On the Physical Design of PRAMs*

Ferri Abolhassan[1] Reinhard Drefenstedt[1] Jörg Keller[2]
Wolfgang J. Paul[1] Dieter Scheerer[1]

[1]Universität des Saarlandes
Computer Science Department
Postfach 1150
66041 Saarbrücken
Germany

[2]CWI
Dept. AA
Postbus 94079
1090 GB Amsterdam
The Netherlands

Abstract

The *Saarbrücken Parallel Random Access Machine* (SB-PRAM) is a scalable shared memory machine. At the gate level it is a re-engineered version of the Fluent machine [A. G. Ranade, S. N. Bhatt and S. L. Johnson. The Fluent Abstract Machine. In *Proc. 5th MIT Conference on Advanced Research in VLSI*, pp. 71–93 (1988)]. It uses hashing of adresses, combining and latency hiding. A prototype with 128 processors is presently being designed. In this paper we deal with several problems related to the physical design of this machine such as the total number of network chips, the geometrical arrangement of boards in the network and the VLSI realization of certain sorting arrays. We also present an extremely fast method to rehash addresses without use of external memory.

*Research was partially supported by DFG (SFB 124) and SIEMENS AG. A preliminary version of this paper appeared in [1].

1 Introduction

Parallel machines are nowadays classified as multi-computers and multi-processors. In multi-computers, processors exchange data by explicit message passing. In multi-processors, all processors have access to a shared address space. This leads to a more comfortable programming model.

Hardware architectures for both classes do not show much difference. Main stream realizations of scalable shared memory processors tend to have local memories as well as large caches. Transport of cache lines between processors and cache coherence protocols can be viewed as very sophisticated automated message passing protocols. Examples are the Stanford DASH multi-processor [15] and the ALLIANT FX/8 [19]. They deal in no way with the problems of hot spots (multiple processors accessing one memory cell), module congestion or highly non-local access patterns. A consequence of this implementation is a large variation of the memory access time depending on the access patterns of the processors.

Parallel machines which support both the programming model and the uniform timing behaviour of a shared memory are called PRAMs (Parallel Random Access Machine) in the theoretical literature. The problem of simulating PRAMs on processor networks has been studied in depth [10, 17, 20, 24].

A re-engineered version of Ranade's Fluent machine construction [20, 21] was proven in [2] to be cost-effective at the gate level, even in comparison with multi-computers. This motivated the present effort to design and construct a prototype, called the SB-PRAM [1]. The prototype will have 128 processors. The current designs assume a clock speed of 7 Mhz for processors and 28 Mhz for network nodes. This results in a peak performance of 900 MIPS and MFLOPS.

We will focus in this paper on difficulties that occur in the physical design of the SB-PRAM. In section 2 we give a brief overview of Ranade's Fluent Machine and its re-engineered version. In sections 3 to 5 we deal with realizations of fast sorting arrays, butterfly networks, and with rehashing the address space without using secondary storage devices.

2 The Fluent Machine – Re-engineered

2.1 PRAM Emulations

The PRAM model was introduced by Fortune and Wyllie [9]. In a PRAM, N processors work synchronously on a shared memory with unit memory access time. The access time is independent of the access pattern. The PRAM model is widely used to study parallel algorithms. Several variants exist, depending on whether concurrent access to a memory cell is allowed. In the CRCW (Concurrent Read Concurrent Write) PRAM an arbitrary number of processors can access a shared memory cell. There are several possibilities

to resolve write conflicts. We consider the strongest model, where the priorities of the processors are linearly ordered. In the case of concurrent write, the processor with the highest priority wins (priority CRCW PRAM). However, a shared memory with unit access time seems unrealistic with current technology. A lot of work has been done to emulate a PRAM on a processor network. Such a network consists of N processors, memory modules and an interconnection network.

PRAM emulations follow some general principles. First, the shared address space has to be distributed among the memory modules. The distribution has to be done in such a way that memory access requests are distributed almost evenly among the memory modules, no matter what the access pattern might be. If all accessed memory addresses are distinct, then a randomized solution to this is universal hashing, introduced by Wegman and Carter [6] and first used in PRAM emulations by Mehlhorn and Vishkin [17].

In order to access memory, requests are sent via the network to the appropriate memory modules, in case of read requests answers are sent back. Hot spots could appear if several processors concurrently access the same memory cell. This situation is handled by using a combining network, where multiple such requests are merged into one. Although routing in a combining network takes more effort, this pays off already for small numbers of concurrent accesses, because combining in software results in a large overhead [11].

A good example for an emulation is the Fluent Machine.

2.2 Fluent Machine Principles

Ranade's Fluent Machine [20, 21] uses a butterfly network with $N = (n+1)2^n$ switches, processors and memory modules. The links are bidirectional. A butterfly network with $n+1$ columns or stages is defined as a graph $G = (V, E)$ with $V = \{0, \ldots, n\} \times \{0, \ldots, 2^n - 1\}$. For $0 \leq i < n$, Node (i, x) is connected to nodes $(i+1, x)$ and $(i+1, x \oplus 2^i)$. Here, $a \oplus b$ denotes an integer with a binary representation obtained by bitwise exclusive or of the binary representation of integers a and b. We call the nodes $(0, x), \ldots, (n, x)$ a *row*, and the nodes $(i, 0), \ldots, (i, 2^n - 1)$ a *column* or *stage*.

The shared address space has size m. The distribution of addresses to modules is done by a function of the form

$$h(x) = \sum_{i=0}^{O(\log m)} a_i x^i \bmod P \bmod N \quad . \tag{1}$$

P is a prime larger than m, a particular function is chosen randomly by choosing coefficients a_i between 0 and $P - 1$. As each function distributes only very few access patterns in a way that one module gets overcrowded by requests, the chosen hash function will distribute memory traffic well for a given application program with very high probability.

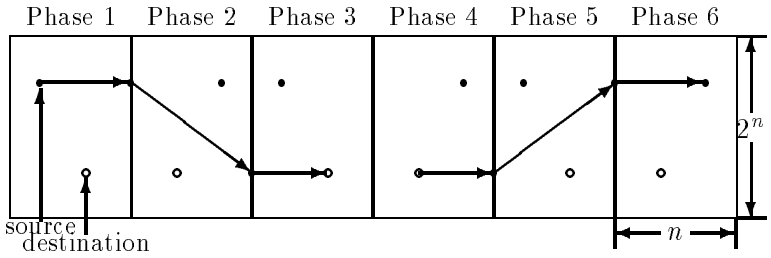

Figure 1: 6 Phase Routing of the Fluent Machine

The routing of requests from processors to memory modules and back is done in 6 phases as shown in figure 1, taken from [21]. At the beginning of phase 1, all processors inject their requests into the network. Requests consist of address and hashed address of the cell to be accessed, of the access type (read or write) and of a data word in case of a write. During phase 1, all requests or packets in a row are shifted to the end of the row and sorted by their hashed address. During phase 2, each request is routed to its destination row. In phase 3, the requests are shifted to their destination columns where memory access takes place. In the last three phases, each request traverses its path in reverse direction and returns to the processor that initiated the request.

During phase 2, the requests are kept sorted. If both input buffers of a routing switch contain packets, the one with the smaller hashed address proceeds. If both addresses are identical, the two packets are combined into one[1]. The sorted order guarantees that all packets destined for the same address will meet and be combined into one. The idea of sorting could introduce additional waiting times. Consider figure 2 which is taken from [21]. Switch B cannot transmit the request it holds, because a request with destination smaller than 25 arriving on the upper input would destroy the sorted order. However, if all requests handled by switch A take the upper output, switch B would not get any further information and request 25 would be stuck.

To avoid this, each switch that transmits a request to one output sends a GHOST message along the other output. The GHOST carries the same address but has a special type GHOST and no data. In figure 2, a GHOST with address 35 sent by switch A would ensure that packet 25 could be sent, because future messages along the upper input all must have addresses larger than 35 due to the sorted order.

The functionality of the switches is presented here in detail because it can be used to reduce network complexity (see section 4.1).

In phases 4 to 6, no routing decisions are made. The decisions of phases 1 to 3 are recorded in "direction queues" in each switch and are used to control the behaviour of the switch in the remaining phases.

[1] To compare addresses, one needs to compare the unhashed addresses as well, because the hash function is not necessarily bijective.

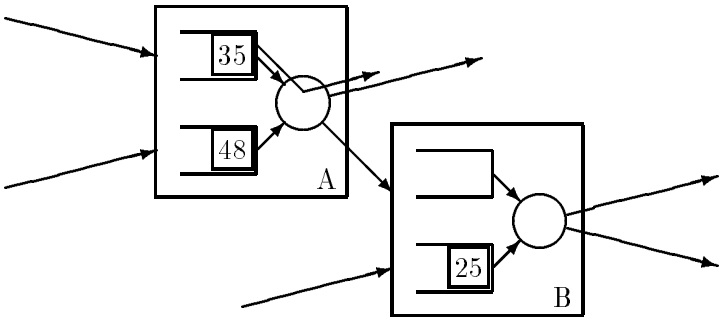

Figure 2: Function of Ghost Packets

Ranade proves [20] that with the hash function (1) and with the above routing algorithm, one step of the PRAM can be emulated in time $b \log n$ with overwhelming probability, when the buffers in the switches can hold b packets.

In the memory modules of the Fluent Machine the problem of secondary hashing arises. The solution sketched in [20] blows up the size of the modules by a constant factor.

2.3 Re-engineering the Fluent Machine

A drawback of Ranade's design is that processors are idle most of the time. In order to change this, it would be necessary to pipeline the 6 routing phases. It is not necessary to build six butterfly networks, because only in phases 2 and 5 the butterfly network is used. In phases 1 and 6, the rows are sorted, in phases 3 and 4, rows are shifted. We therefore will use two butterfly networks to realize phases 2 and 5, use linear sorting arrays [14] for phases 1 and 6, and use 2^n modules with multiple banks to omit phases 3 and 4. A more detailed description of the changes made can be found in [2].

We realize the processors of one row by one physical processor that runs cn virtual processors in a pipeline. We obtain a total of $p = 2^n$ physical processors and $N = cnp$ virtual processors. Each virtual processor has its own register set in hardware. The instruction set is similar to that of the Berkeley RISC processor [18], each instruction takes the same amount of time. The pipeline depth of the physical processor is adjusted to the delay of a read request. Thus the memory access latency is hidden from the user. With the help of the delayed load technique[2] c can be reduced by a factor of 2. The concept of multiple register sets in hardware to hide latency is also used in the MIT Alewife [4] and in the TERA computer [5], it was first used in the Denelcor HEP [23].

Also the hash function is changed. Evaluation of a polynomial of degree $O(\log m)$ is slow and expensive in terms of hardware. We will use a linear function of the form $h(x) = ax \bmod m$. The upper n bits of h denote the module, lower bits define the local address within the

[2]The answer to a load instruction is assumed to be available in the next but one instruction.

module. Simulations show that performance with this simple function is sufficient [8] and the constant c can be bounded by 3. As our processor has a load/store architecture[3] we can use one multiplier for both multiply instructions and for hashing. This significantly reduces the amount of extra hardware needed for hashing. Also h is bijective, so no secondary hashing is necessary.

In [2] some of the authors investigated cost and speed of the two designs in a formal framework. The basic idea is to describe designs at the gate level, and to evaluate their cost and speed as gate equivalents and gate delays. It turned out that the re-engineered design was slightly cheaper in terms of hardware and 5.5 times faster than the original Fluent Machine. As both machines have identical processors and instruction sets, this comparison is independent of a particular benchmark.

3 Sorting in Linear Arrays

To realize phase 1 of the routing algorithm, we need to sort the requests sent by a processor, before sending them to the butterfly network for phase 2. The requests arrive sequentially and they have to be sent sequentially into phase 2. Our goal is to realize the sort with a cheap sorting device as fast as possible. The device shall be built of elements that can store two requests and can make one comparison per cycle. Assuming that t requests arrive per round to sort, the fastest we can achieve is to sort in time $2t$, which means that the first output appears immediately after the last input has arrived.

The problem of sorting items that arrive sequentially has been investigated before. Leighton gives a simple algorithm that uses t sorting elements bi-directionally connected as a one-dimensional array [14, p. 5ff]. Data items enter and leave the array at the leftmost element of the array. The algorithm works in 3 phases. In the first phase, each element accepts an item coming from its left neighbour, and either stores the item or compares it to the item that is currently held. The larger one proceeds to the right, the smaller one is stored. At the end of this phase, the array contains the items in sorted order. The second phase consists of informing all elements about the end of input by sending a tag from element 0. In the third phase, the items leave the array in sorted order. The phases can be partly overlapped. The tag of the second phase can be attached to the last entering item. When this item reaches its position in the sorted sequence, the tag proceeds alone. Each element starts to send items to the left as soon as the tag has passed. Hence, the first item leaves the array immediately after the last item entered the array. Subsequent items leave the array every other cycle, because the tag needs i steps to reach element i and the item stored there needs i steps to leave the array at element 0. The algorithm runs in $3t$ cycles: t until the last item enters the array and $2t$ until all items have left the array.

We use a method from [3] to improve the runtime of the algorithm. We use a global control wire to inform all elements of the device after t cycles simultaneously about the end of

[3] Memory access only happens in load and store instructions, compute instructions only use registers.

input. But when all elements start to send their items towards the output element 0, not all items have reached yet their final positions. Therefore, in the third phase all elements also have to compare items. Smaller ones proceed further to the left, larger ones are kept until they reach their position. The runtime of the algorithm is reduced to $2t$.

In order to have the algorithm perform combining, we extend the first element such that it holds each item for one extra cycle before it leaves the array. In the next cycle this item is compared with the next one to follow. If the items are identical, the first one is erased. This only extends the first array element by some control logic, a register and a comparator unit.

In order to realize the reverse sorting of phase 6, we extend the first array element by a counter that adds a tag to each item as it enters. The tag specifies the item's position in the input stream. When the sorted sequence leaves the array, the tags are stored in a FIFO queue. To realize phase 6, we use a second sorting array. Answers coming from the network are given a tag from the FIFO queue and then are sorted by the tags.

Both extensions work for arbitrary sorting devices with sequential input and output.

While we need $2t$ cycles to sort t items, we are still able to sort pieces of length t of a continuous stream of requests with one sorting array. In cycle $2t - i$, where $1 \leq i \leq t$, only the first i elements of the sorting array are still occupied. This means that after t cycles, we can start to feed another piece of t items at element $t - 1$ into the array and reverse the directions in the algorithm. In general, we use elements 0 and $t - 1$ alternately to feed t items into the array.

4 Reduction of Network Complexity

Network design is always difficult, because network latency is of crucial importance for the machine performance. The main problems are how to map network nodes onto network chips, how to place these chips onto printed circuit boards (PCB's) and how to arrange these boards such that wiring between them allows one to use standard connector and rack/cabinet technology.

4.1 Reducing Chip Count

The network will be built out of semi-custom chips designed in sea-of-gates technology. On each chip we realize a u-stage butterfly network with $u2^{u-1}$ nodes and 2^u input and output links. Assume we have a fixed pin count of W pins on each chip. In order to reach a link width of $w \leq \lfloor W/4 \rfloor$, we have to choose u such that $2^{u+1}w \leq W$, hence $u \leq \log(W/w) - 1$ ($\lfloor W/4 \rfloor$ is the maximum possible link width because a single butterfly node already has 2 inputs and outputs). If we want to reach a fixed number of stages u, we have to choose w such that $2^{u+1}w \leq W$, hence $w \leq W/2^{u+1}$.

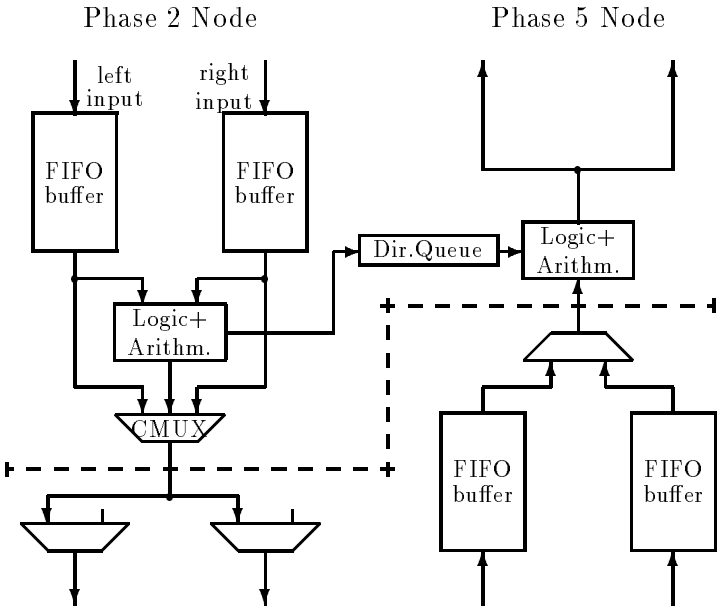

Figure 3: Data Paths of a Pair of Network Nodes

We will use a method from [7] that, starting from a given mapping of the above type, will increase the number of nodes per chip by a factor of two without changing pin count or reducing link widths. Furthermore, the new network built of those chips still is a butterfly network.

Consider again a network node. As described in section 2, only one packet is selected and transmitted at each cycle. On the other outgoing link of the node, a GHOST packet is transmitted, that is identical to the original packet except for the type. This leads to an implementation of a node as shown in figure 3. The central multiplexer CMUX selects one packet, the multiplexers on the outgoing links serve to replace the original request type by GHOST on one side. The right part of the figure shows the node for phase 5. The dashed line in the middle of the figure shows that we can make a cut through each node that has essentially the width of one link (plus one or two control signals).

Now, instead of taking a u-stage butterfly, we implement a $(u+1)$-stage butterfly on one chip, but take only the lower half of the nodes in the first stage, and only the upper half of the nodes in the last stage. An example for $u = 1$ is shown in figure 4. With this implementation we still have 2^u input and output links, but $u-1$ stages of full nodes, one stage of upper halfs and one stage of lower halfs of nodes, totaling to $u2^u$ nodes per chip. Hence we use twice the number of gates per chip.

Note, that in the first (last) network stage of Ranade's algorithm, only one input (output) per node is used, and that it is therefore sufficient to use lower (upper) half nodes in those stages.

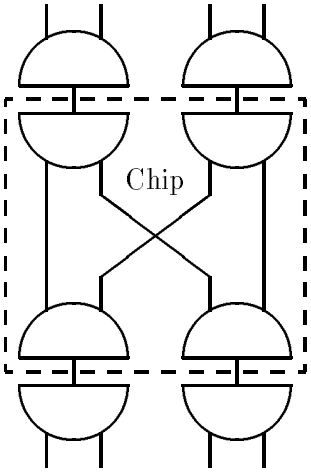

Figure 4: Partitioning of Network Nodes for $u = 1$

For our prototype, the network chips will be heavily restricted by pin count. Links in the prototype have a width of $w = 104$ bits [1], gate arrays we can afford are restricted to $W = 220$ pins. We see that we must already send a packet in two pieces in order to be able to implement four links on a chip. The first piece contains the address and the access mode, the second piece contains the data. Because we do not want to slow down our network further, we take $u = 1$. We implement a 2-stage butterfly network on one chip, but take only upper (lower) half nodes for the first (last) stage.

The chips still remain connected as a butterfly network [7].

4.2 Arranging Network Boards in Three-dimensional Space

Although butterfly networks have nice properties when viewed as graphs, they are hard to arrange physically. Vitányi proves that the average wire length in a butterfly network cannot be a constant even under relaxed conditions (nodes have unit volume and arbitrary shape, wires have zero volume) [25]. Moreover, an obvious implementation leads to wiring that is not regular and therefore not suited for an implementation with standard components.

Wise and Knight propose solutions by embedding butterfly networks on printed circuit boards in three-dimensional space. Wise proposes to cut a u-stage butterfly network in two parts after $u/2$ stages [26]. Each part decomposes into $2^{u/2}$ butterflies with $u/2$ stages each. These smaller butterflies form a complete bipartite graph. Assuming that each $(u/2)$-stage butterfly network fits on one board, the boards can be arranged as shown in figure 5.

Knight proposes to switch to a topologically equivalent network where wiring between all stages is identical [13]. Assume that one stage of nodes plus wiring to the next stage fits onto one board. He connects boards vertically with special connectors and uses a stack of boards to implement the network.

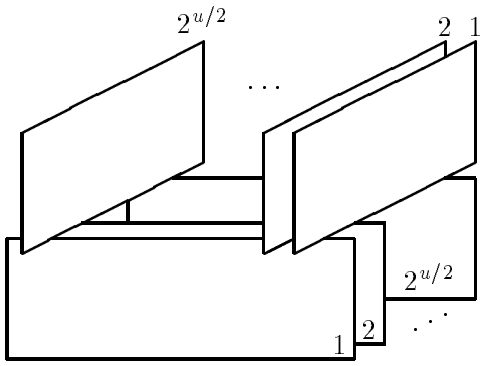

Figure 5: Wise's Arrangement of Boards

Both solutions suffer from the disadvantage that they do not scale because of the restriction of board sizes. Furthermore, they require special technology.

Now assume that we make two cuts in the network: one after $u/3$ stages and one after $2u/3$ stages. Then we have three parts, each consisting of $2^{2u/3}$ butterfly networks, each with $u/3$ stages.

Theorem 1 *We can number the butterflies of each part from* $(1,1)$ *to* $(2^{u/3}, 2^{u/3})$ *such that butterfly* (i,j) *of part one only is connected to butterflies* (k,j) *of part two for all* k *in* $\{1,\ldots,2^{u/3}\}$*, and that butterfly* (i,j) *of part two only is connected to butterflies* (i,k) *of part three for all* k *in* $\{1,\ldots,2^{u/3}\}$*.*

A proof of Theorem 1 can be found in [11].

The above numbering motivates an arrangement as shown in figure 6 for $u = 6$. To make the arrangement symmetric, parts one and three have been split into two halves each. Assume that a $(u/3)$-stage butterfly network fits onto one board. The boards of each part are arranged in a square, the squares are arranged in a manner that all wiring between boards is horizontal or vertical. Two example wirings are shown in figure 6.

Note that similar arrangements can be made if u is not a multiple of 3. If $u = 3u' - 1$ then boards in part three house two $(u'-1)$-stage butterflies, whereas boards in parts one and two house u'-stage butterflies. If $u = 3u' - 2$ then boards in parts one and three house two $(u'-1)$-stage butterflies.

With this method, we are able to use standard technology to put boards into racks and staple racks into cabinets. If links are narrow, then they can leave the boards via connectors to the backplane, the wiring can be made on the rear side of the cabinet. If links are wide, then links can leave boards via connectors at the front of the cabinet. The links form wiring channels between the boards. However, boards still can be removed easily, and cooling and power supply do not collide with wires.

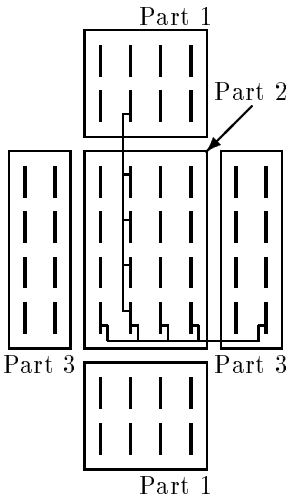

Figure 6: Arrangement of Boards for $u = 6$

For our prototype, we have to implement an 8-stage butterfly network. With the method described in the previous subsection, this reduces to a 7-stage butterfly network of chips. The network boards in parts one and three contain two 2-stage butterflies (8 network chips), the boards in part two contain a 3-stage butterfly (12 chips). In each part there are 16 boards.

5 Rehashing

Hash functions normally distribute memory access requests well among the memory modules. If however some memory module is requested much more than the others, latency increases, in the worst case to a time linear in the number of processors. Then it is necessary to choose a new hash function for this particular application. A solution is to interrrupt execution of the application, choose a new hash function h', redistribute data according to h' and continue with the application. This is called *rehashing*. Lipton and Naughton show in [16] how to construct cases where rehashing is needed.

There might be cases where rehashing does not pay off, e.g. when an application will finish shortly after detection. Then it might be better to finish with the old hash function. The exact bounds when to invoke rehashing depend on the particular implementation and application. They can be examined when the prototype is available.

Rehashing looks like a simple task if each of the p physical processors has access to secondary storage (such as disks) of size m/p. Each virtual processor reads m/N words from the shared memory (using the old hash function h) and stores them locally. Now we have a backup of the complete shared memory. The hash function is switched to h' and each processor

writes its data back to shared memory. This solution runs asymptotically in optimal time $O(m/p)$. Unfortunately the hidden constant factor is large because disks are slow compared to processor speed. Furthermore, using additional space of size m is a waste. We show how to detect the necessity to rehash and how to rehash fast without using secondary storage.

5.1 Detection

The necessity to rehash can be detected by counting the fraction of stalled cycles in the last x cycles. If this fraction gets larger than a certain user-defined threshold $1/t$, then rehashing is initiated. This detection can be done by maintaining two counters $\mathrm{CO}_{\mathrm{ST}}$ and $\mathrm{CO}_{\mathrm{TO}}$ for the number of stalled and the number of total cycles, and a register R for storing t. In the beginning, both counters are set to zero. If $\mathrm{CO}_{\mathrm{TO}}$ reaches x, we want to check whether

$$\frac{\mathrm{CO}_{\mathrm{ST}}}{\mathrm{CO}_{\mathrm{TO}}} > \frac{1}{t}.$$

To do this, we multiply $\mathrm{CO}_{\mathrm{ST}}$ with t and subtract $\mathrm{CO}_{\mathrm{TO}}$ from it. If the result is positive, we initiate rehashing. Afterwards, the counters are set to zero again.

This allows the user to define a threshold in a wide range, and detection can be made without floating point operations or divisions. The value of t might depend on the application. Typical values have to be figured out after completion of the prototype.

5.2 Fast Execution

We assume that both the old and new hash function h and h' bijectively map addresses to cells. Then the redistribution of data can be viewed as a permutation π of the addresses while h is still used. After rehashing, address x will be mapped to cell $y = h'(x)$. But before rehashing, address $h^{-1}(y)$ is mapped onto cell y. Hence $\pi = h^{-1} \circ h'$. Permuting a set of data items without additional storage is normally done by splitting the permutation into its cycles and permuting cycles one by one. If we want to do this in parallel we face two problems: extracting the cycle structure from π, and scheduling the cycles among processors such that work is evenly distributed. We solve these problems for the case that h and h' both are linear functions.

Let $h(x) = ax \bmod m$ and $h'(x) = a'x \bmod m$ where a and a' are relatively prime to m. In $\mathbf{Z}/m\mathbf{Z}$, the numbers relatively prime to m form a multiplicative group, the group of units [22, p. 119]. It follows that a and a' can be inverted and that h and h' are bijective. Then $\pi(x) = h^{-1}(h'(x)) = a^{-1}a'x \bmod m$. As a and a' are units, $b = a^{-1}a' \bmod m$ also is a unit and $\pi(x) = bx \bmod m$ is bijective. We investigate $m = 2^u$. The group of units here is the set of odd numbers between 1 and $m - 1$.

For x in $\{0,\ldots,m-1\}$ we define $j(x) = \max\{k | x \text{ can be divided by } 2^k\}$. Then every x in $\{0,\ldots,m-1\}$ has a unique representation $x = 2^{j(x)}x'$ where $0 \leq j < u$ and $x' < m/2^{j(x)}$ is odd. We can now partition the set $U(m) = \{0,\ldots,m-1\}$ into sets

$$U_k(m) = \{x \in U(m) | j(x) = k\} = \{x \in U(m) | x = 2^k x' \text{ and } x' < m/2^k \text{ odd}\} \quad .$$

We apply π to an address x in $U_k(m)$. $\pi(x) = bx \bmod m = b2^k x' \bmod m$. As b and x' are units, $\tilde{x} = bx' \bmod m/2^k$ also is a unit and $2^k\tilde{x} \bmod m = 2^k(bx' - rm/2^k) \bmod m$ (for some r) $= \pi(x)$. Hence $\pi(x)$ is an element of $U_k(m)$, too. We conclude that each cycle of π is contained completely in one of the $U_k(m)$. Furthermore $\phi_k(x) = x/2^k$ is a bijection from $U_k(m)$ to $U_0(m/2^k)$, $\pi_k(x') = bx' \bmod m/2^k$ is a permutation on $U_0(m/2^k)$ and for $x \in U_k(m)$ we have $\pi(x) = \phi_k^{-1}(\pi_k(\phi_k(x)))$. We therefore restrict our attention to the problem of permuting odd numbers ($U_0(m/2^k)$) and then apply this method by using ϕ_k^{-1} to permute $U_k(m)$.

Note that $U_0(m)$ is the set of units and hence a multiplicative group. Consider the cycles of π when applied on $U_0(m)$. A cycle starting with an element x has the form $x, bx, b^2x \ldots, b^{l-1}x, x$. Then l is the order of b in $U_0(m)$. We can conclude that all cycles have the same length, which must be a power of two because the order of $U_0(m)$ is a power of 2. The number of cycles $\sigma = |U_0(m)|/l$ then also is a power of two. Scheduling of cycles is easy because all cycles have the same length. To obtain the structure of the entry elements, we exploit the fact that U_0 is generated by -1 and 5. Thus each element of $U_0(m)$ has a unique representation $(-1)^\alpha 5^\beta$, where $\alpha \in \{0,1\}$ and $\beta \in \{0,\ldots,m/4-1\}$ [22, p. 124].

Let $b = (-1)^f 5^g$. Then $b^l = (-1)^{lf} 5^{lg}$ and $b^l = 1$ because l is the order of b. It follows that $lg = 0 \bmod m/4$ and hence either $g = 0$ or g is a multiple of $m/(4l)$. We substitute l by $|U_0(m)|/\sigma = m/(2\sigma)$ and obtain that g is a multiple of $\sigma/2$. With this we obtain

Theorem 2 *For $b \neq -1$, all elements of the form 5^d and $(-1)5^d$, $0 \leq d < \sigma/2$, are on different cycles. For $b = -1$, all elements of the form 5^d, $0 \leq d < \sigma$, are on different cycles.*

A proof of Theorem 2 is straightforward and can be found in [12].

We now proceed as follows. When rehashing is invoked, we compute the order of b in $U_0(m)$ and compute the number σ of cycles. If $\sigma \geq N$, then each virtual processor is assigned σ/N cycles that it permutes sequentially. If there are less than N cycles, then N/σ processors work together on one cycle, each permuting a piece of length $l/(N/\sigma) = |U_0(m)|/N$.

For each processor the first entry element for a cycle in $U_0(m)$ can be computed from table of values 5^{2^i} in time $O(\log m)$. Each subsequent entry element of a cycle in $U_0(m)$ requires time $O(1)$. The table can be provided to each processor in non-volatile memory. Because there are $\log m$ sets $U_k(m)$, the total overhead is $O((\log m)^2 +$ total number of cycles$/p)$. The runtime of the rehashing algorithm is $O(m/p + \log^2 m)$.

References

[1] F. Abolhassan, R. Drefenstedt, J. Keller, W. J. Paul and D. Scheerer, On the physical design of PRAMs. In J. Buchmann, H. Ganzinger and W. J. Paul, (eds.), *Informatik — Festschrift zum 60. Geburtstag von Günter Hotz*, pp. 1–19. Teubner, Stuttgart (1992).

[2] F. Abolhassan, J. Keller and W. J. Paul, On the cost–effectiveness of PRAMs. In *Proc. 3rd Symp. on Parallel and Distributed Processing*, pp. 2–9. IEEE CS Press, Los Alamitos (1991).

[3] F. Abolhassan, J. Keller and D. Scheerer, *Optimal sorting in linear arrays with minimum global control.* Report CS–R9244, CWI, Amsterdam, The Netherlands (1992).

[4] A. Agarwal, D. Chaiken, K. Johnson *et. al.*, *The MIT Alewife machine: A large scale distributed–memory multiprocessor.* Technical Report MIT/LCS/TM–454, Massachusetts Institute of Technology, Cambridge, MA (1991).

[5] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield and B. Smith, The Tera computer system. In *Proc. 1990 Internat. Conf. on Supercomputing*, pp. 1–6. ACM, NY (1990).

[6] J. Carter and M. Wegman, Universal classes of hash functions. *J. Comput. System Sci.*, **18**, 143–154 (1979).

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

[8] C. Engelmann and J. Keller, Simulation-based comparison of hash functions for emulated shared memory. In *Proc. PARLE '93, Parallel Architectures and Languages Europe*, pp. 1–11. Springer, Berlin (1993).

[9] S. Fortune and J. Wyllie, Parallelism in random access machines. In *Proc. 10th Symp. on Theory of Computing*, pp. 114–118. ACM, NY (1978).

[10] A. R. Karlin and E. Upfal, Parallel hashing: An efficient implementation of shared memory. *J. Assoc. Comput. Mach.*, **35**, 876–892 (1988).

[11] J. Keller, *Zur Realisierbarkeit des PRAM Modells.* PhD thesis, Computer Science Department, Universität des Saarlandes, Saarbrücken (1992).

[12] J. Keller, Fast Rehashing in PRAM Emulations. In *Proc. 5th Symp. on Parallel and Distributed Processing.* IEEE CS Press, Los Alamitos (1993).

[13] T. F. Knight Jr., Technologies for low latency interconnection switches. In *Proc. 1989 Symp. on Parallel Algorithms and Architectures*, pp. 351–358. ACM, NY (1989).

[14] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publ., San Mateo (1992).

[15] D. Lenoski, J. Laudon, K. Gharachorloo *et. al.*, The Stanford DASH multiprocessor. *Comput.*, **25**, 63–79 (1992).

[16] R. J. Lipton and J. F. Naughton, Clocked Adversaries for Hashing. *Algorithmica*, **9**, 239–252 (1993).

[17] K. Mehlhorn and U. Vishkin, Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Inform.*, **21**, 339–374 (1984).

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

[19] R. Perron and C. Mundie, The Architecture of the ALLIANT FX/8 Computer. In *Proc. COMPCON Spring 86*, pp. 390-393. IEEE CS Press, NY (1986).

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

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

[22] H.-J. Reiffen, G. Scheja and U. Vetter, *Algebra*. B.I.–Wissensch.v., Mannheim (1984).

[23] B. Smith, A pipelined shared resource MIMD computer. In *Proc. 1978 Internat. Conf. on Parallel Processing*, pp. 6–8. IEEE, NY (1978).

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

[25] P. M. B. Vitányi, Locality, communication and interconnect length in multicomputers. *SIAM J. Comput.*, **17**, 659–672 (1988).

[26] D. S. Wise, Compact layouts of Banyan/FFT networks. In H. T. Kung, B. Sproull and G. Steele (eds.), *Proc. CMU Conference on VLSI Systems and Computations*, pp. 186–195 (1981).