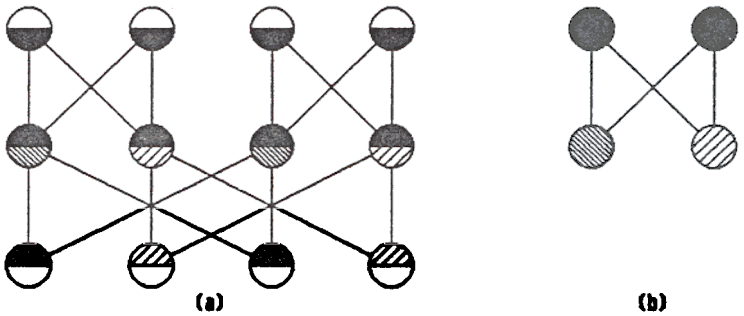

Figure 3: (a) Partitioning of the butterfly nodes. (b) Implementation of the nodes on chips

part k	stage i	board
1	0,1	$\left(1, \left\lfloor \frac{x}{4} \right\rfloor\right)$
2	2,3,4	$\left(2, 4 \left\lfloor \frac{x}{16} \right\rfloor + x \bmod 4\right)$
3	5,6	$(3, x \bmod 16)$

Table 1: Board for node $\langle i, x \rangle$

The first half of network nodes in the first stage and the second half of network nodes in the last stage can be deleted because in RANADE'S algorithm they only have one input (output).

We will denote chip $x \in \{0, \ldots, 63\}$ of stage $i \in \{0, \ldots, 6\}$ with $\langle i, x \rangle$. For $i < 6$ chip $\langle i, x \rangle$ is connected to chips $\langle i+1, x \rangle$ and $\langle i+1, x \oplus 2^i \rangle$ where $a \oplus b$ here denotes the number which has a binary representation that is obtained by the bitwise exclusive or of the binary representations of a and b. Because we will only talk of the network of chips we will call the chips also nodes.

6.2 Mapping Chips to Boards

Available Printed Circuit Boards with standard size have an area of $366mm \times 340mm = 124,440mm^2$ [5]. An HDC064 chip has an area of $2237.3mm^2$ [12]. If we consider that wiring on the board and connectors also consume a large amount of the board's area, the chips can only cover about 30% of the board, resulting in at most 16 chips per board. In order to reduce the number of links between boards, one board should contain a butterfly of appropriate size. In this case this is a butterfly with 3 stages and 4 chips per stage. The board then has 12 chips and 16 connectors.

Because of the 7 stages we have to install at least 3 network parts. We choose to design two kinds of boards. The first kind looks like sketched above, for the second we delete the third stage and obtain a board with two 2×2 butterflies. If we cut the network after the second and after the fifth stage we obtain a number of small butterflies that exactly fit on the boards designed above. The first and the third part are made of boards of the second kind, the second part is made of boards of the first kind. Each part consists of 16 boards. Board $j \in \{0, \ldots, 15\}$ of part $k \in \{1, 2, 3\}$ is called (k, j) .

The following tables shows how the nodes $\langle i, x \rangle$ are distributed on the boards. Table 1 gives for each node the board on which it is mapped. Table 2 gives for each board the nodes that it contains.

The 256 links between boards are the most critical ones because they traverse the longest distances. We have to take care of them when arranging the boards.

part k	stage i	nodes
1	0,1	$\langle i, 4j\rangle$,..., $\langle i, 4j+3\rangle$
2	2,3,4	$\langle i, \tilde{j}\rangle$, $\langle i, \tilde{j}+4\rangle$, $\langle i, \tilde{j}+8\rangle$, $\langle i, \tilde{j}+12\rangle$ $\tilde{j} = 16\left\lfloor \frac{j}{16} \right\rfloor + j \bmod 4$
3	5,6	$\langle i, j\rangle$, $\langle i, j+16\rangle$, $\langle i, j+32\rangle$, $\langle i, j+48\rangle$

Table 2: Nodes for board (k, j)

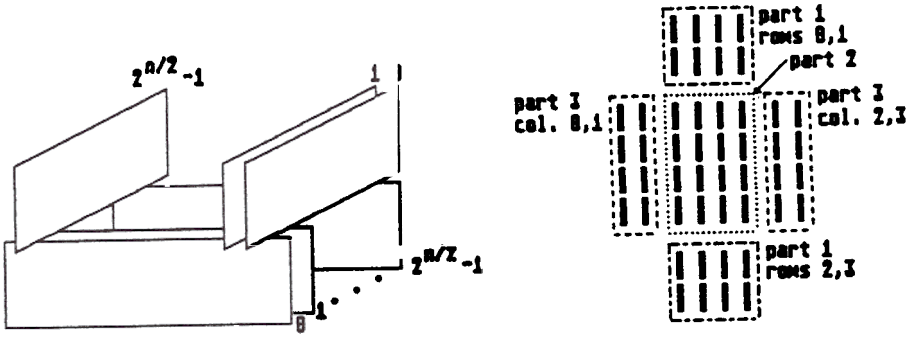

Figure 4: (a) Wise's arrangement of boards. (b) New arrangement of the boards

6.3 Arrangement of Network Boards

Arrangements of butterfly networks normally assume in contrast to reality that the implementation has a homogenous area of sufficient size, e.g. a VLSI plane or a large PCB [4]. WISE proposed in [20] a 3 dimensional arrangement of boards to implement a butterfly. This is the only paper known to us which addresses the problem. Assume that we have a butterfly with n stages and 2^{n-1} nodes per stage. Assume further that n is even. WISE makes a cut after $n/2$ stages and obtains boards that contain butterflies with $n/2$ stages and $2^{(n/2)-1}$ nodes per stage. Each of the two parts contains $2^{n/2}$ of these boards. One can prove that each board of the first part is only connected to all boards of the second part. WISE suggests the following arrangement: all boards stand vertical, the inputs of a board are on its top, the outputs on its bottom. The first part stands on top of the second part. The arrangement looks like given in figure 4(a).

This arrangement has the advantage that the parts ideally can be connected directly without any cables. The longest wire is on one of the boards, that means it is relatively short. The arrangement unfortunately has several disadvantages.

- Because each board can only hold a 3 stage butterfly (see subsection 6.2), the

arrangement is only suitable for up to 6 stage butterflies, i.e. butterflies with 64 inputs.

- A direct connection of the boards with standard connectors requires using rectangular connectors which have a length of $8cm$ [5] for a 64 bit connection. Thus a board connected with 8 other boards would have a minimum length of $64cm$. Furthermore removal of single boards would require a large physical force due to the number of connectors.

- If the boards are directly connected they do not fit in standard racks, because in racks only connections in the front and back of boards are usually allowed. If one puts the boards in two racks one on top of the other and preserving the order of the boards as given in figure 4(a) one has to use cables to connect them. This offers the possibility to place all connectors in a way that the boards can have reasonable size but the cables have to be longer than one board (minimum length about $60cm$). Otherwise the boards cannot be removed anymore. These racks still do not fit in standard cabinets because there the front of all racks has to be on one side of the cabinet.

  If we turn the upper rack to use standard cabinets we have an arrangement similar to that in the DATIS–P machine [18]. But there the cables between network boards have length $150cm$. This is not too long for the DATIS–P machine which works at 16 MHz, but it might be too long for a frequency of 25 MHz needed here.

We will use a different arrangement based on an observation how the boards are connected if we cut the network in three parts as described in 6.2.

Theorem 1 *If the boards of each of the three parts are numbered with*

$$\varphi : \{0, \quad ,15\} \to \{0, \quad ,3\} \times \{0, . \quad 3\}, \varphi(x) = \left(\left\lfloor \frac{x}{4} \right\rfloor, x \bmod 4\right)$$

then boards $(1,(i,0)),\ldots,(1,(i,3))$ *of the first part are only connected to boards* $(2,(i,0)),\ldots,(2,(i,3))$ *of the second part for* $0 \le i \le 3$ *and boards* $(2,(0,i)),\ldots,(2,(3,i))$ *of the second part are only connected to boards* $(3,(0,i)),\ldots,(3,(3,i))$ *of the third part for* $0 \le i \le 3$.

Proof: Board $(1,j)$, $j \in \{0,\ldots,15\}$ contains nodes $\langle 1,4j\rangle$ to $\langle 1,4j+3\rangle$ (see table 2). These nodes are connected to nodes $\langle 2,4j+l\rangle$, $l \in \{0,\ldots,3\}$ because node $\langle 1,x\rangle$ is connected to nodes $\langle 2,x\rangle$ and $\langle 2,x \oplus 2\rangle$ for all $x \in \{0,\ldots,63\}$. Node $\langle 2,4j+l\rangle$, $l \in \{0,\ldots,3\}$ belongs to board $(2,4\lfloor j/4\rfloor + l)$ (see table 1). Thus the first part of the claim holds.

Board $(3,j)$, $j \in \{0,\ldots,15\}$ contains nodes $\langle 5,j\rangle$, $\langle 5,j+16\rangle$, $\langle 5,j+32\rangle$, $\langle 5,j+48\rangle$ (see table 2). These nodes are connected to nodes $\langle 4,j+16l\rangle$, $l \in \{0,\ldots,3\}$ because node $\langle 4,x\rangle$ is connected to nodes $\langle 5,x\rangle$ and $\langle 5,x \oplus 16\rangle$ for all $x \in \{0,\ldots,63\}$. Node

$(4, j + 16l)$ belongs to board $(2, 4l + j \bmod 4)$ (see table 1). Thus board $(3, j)$ is connected to boards $(2, 4l + j \bmod 4)$, $l = 0, \ldots, 3$ and the second part of the claim holds. ∎

The theorem indicates the following arrangement: the boards of the each part are arranged in a 4×4 square, the square of part 1 on top of the square of part 2, and the square of part 3 on the right of the square of part 2. Then all connections are horizontal or vertical. In order to have the arrangement symmetric, the first and the third square are split in two rectangles: the boards of the first part are arranged in two rectangles on the top and on the bottom of the second square. The upper rectangle holds rows 0,1 the lower holds rows 2,3. The boards of the third part are arranged in two rectangles on the right and left of the second square. The left rectangle holds columns 0,1 the right holds columns 2,3. The arrangement is shown in figure 4(b). It has several advantages.

- The boards can be put in standard racks and cabinets.
- All wiring between boards is horizontal or vertical.
- The arrangement can even be extended for butterflies with 9 stages when for all parts the boards with 3 stage butterflies are used.

A complete geometric design has not yet been worked out. If electrical wiring is too long, one can consider using optoelectronic transmission.

7 Optoelectronic Transmission of Signals

To realize the network in the prototype it is necessary to transmit data across long distances. Therefore we check whether optoelectronic transmission should be used.

7.1 Components for Optical Point–to–Point Connections

Data transmission by fiber optic operates sequentially. This is in contrast to the demand for parallel transmission between the network nodes. To improve the total throughput one can use several channels of the same kind. Figure 5 shows a schematic outline how to build up the point–to–point connection by optical components. In each cycle of the network clock w_1 bit data are injected into the network. They pass the parallel/serial converter on board 1, the optical transmitter, the fiber, the optical receiver and the serial/parallel converter on board 2.

The necessary transmission speed can be computed as follows: Like mentioned in section 6 the number of multiplexed electric lines w involves $w_1 = 36$ (forward) and $w_2 = 17$ (backward) for each link (see section 6). Let t be the period of the network clock and p_f and p_b the number of parallel channels in forward and backward direction. Thus the necessary transfer rate d_r in the optical medium for the way

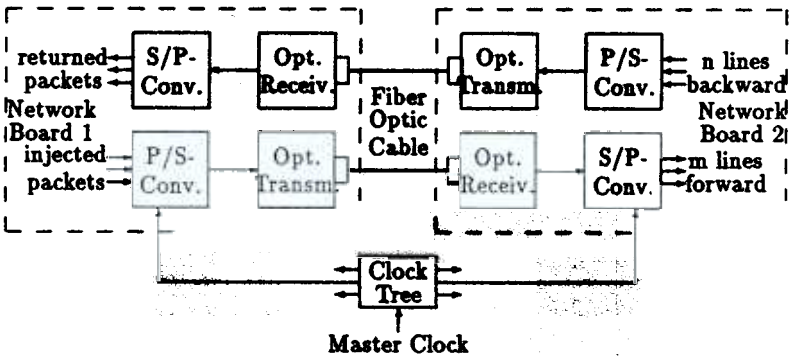

Figure 5: Components for optical data transmission

forward is $d_r = 2 \cdot m/(t \cdot p_f)$. For the way back is $d_r = 2 \cdot n/(t \cdot p_b)$. For given transfer rates we can use this equation to obtain the number of necessary channels.

To realize optical channels with these transfer rates we have studied two possible solutions:

First: Separate optical transmitters and receivers for *data communication* are available up to 1.2 Gbit/s [6]. For a reasonable price we can get compact and small modules (e.g. $15 \times 15 \times 60mm^3$) with a transmission rate of $d_r = 266$ Mbit/s, but an external P/S–converter made from ECL–chips is needed. If $t = 50ns$ (the clock cycle time of the network switches) then the above equations yields $p_f = 5$ and $p_b = 3$. Thus we need a total number of 8 sets of fiber optic cables, transmitters and receivers for each link.

Second: The optical unit and the P/S–converter chip are mounted together in a metal cover. The Transparent Asynchronous Xmitter–receiver Interface–chip set (TAXI) [6] provides a high performance transparent fiber optic 8 bit interface. Data transfer rates are up to 125 Mbit/s and the transfer is performed with error detection. Because of the integrated P/S-converter there is a lower bound of $t = 80ns$ for the clock period. This increases the number of bits to be transferred in parallel by a factor of 1.6 because of the $50ns$ clock of the network switches. In this case we obtain $p_f = 9$ and $p_b = 5$. Thus 14 pairs of the TAXI–chip set and logic are necessary to realize one link. The logic includes an interface between network link and the TAXI chip set, the details are not yet worked out.

7.2 Network design based on optical links

We assume that the components of one link occupies a reasonable area of PCB (e.g. $22 \times 7cm^2$), the transceiver board. But the costs of one pair of transceiver boards representing one link are still high (up to 9500,– DM). Therefore we use the advantages of fiber optic only to substitute the set of longest wires. This however

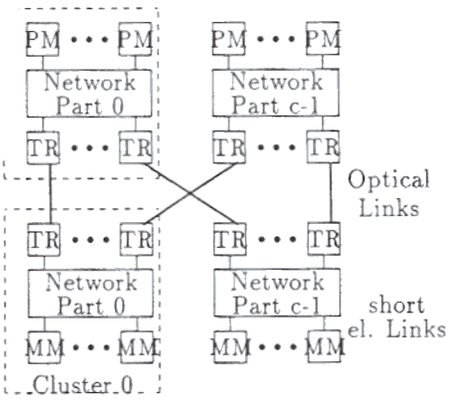

Figure 6: Optical links connect clusters

changes the network.

Fig. 6 shows the situation if we make a cut through the network horizontally. It is divided into $2 \cdot c$ clusters of boards. A cluster is a number of functionally associated boards. In our case it includes one network part of depth $s \in \{3,4\}$, $2^s \in \{8,16\}$ processor or memory boards and the same number of transceiver boards. Because the distances within the clusters are short we avoid long electrical links. But then we have one set of long optical links to connect the clusters with each other.

In the following table we compare cost and time for transmitting data by electrical lines and by fiber optic (using the TAXI chip set). Electrical lines may be single ended lines or twisted pair cables. Two commonly used types of driver/receiver combinations are listed. The length of electrical links may be increased using high speed trapezoidial bus drivers [3].

Costs for electric lines are computed as follows: $4 \cdot 128$ electrical interfaces are necessary to connect the three parts network boards with the PM's and MM's (refer to the whole design in section 6).

Because of the more compact layout shown in figure 6 additional electrical links may be avoided by using fiber optic. In this case the costs of 128 pairs of transceiver boards are listed. The technical data are taken from several data sheets. In general each link between two boards adds one or two delay units of network clock. If it is possible to use special low voltage swing drivers the power dissipation can be decreased [11] — however they are not yet commercially available.

	single line	twisted pair	fiber optic
driver/receiver type	AS1034B/F14	26LS31/32	DL6000
max. prop. delay of IC's	12.5 ns	47 ns	80 ns
power dissipation per link	15 W	19 W	43 W
number of links	$4 \cdot 128$	$4 \cdot 128$	128
total power dissipation [kW]	7.7 kW	9.7 kW	5.5 kW
IC's per link	2×10	2×11	2×14
area $[cm^2]$	50	70	154
relative costs	1	2.9	25

The table reveals that there is a large trade–off between costs on the one hand and power dissipation and wiring overhead on the other hand. In the near future we will test some types of electrical links and optical channels in order to decide which type of link provides the necessary throughput considering the real distances between the boards.

8 Basic Software Issues

A new architecture does not only has to have new efficient and powerful hardware, it also has to support hardware by suitable software, especially by an operating system with efficient resource management and by a compiler for a high–level language. A high–level language called FORK has been proposed [10] that is suited for a PRAM. The work on a compiler already has started. The operating system still has to be developed.

As examples of the problems that have to be solved we will present solutions for parallel storage management and for synchronization of multiple instruction streams. Both synchronization and memory management take advantage of the multiprefix (MP) [17] and SYNC (MP without return values) commands that are supported by hardware [2].

8.1 Parallel Storage Management

In a parallel machine that presents its user a shared global memory handling storage management is much more complicated than in a distributed machine where each processor has its private local memory on which it acts (and allocates memory) just like a sequential computer. In a shared memory machine several processors could try to allocate storage at the same time.

First we consider a simple solution for parallel storage allocation without worrying about freeing memory. Let $s(i)$ be the content of memory cell i in global memory and let cell 0 contain a pointer to the first cell of free global memory. If several processors $P_i, i \in I$ want to allocate memory of sizes $m(i)$ they execute a multiprefix command MP $0, +, m(i)$. As a result each processor P_i receives $s(0) + \sum_{j \in I, j<i} m(j)$ and the content of cell 0 is $s(0) = \sum_{j \in I} m(j)$. Each processor thus receives a pointer to its

requested memory block and cell 0 contains a pointer to the new begin of free memory. Correct freeing is only possible if the program guarantuees that the freeing operations are performed in reverse order to the allocating operations.

If we choose a fixed block size and allow processors only to allocate a number of not necessarily subsequent blocks, the problem becomes a little bit easier. Let cells 1 to max contain pointers to free memory blocks and let cell 0 contain a pointer to max. If processors $P_i, i \in I$ want to allocate $m(i)$ blocks they execute MP $0, -, m(i)$. Each processor P_i receives a pointer to a cell $x = s(0) - \sum_{j \in I, j<i} m(j)$. Cells $x, \ldots, x - m(i) + 1$ contain pointers to the $m(i)$ memory blocks for P_i. The pointers have to be copied. If processors want to free memory blocks they execute MP $0, +, m(i)$. Each processor receives a pointer to a cell x and writes to cells $x, \ldots, x + m(i) - 1$ the pointers to the $m(i)$ memory blocks it wants to free.

This stack mechanism only works correctly if we prevent the machine from freeing while others are allocating and vice versa. This can be done by programming a semaphore. One can get rid of this by using a FIFO queue. Allocate and free operations now take time proportional to the number of blocks. However allocation of memory with subsequent addresses larger than one block is not possible.

8.2 Synchronization Primitives

FORK provides synchronous execution of high level language commands. The runtime of this code however is often not predictable at compile time. In that case synchronization is necessary. A simple synchronization could be realized in hardware. But several synchronizations can happen simultaneously. Thus one has to provide several synchronizations within "groups" of processors.

Suppose each processor of a group that has to be synchronized later on knows the address a of a cell in global memory. First all processors store 0 in that cell. Then each processor executes SYNC $a, +, 1$. The cell then contains the number of processors in the group. The processors now execute the code for the high level language command after which they have to be synchronized. Each processor that has finished execution of that code executes MP $a, -, 1$. After that it reads the content of a until this content is 0. Then all processors of the group are synchronized again.

A problem that has not been mentioned is what happens if a conflict occurs because of one processor executing an MP on a and one loading the content of a at the same time step. We avoid that by only executing MP commands in time steps with even numbers and LOAD commands in time steps with odd numbers when synchronizing. Our processor design supports this by a modulo flag in the processor status word that is flipped after each machine instruction[1] and with a conditional jump on the value of that flag. If the address a is stored in register R_a the code for synchronizing has the form

all processor have reached this point at the same time

[1] Remember that all machine instructions take equal amounts of time

2. STORE $R_a, R0, R0$
3. SYNC $R_a, +, 1$
4. code for high level language command
5. processors could reach this point at different times
6. JMP modulo clear PC,PC,R0
7. MP $R_a, -, 1$
8. LOAD $R_a, R0, R0$
9. JMP zero set $PC, PC, \# - 1$
10. all processors reach this point at the same time

If all processors reach line 5 at the same time, the synchronization overhead is 7 commands: (2), (3), (6), (6), (7), (8), (9).

Reality is somewhat worse because the LOAD in line 8 is delayed and therefore at least one NOP has to be performed before the content of a and thus the correct zero bit is available. In order to have line 8 executed always in an odd time step we have to include two NOP commands after line 8.

It now can happen that a part of the processors in the group reaches line (10) at the same time and the rest of the group reaches line (10) two steps later. This gap can be closed by a second synchronization part where the processors perform a SYNC on a cell with previous content zero, then load the new value and check whether all processors have reached this point. The check will fail for the "faster" processors and succeed for the "slower" ones. If it fails processors execute two NOP's. The final synchronization needs 7 commands: SYNC, LOAD, NOP, CMP, conditional JMP, 2NOP's. The overhead for the synchronization is then 9 for the first part of code and 7 for the second part, in total 16 commands.

One can still argue that 16 commands is too much for synchronizing when compared to about 10–20 commands needed as code for one high level language command. However static analysis of programs allows compilers to reduce the number of necessary synchronizations. Synchronization is only necessary at points where several instruction streams are split and merged later on and where runtime of different streams is not predictable. The FORK compiler uses this kind of analysis [10]. The exact factor of reduction however still has to be determined in practice.

Acknowledgements

We would like to thank Helmut Seidl for helpful discussions about low level software topics.

Bibliography

[1] F. Abolhassan, J. Keller, and W. J. Paul. On the cost–effectiveness of PRAMs. In *Proc. 3rd IEEE Symp. on Parallel and Distr. Processing.* IEEE, Dec. 1991.

[2] F. Abolhassan, J. Keller, and W. J. Paul. On the cost–effectiveness and realization of the theoretical PRAM model. FB 14 Informatik, SFB-Report 09/1991, Universität des Saarlandes, May 1991.

[3] R. V. Balakrishnan. Weniger Störungen auf Mikrocomputerbussen. *Elektronik*, 4:106–112, 1989.

[4] R. Beigel and C. P. Kruskal. Processor networks and interconnection networks without long wires. In *Proc. 1989 ACM SPAA*, pp. 42–51. ACM, 1989.

[5] Bicc Vero Electronics. *Elektronik Handbuch*, 1988.

[6] BT&D Technologies Ltd. *Data Communication Products*, 1991.

[7] Y. Chang and J. Simon. Continuous routing and batch routing on the hypercube. In *Proc. 5th ACM Symp. on Principles of Distr. Comp.*, pp. 272–281, 1986.

[8] The Influence of Technology on the Choice of a Multiprocessor Interconnection Network. *Proc. 2nd Workshop on Parallel and Distr. Processing*, pp. 91–110, 1990.

[9] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th ACM STOC*, pp. 114–118, 1978.

[10] T. Hagerup, A. Schmitt, and H. Seidl. FORK: A high–level–language for PRAMs. In *Proc. PARLE 91*, 1991.

[11] T. F. Knight and A. Krymm. A Self-Terminating Low-Voltage Swing CMOS Output Driver. *IEEE Jounal of Solid-State Circuits*, 23(2), 1988.

[12] Motorola, Inc. ASIC Division, Chandler, Arizona. *Motorola High Density CMOS Array Design Manual*, July 1989.

[13] Silvia M. Müller and Wolfgang J. Paul. Towards a formal theory of computer architecture. In *Proceedings of PARCELLA 90, Advances in Parallel Computing.* North–Holland, 1990. 94.

[14] C. A. Neugebauer. Materials for High-Density Electronic Packaging and Interconnections in the Higher Packaging Levels. *Journal of Electronic Materials*, 18(2), 1989.

[15] D. A. Patterson and C. H. Sequin. A VLSI RISC. *Comput.*, 15(9):8–21, 1982

[16] A. G. Ranade. How to emulate shared memory. *J. Comput. Syst. Sci.*, 42(3):307–326, 1991.

[17] A. G. Ranade, S. N. Bhatt, and S. L. Johnson. The Fluent Abstract Machine. In *Proc. 5th MIT Conf. on Advanced Research in VLSI*, pp. 71–93, 1988.

[18] D. Scheerer. Entwurf und Realisierung eines Verbindungsnetzwerkes als Teil des Multiprozessorsystems DATIS–P–256. Master's Thesis, Universität des Saarlandes, FB Informatik, 1989.

[19] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, (Ed.), *Handbook of Theoretical Computer Science, Vol. A*, pp. 943–971. Elsevier, 1990.

[20] D. S. Wise. Compact layouts of banyan/FFT networks. In H. T. Kung, B. Sproull, G. Steele, (Ed.), *Proc. CMU Conf. on VLSI Syst. and Comput.*, pp. 186–195, 1981.