Hashing and rehashing in emulated shared memory

J. Keller

Computer Science/Department of Algorithmics and Architecture

CS-R9240 1992

Hashing and Rehashing in Emulated Shared Memory

Jörg Keller
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

The PRAM model is widely used to formulate parallel algorithms because of its shared memory and its synchronous behaviour. The model however bears little resemblance to real parallel machines. This has led to various approaches to emulating PRAMs on processor networks. We briefly survey the principles behind these emulations and show why hashing is an important part of them. We discuss the commonly used types of hash functions and their theoretical properties and relate these to their behaviour in simulations. Both synthetic and application based traces are used as input data for these simulations. The simulation results suggest the use of linear functions with randomly chosen coefficients. These functions also have the advantage of short evaluation time and bijectivity. But no matter which type of hash function is chosen, it may be necessary to choose a new hash function on the fly. We present an algorithm to rehash linear functions in optimal time without using shade memory or mass storage systems during rehashing.

1991 Mathematics Subject Classification: 68M07, 68M20, 68Q25
CR Categories: B.3.2, B.3.3, E.2, F.2.2
Keywords and Phrases: Hashing, Rehashing, Linear Functions, Performance Evaluation, Emulated Shared Memory
Note: Part of this work was done while the author was working at the Univ. des Saarlandes, Saarbrücken, Germany. This research is partially supported by the German Science Foundation (DFG) under SFB 124 – D4 and by the Dutch Science Foundation (NWO) through NFI Project ALADDIN under Contract number NF 62–376. The simulation–based comparison is joint work with Curd Engelmann of Univ. des Saarlandes, Saarbrücken, Germany.

1 Introduction

Parallel shared memory machines provide its user with the view of one global address space which simplifies the design of parallel algorithms. There is no need of partioning data or communication between processors as in distributed memory machines. A synchronous machine model of the first kind is wellknown in theoretical computer science as a *PRAM (parallel random access machine)* [7]. A large body of parallel algorithms exists for the PRAM model [8, 11]. Parallel machines which implement a shared memory by a physical global memory connected with the processors by a bus and a cache are however restricted to small size (at most 32 processors currently) because a bus cannot be scaled. Examples are ALLIANT FX/8 or ENCORE Multimax [9].

Approaches for realizing shared memory machines of larger size use distributed machines with $n = 2^t$ processors and memory modules interconnected by a packet switching network. The address space $M = \{0, \ldots, m-1\}, m = 2^u$, is mapped onto cells in range M via a hash function $h : M \rightarrow M$. For address $x \in M$, $h(x)$ div m/n is the number of the module on which cell $h(x)$ is located, $h(x)$ mod m/n gives the local address of $h(x)$ on that module. Memory access to address x now consists of sending a packet containing $h(x)$ to the right module, accessing the appropriate cell locally on that module and, in case of a LOAD, sending the content of that cell back.

Emulations of that kind have been studied in depth by both theoretical and practical computer scientists. Valiant gives a good overview of theoretical results [21, 20], Ranade developed one of the first optimal routing algorithms for machines of that kind [16, 17]. Abolhassan, Keller and Paul showed the efficiency of these approaches in a formal model [2] (it was formerly believed that despite optimal asymptotic efficiency the constants in these constructions were too large for practical use) and presented the design of a prototype [1]. On the practical side some of these features were implemented already in the IBM RP3 [15]. The Tera Computer uses some of these features too [3], and the next generation transputers are announced to be suited for that [12].

Hash functions for shared memory machines should distribute (almost) all access patterns as evenly as possible among memory modules, they should be evaluable fast and should be bijective to avoid secondary hashing on modules. The reasons to demand this are obvious: If cases happen where the number of requests per module (the so called *module congestion*) is too high, then performance gets very poor, a long evaluation time increases memory access time, and secondary hashing on memory modules complicates an implementation unnecessarily.

Several types of hash functions have been proposed. But their theoretically provable properties are asymptotical results. As currently available machines are quite small (the number n of processors and memory modules usually is less than 1000) the actual behaviour of the chosen hash function can differ quite much from these theoretical properties. The lack of experimental data makes the selection of a particular hashing scheme difficult in practice. We are not aware of comparisons of hash functions based on simulated behaviour.

The first goal of this investigation is to provide these data by comparing four types of hash functions by simulations. The simulations suggest the use of linear functions as $h(x) = a \cdot x \bmod m$. They fulfil all three requirements.

However if a program exhibits many patterns that are badly distributed by a particular function one has to choose a new one and redistribute the address space. This is called *rehashing*. We will show that rehashing in case of linear functions can be easily implemented with optimal runtime $O(m/n)$ for an address space of size m and n processors. The constant involved is very small. The algorithm does neither need shade memory nor mass storage systems during rehashing.

The paper is now organized as follows: In section 2 the most common types of hash functions are introduced. Section 3 describes the experiments made and discusses the results. Section 4 presents the rehashing algorithm.

2 Hash Functions

As already mentioned, a hash function serves to map a global address space onto distributed memory modules. More formally, for an address space M of size $m = 2^u$ and a set N of $n = 2^t$ memory modules, the mapping is a function $h : M \to M$ that maps addresses to memory cells. The function $mod : M \to N, mod(x) = x$ div 2^{u-t} specifies the module of a memory cell x, the function $loc : M \to M', loc(x) = x \bmod 2^{u-t}$ specifies the local address of cell x.

An optimal mapping function h should guarantee low module congestion for almost all possible access patterns (if all addresses of one pattern are distinct). This is achieved by using classes of functions in which each function has low module congestion for almost all patterns. A particular function is randomly chosen. This guarantees with very high probability that the current application does not exhibit the patterns on which the chosen function produces hot spots. This method is called *universal hashing* and was introduced by Wegman and Carter [4].

An additional problem consists in patterns with several processors concurrently accessing one cell. This problem cannot be solved by hashing. However there exist routing algorithms that perform *combining.* Requests that access the same cell are merged during routing, answers are duplicated. Ranade's emulation algorithm [16] is a good example. Therefore, concurrent access does not increase module congestion.

A class that restricts module congestion to $O(\log n)$ is

$$\mathcal{H} = \left\{ p(x) = \left(\sum_{i=0}^{\xi} a_i \cdot x^i \right) \bmod P \bmod m : 0 \leq a_i < P \right\}$$

P is a prime larger than m, $\xi = O(\log n)$. A function of $\mathcal{H}$ is obtained by randomly choosing the values for a_i. This class was used in several theoretical investigations [10, 13, 16] to emulate shared memory on a processor network. The module congestion of $O(\log n)$ is sufficient because access from processors to memory modules across a constant–degree interconnection network needs time $\Omega(\log n)$ anyway.

However the functions in $\mathcal{H}$ are not bijective. This means that several addresses of the shared memory could be mapped onto the same cell. This requires secondary hashing on each memory module. Ranade [16] describes a method that performs secondary hashing in constant time and increases the size of the memory module only by a constant factor.

In practice however secondary hashing and waste of memory should be avoided because a constant factor of performance loss can destroy an asymptotically good result. Furthermore, the time to evaluate the hash function should be short. The functions in $\mathcal{H}$ require $\xi = O(\log n)$ multiplications and additions and a modulo division by a prime which needs a lengthy computation.

Therefore some alternatives were proposed:

- For $\xi = 1$ one obtains a linear function. This reduces evaluation time to one multiplication, one addition and one modulo division. The function is still not bijective.
- Furthermore if the modulo division by a prime is skipped and the coefficient a_0 is set to zero, the evaluation time is reduced to one multiplication. The operation modulo m is not counted because m is a power of two. If only odd values are chosen for a_1 the function also is bijective.
- If the binary representation of an address is seen as a boolean vector, the hash function consists of multiplying this vector with an invertible boolean matrix. The time to evaluate this function is shorter than one multiplication.

Dietzfelbinger proved that the first alternative is asymptotically equivalent to the second [5]. Furthermore he proves that linear functions can result in a module congestion of $\Theta(\sqrt{n})$ for patterns with addresses of the form $b + s \cdot i, i = 0, \ldots, n-1$ for constant base b and stride s [6]. This means that linear functions modulo a power of two are asymptotically worse than polynomials. However, Ranade who used polynomials in the proof of the efficient distribution of variables in a PRAM emulation [16] already mentioned that in his simulations a linear function was sufficient.

The third alternative was used in the design of the IBM RP3. Norton and Melton [14] introduce a class of boolean matrices where all matrices are invertible (which means bijectivity). Optimal distribution can be guaranteed for patterns with strides where s is a power of two and where in the binary representation of base b bits s to $s + \log n - 1$ are zero. For other bases the module congestion is at most 2. No theoretical results are given for other patterns, but their simulations hinted that distribution was acceptable for other patterns, too. One particular matrix is obtained by randomly choosing several bits of the matrix and then computing all the other bits with respect to the above properties.

3 Simulations

In order obtain effective simulation results, we had to choose our workloads carefully. Subsection 3.1 describes in detail which workloads were chosen and why. The experiments made are described in

```
(* Init rank $R$ *)
**for** $i := 0$ **to** $n-1$ **pardo**
  **if** $F[i] = i$ **then** $R[i] := 0$ **else** $R[i] := 1$
**od** ;
(* Compute rank $R$ *)
**for** $t := 1$ **to** $\lceil \log n \rceil$ **do**
  **for** $i := 0$ **to** $n-1$ **pardo**
    $R[i] := R[i] + R[F[i]]$ ;
    $F[i] := F[F[i]]$ ; (* Pointer jumping *)
  **od** ;
**od** ;
```

FIGURE 1. Code for list ranking

subsection 3.2. Subsection 3.3 discusses the results in two ways: the behaviour of one type of hash function on the different benchmarks and the behaviour of different types of hash functions on one benchmark.

3.1 Workloads

The workloads are chosen to compare the hash functions with respect to known differences, especially behaviour on access patterns with strides, and with respect to patterns taken from applications. Therefore both synthetically generated patterns and application traces were taken.

The synthetic traces consist of randomly chosen patterns as a reference and strides with $s = 1, 13, 32$. The strides were chosen to compare matrix hashing and the other hash functions and to check whether linear functions get worse on these patterns. For $s = 32$ and $s = 1$ matrix hashing is optimal [14]. Theoretical results about the performance on the others are not known.

The traces were taken from three application programs: list ranking, matrix multiplication and connected components. The reasons for taking traces from applications are the variety of produced patterns and the structure of single patterns that often is more complex and less regular than in synthetical traces.

The list ranking algorithm is taken from a survey [11]. For a given linked list of n elements, the distance (or *rank*) to the end of the list is computed for each element. The algorithm needs n processors and $O(\log n)$ time on a PRAM. The list is represented as an array F, where $F[i]$ means successor of i in the list. For the last element of the list, $F[i] = i$. The rank is contained in array R. The PARDO code is shown in figure 1.

The access patterns of this algorithm partly depend on the structure of the list and partly are strides with $s = 1$.

In the matrix multiplication algorithm $C = A \cdot B$, each processor computes one element of the destination matrix C. In order to avoid concurrent accesses, all processors start at different rows and columns of the matrices A and B. The PARDO code is shown in figure 2. The dimensions of the matrices are as follows: For $n = 2^{2z}$, all matrices have dimension $2^z \times 2^z$. For $n = 2^{2z+1}$, C and A have dimension $2^z \times 2^{z+1}$, B has dimension $2^{z+1} \times 2^{z+1}$. The algorithm needs n processors and takes time $O(n^{1/2})$.

The access patterns of this algorithm only depend on the dimensions of the matrices.

The connected components algorithm was adapted from Shiloach and Vishkin [19]. For a given undirected graph with $n = \max(2 \cdot \#edges, \#nodes)$, the connected components are computed. The algorithm needs n processors and takes time $O(\log n)$. The graph is represented by two arrays $HEAD, TAIL$. For a given edge e, $HEAD[e]$ and $TAIL[e]$ give the nodes to which e is adjacent.

```
(* $n = 2^{2z+w}$, $w \in \{0,1\}$ *)
$k := 2^z$; $m := 2^{z+w}$; $l := 2^{z+w}$;
**for** $(i,j) := (0,0)$ **to** $(k-1, m-1)$ **pardo**
  $C[i,j] := 0$ ; (* Init $C$ *)
**od** ;
**for** $r := 0$ **to** $l-1$ **do**
  **for** $(i,j) := (0,0)$ **to** $(k-1, m-1)$ **pardo**
    $C[i,j] := C[i,j] +$
      $A[i, (i+j+r) \bmod l] \cdot B[(i+j+r) \bmod l, j]$
  **od**
**od** ;
```

FIGURE 2. Code for matrix multiplication

The components are represented by an array F. Two nodes u, v are in the same component if and only if $F[u] = F[v]$ after running the program. The PARDO code is shown in figure 3.

The access patterns partly depend on the structure of the input graph and partly are strides with $s = 1$.

3.2 Experiments

All experiments were carried out for $m = 2^{22}$, the prime P was chosen closest to m. We simulated machines with $n = 2^t$, $t = 5, \ldots, 10$ processors and 5 jobs per processor. This serves to hide the network latency from processes. More exactly, the number c of processes per processor is proportional to $\log n$. We chose a fixed c to obtain comparable results. The value $c = 5$ was taken as an average from a machine size of $n = 128$ [2]. Therefore in each step $5n$ requests are made. Step in this context means synchronous execution of one instruction on each process.

As polynomials we used functions of degree $\xi = 2, 10, 20$. Thus we used 6 different classes of hash functions on 7 workloads and for 6 machine sizes. Each of the $6 \cdot 7 \cdot 6 = 252$ experiments was done 5 times with randomly chosen hash functions. More exactly, for each class five functions were randomly chosen and then used for all workloads and machine sizes.

For the synthetic traces, 4 steps were simulated. In the workloads with strides, the base b was increased each step by $5ns$.

As input for list ranking a list of length $10n$ was randomly chosen. As input for connected components, a graph with $10n$ nodes and $5n$ edges was randomly chosen. The problem size is twice as large as the number of processes in these applications. Each process simulates two program processors step by step. A problem size larger than a machine size is needed to obtain access patterns depending on the list or graph.

In matrix multiplication, the dimensions of the matrices were changed as follows: For $n = 2^{2z}$, matrix B has dimension $5 \cdot 2^z \times 5 \cdot 2^z$, matrices A and C have dimension $2^z \times 5 \cdot 2^z$. For $n = 2^{2z+1}$, matrix B has dimension $5 \cdot 2^{z+1} \times 5 \cdot 2^{z+1}$, matrices A and C have dimension $2^z \times 5 \cdot 2^{z+1}$.

In each experiment we measured for each step of the trace the maximum module congestion c_{max} and then computed the expected value of all c_{max} averaged over all steps. The analysis is a kind of (expected) worst case analysis. Each expected value was checked for significance by looking at the variance. The five values obtained by using five functions of one class for each experiment were checked against significant differences. In case there were none, the average was taken. In case there were some, ten additional hash functions were chosen and the average was taken from these 15 values. Significant differences appeared only for stride $s = 13$, $n = 2^7, \ldots, 2^9$ in both linear functions and for stride $s = 32$, $n = 2^9, 2^{10}$ in the linear function modulo power of two.

```
for $u \in V$ pardo $F[u] := u$; od;
for $t := 1$ to $2 \log |V|$ do
   for $u \in V$ pardo $change[u] := 0$; od;
   starcheck;
   for all $(u, w)$ with $\{u, w\} \in E$ pardo
      if $star[u]$ and $F[w] < F[u]$ then
         $F[F[u]] := F[w]$;
         $change[F[u]] := 1$;
         $change[F[w]] := 1$;
      fi;
   od;
   starcheck;
   for all $(u, w)$ with $\{u, w\} \in E$ pardo
      if $star[u]$ and not $change[F[u]]$
                  and $F[w] \neq F[u]$ then
         $F[F[u]] := F[w]$;
      fi;
      $F[u] := F[F[u]]$
   od
od.

proc starcheck ;
begin
   for $i \in V$ pardo
      $star[i] :=$ true;
      if $F[F[i]] \neq F[i]$ then
         $star[F[F[i]]] :=$ false;
      fi;
      $star[i] := star[F[F[i]]]$
   od
end;
```

Figure 3. Code for connected components

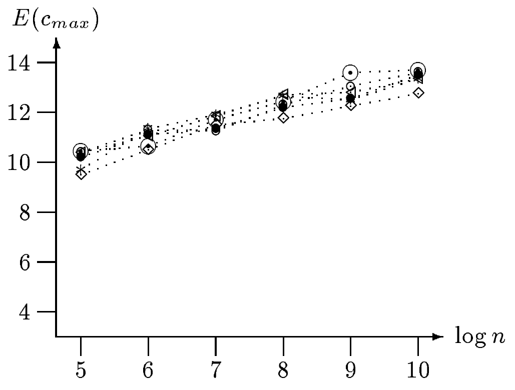

FIGURE 4. Performance on random patterns

Because of mapping $5n$ requests per step onto n memory modules, $E(c_{max}) \geq 5$. The only exception is connected components, because not necessarily all processors make accesses in IF statements.

3.3 Results

The results of the experiments are presented in two ways. First we show the performance of the hash functions sorted by benchmarks. In figure 4 the performance on random patterns is given as a reference. The legend of the hash functions is shown in figure 5. Figure 5 shows all other benchmarks. Second we show the performance sorted by hash functions in figure 6.

All figures are built as follows: the x–axis shows $\log n$ in range $5 \dots 10$, the y–axis shows the expected value of the maximum module congestions in range $4 \dots 14$.

The performance on random patterns (see figure 4) is similar for all hash functions. Thus none of the hash functions is bad in an obvious way. The maximum module congestion rises from 10 for $n = 32$ to 12 for $n = 1024$. This will serve as a reference to analyse the performance on the other benchmarks.

3.3.1 Analysis of benchmarks

The curves of figure 5 show similar shapes for all benchmarks: the polynomials of different degrees behave in a similar way and so do the three other hash functions. The behaviour of the polynomials furthermore is on all workloads worse than the behaviour of the simpler hash functions. Among the linear functions, the one modulo a prime always behaves a little bit worse than the linear function modulo a power of two. Thus the most interesting part is the comparison of our simple linear function with the boolean matrix hashing.

For strides that are a power of two, the boolean matrix hashes values optimally (see (a) and (c)) and reaches a module congestion of 6. The module congestion reached by the linear function lies between 6.5 and 7.5, so it is not far away.

A similar behaviour of linear function and boolean matrix can be seen in (d) and (f). This results from the fact that part of the accesses in these workloads are strides 1, when processors load or store values in arrays in the manner that processor i reads or writes $F[i]$.

However, as soon as we obtain other patterns, the boolean matrix hashing gets worse than the linear function (see (b) and (e)). Even for the matrix multiplication workload, where accesses always consist of $5 \cdot n^{1/2}$ strides with $s = 1$ and $n^{1/2}$ processors involved in each stride, the linear function is better.

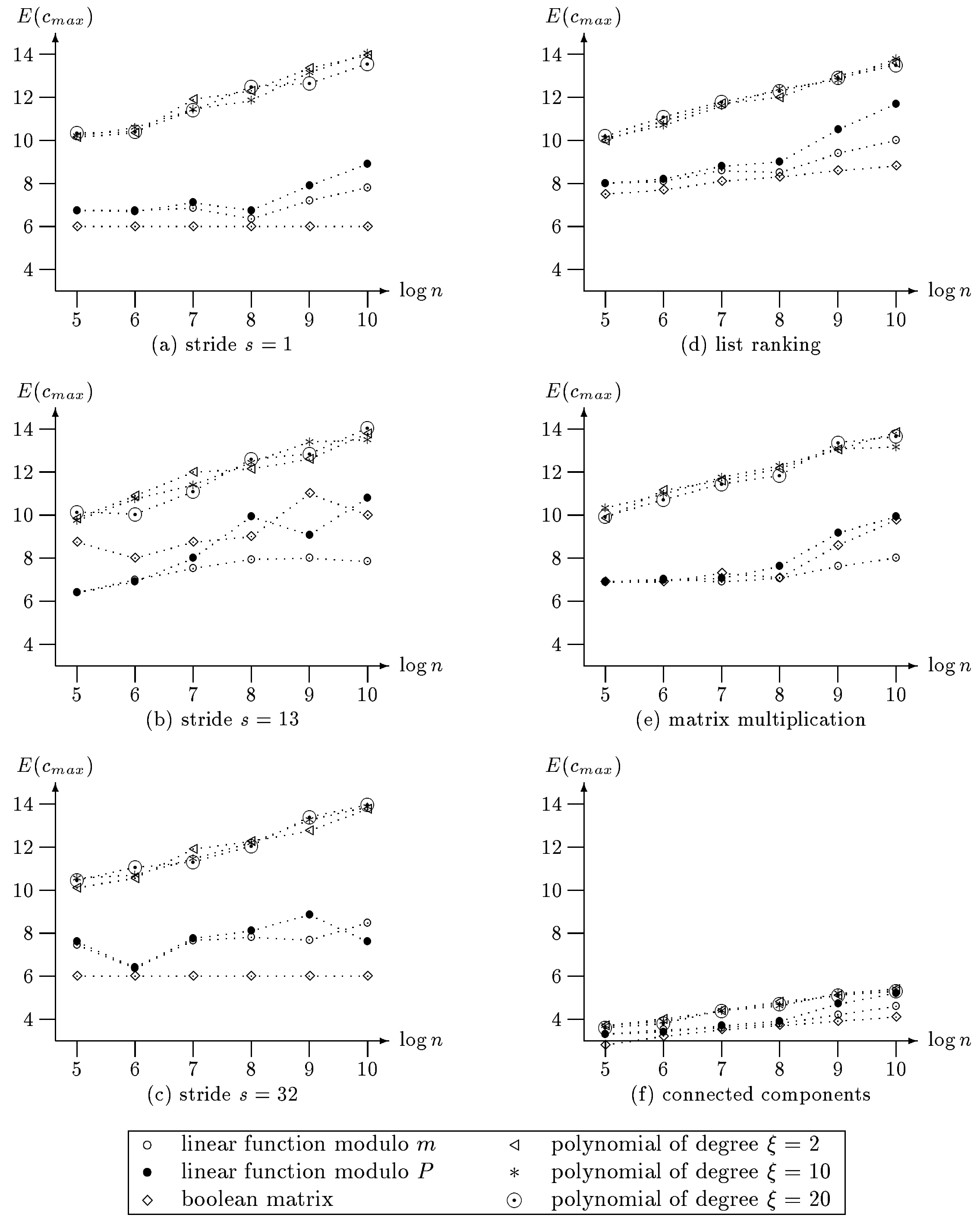

FIGURE 5. Performance on benchmarks

3.3.2 Analysis of hash functions

Figure 6 shows the performance of the different hash functions. Because the connected components benchmark is not comparable to the others, it is not shown here. The first observation is, that all hash functions behave on all workloads not worse than on random patterns. The second observation is that all polynomials show roughly the same behaviour on all workloads as they do on random patterns (see (d) to (f)). We conclude that their performance is independent of the application. That is what we expected. But this performance is bad in comparison to what is reached by the other functions that behave better than on random patterns on all workloads.

The linear function (see (a)) shows almost uniform behaviour on all workloads, too, but it varies between 6.5 and 8, which is significantly better than the behaviour on random patterns.

The behaviour of the linear function modulo a prime is not uniform and varies between 6.3 and 10.

The behaviour of the boolean matrix hashing function can be divided in an expected optimal behaviour for strides with s a power of two and a significantly higher module congestion for other patterns, which is however still below the one produced by random patterns.

3.3.3 Conclusions

The above experiments show surprisingly that linear functions modulo a power of two and boolean matrix functions show best performance for practical use. Both have the additional properties of bijectivity and short evaluation time. The choice between these two depends on the expected user profile (if such exists) and the surrounding machine architecture. For machines that already contain hardware multipliers this could be used to perform hashing in the case of the linear functions. Moreover, the use of matrix hashing is restricted by the fact that an implementation needs $(\log m)^2$ bit register hardware to store the boolean matrix. Therefore, if no user profile is known and chip area is restricted (or a multiplier already available), the use of the linear function is better.

4 Rehashing

The use of classes of hash functions where each function is "'bad"' only for a few patterns has the advantage that the probability of choosing a function that is bad for current program is very small. However if this case appears a bad pattern tends to happen multiple times because the patterns of a specific program are not randomly distributed. Then it could be better to choose a new hash function, redistribute the address space and then continue. This is called *rehashing*.

Rehashing in the case of linear functions consists of choosing a new factor a' and redistributing the address space using the new hash function $h'(x) = a' \cdot x \bmod m$. If the address space has size m and n processors are involved, redistribution should take time $O(m/n)$.

If each processor has a local harddisk of size m/n this works as follows: processor $P_i, 0 \leq i < n$ loads the contents of addresses $i \cdot m/n, \ldots, (i+1) \cdot m/n - 1$ using the old hash function and stores the contents on its local hard disk. Then it writes the contents these addresses back to the global memory using the new hash function. However the constant factor in the time $O(m/n)$ is quite large because hard disks are very slow compared to processor speed. The use of a *shade memory* where each processor owns an additional local memory of size m/n reduces this constant factor significantly but also increases the cost of the machine unnecessarily.

Therefore one should try to rehash without using secondary storage. The obvious solution is to load the content of an address x using $h(x)$. Before one can store this content in its new cell $h'(x)$ one has to save the content of this cell which is the content of address $x' = h^{-1}(h'(x))$. After saving x' in a temporary location (e.g. a processor register) and storing x to its new cell, we continue with x' as we did with x. This can be done until all cells have been moved. The method looks inherently sequential but it turns out that it can be parallelized.

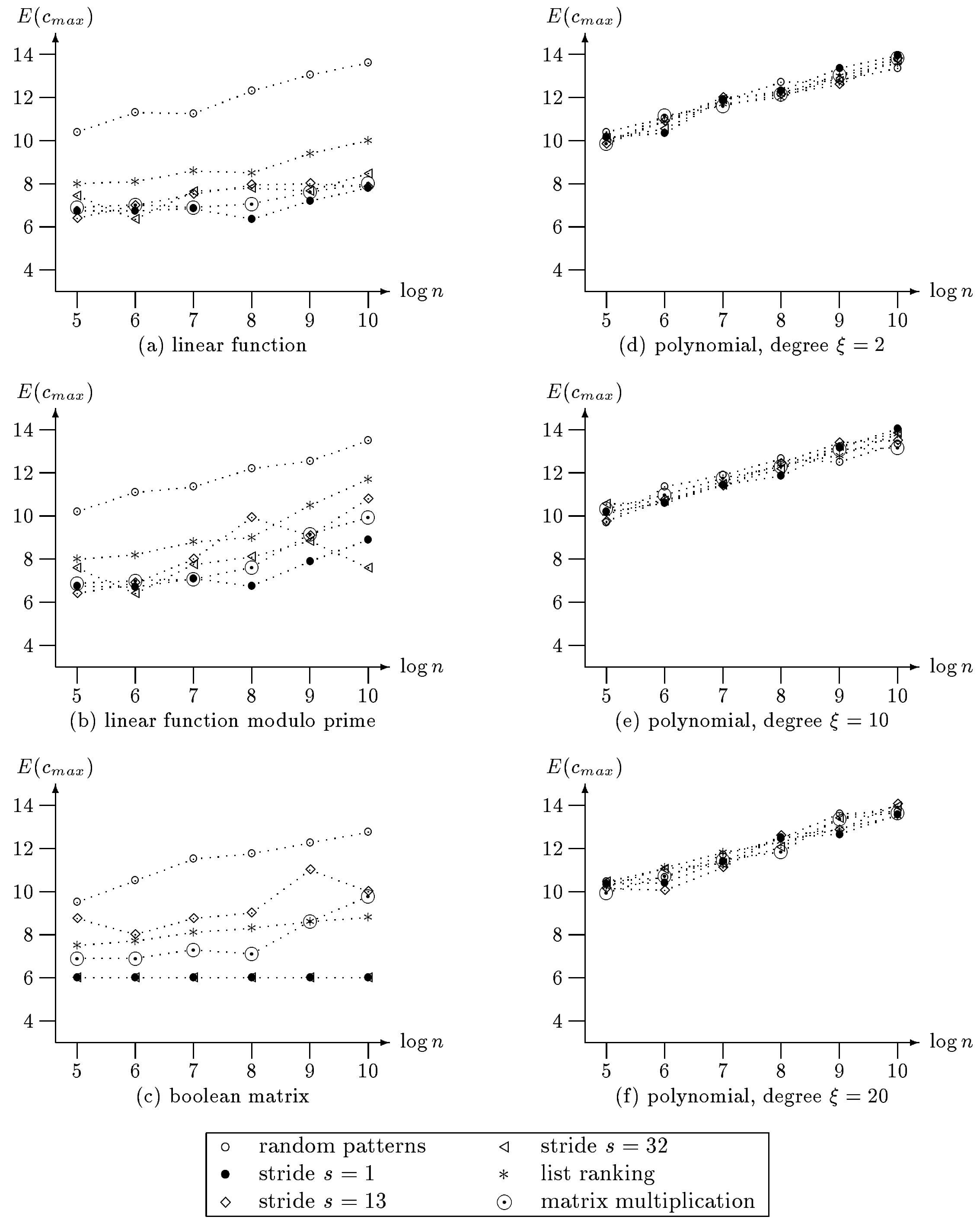

FIGURE 6. Performance of hash functions

4.1 Finding Cycles

We consider the algebraic structure of the address space $M = \{0, \ldots, 2^u - 1\}$. Together with addition and multiplication modulo 2^u it has the structure of a ring. The set $U = \{1, 3, \ldots, 2^u - 1\}$ of odd numbers is the set of invertible elements. Therefore a function $h : M \to M, h(x) = a \cdot x \bmod 2^u$ is bijective iff. a is odd. The inverse function to h is $h^{-1} = a^{-1} \cdot x \bmod 2^u$.

An address x is mapped onto cell $h(x)$ using the old hash function and onto cell $h'(x)$ using the new hash function. However when using the old hash function, cell $h'(x)$ still contains address $h^{-1}(h'(x))$. The addresses then are moved by a function $move(x) = h^{-1}(h'(x))$ which has the form $move(x) = b \cdot x \bmod m$ with $b = a^{-1} \cdot a' \bmod 2^u$. Because a and a' are odd and $a \neq a'$, b is odd, too, and not equal to 1.

a^{-1} can be computed using the euclidian algorithm for the greatest common divisor of a and 2^u.

However we must avoid to compute b with the help of a^{-1} because the runtime of the euclidian alogrithm can be very long. We overcome this by randomly choosing an odd integer $b \neq 1$ and computing $a' = a \cdot b \bmod 2^u$.

We now want to explore the structure in the movement by *move* to obtain parallelism. If we start with an address $c = 2^j \cdot c'$ where $j \geq 0$ is an integer and c' is odd then c is moved to $move(c) = c \cdot b \bmod 2^u$, this address is moved to $move(move(c)) = c \cdot b^2 \bmod 2^u$ and so on. After moving a certain number l of addresses we reach c again. We call the sequence $S_{c,b} = \{c, cb, \ldots, cb^{l-1}\}$ a *cycle*. It holds

$$c \equiv c \cdot b^l \bmod 2^u$$

With the above representation of $c = 2^j \cdot c'$ we obtain $1 \equiv b^l \bmod 2^{u-j}$. $U_j = \{1, 3, \ldots, 2^{u-j} - 1\}$ together with multiplication modulo 2^{u-j} forms a group. Then l is the order l_j of element $b_j = b \bmod 2^{u-j}$ in group U_j. As the order of the group which is 2^{u-j-1} is a multiple of the order of an element, l_j must be a power of two. The order of b_0 can be obtained in time $O(\log l_0) = O(u) = O(\log m)$ by successively computing $b_0^{2^{i+1}} = b_0^{2^i} \cdot b_0^{2^i} \bmod 2^u$, $i = 0, 1, \ldots$ until the result equals 1. then $l_0 = 2^{i+1}$.

The orders of elements $b_1, \ldots, b_{u-1}$ can be obtained in time $O(\log l_0) = O(\log m)$ as well. We use the fact that $\mathrm{ord}(b_j) \leq \mathrm{ord}(b_{j-1})$ for $j > 0$. While computing the b^{2^i}, we check whether $b^{2^i} \bmod 2^{u-j} \equiv 1$. In the beginning, $j = u - 1$. If the condition is fulfilled, we increase j until it is zero.

Let $B_j = \{b_j^q | 0 \leq q < l_j\}$.

As the group U_j is not cyclic if it contains more than two elements, there exist $n_j = 2^{u-j-1}/l_j$ cycles that together form U_j. In order to perform all movements by *move* we need an "entry" element of each cycle in M. This can be obtained by first computing an element c' of each cycle in U_j, $0 \leq j < u$. In order to obtain an element c of the original cycle in M we only have to compute $c = 2^j \cdot c'$ which takes constant time supposed the instruction set supports shifts.

4.2 Entry elements of cycles

To compute entry elements of all cycles in U_j means to find a set $C_j = \{c_{j,0}, \ldots, c_{j,n_j-1}\}$ such that with $C_{j,k} = c_{j,k} \cdot B_j$ the following properties are fulfilled:

$$C_{j,k} \cap C_{j,k'} = \emptyset \text{ for } k \neq k' \tag{1}$$

$$\cup_{k=0}^{n_j-1} C_{j,k} = U_j \tag{2}$$

U_j is generated by $\mu = -1$ and $\nu = 5$ [18]. Each $\alpha \in U_j$ thus can be represented as $\alpha \equiv \mu^{\alpha'} \cdot \nu^{\alpha''} \bmod 2^{u-j}$ where α' and α'' are unique modulo 2 and 2^{u-j-2}, respectively. Let $b_j = \mu^{\beta'} \cdot \nu^{\beta''}$. If $\beta'' \not\equiv 0 \bmod 2^{u-j-2}$, we choose

$$c_{j,k} = \begin{cases} \nu^k & : \quad 0 \leq k < n_j/2 \\ \mu \cdot \nu^k & : \quad n_j/2 \leq k < n_j \end{cases}$$

If $\beta'' \equiv 0 \bmod 2^{u-j-2}$, we choose $c_{j,k} = \nu^k, 0 \leq k < n_j$.

In order to prove that C_j fulfils (1), it is sufficient to show that for any $0 \leq k < n_j/2$, there does no $k' > 0$ exist such that $c_{j,k} \equiv b^{k'} \bmod 2^{u-j}$. This is evident for $\beta'' = 0$. If $\beta'' \neq 0$, assume that such a k' exists. Then $\nu^k = c_{j,k} = b^{k'} = \mu^{k'\cdot\beta'} \cdot \nu^{k'\cdot\beta''} \bmod 2^{u-j}$ and therefore $k \equiv k' \cdot \beta'' \bmod 2^{u-j-2}$. But β'' is a multiple of $n_j/2$, because $1 = b_j^{l_j} = \mu^{l_j\cdot\beta'} \cdot \nu^{l_j\cdot\beta''}$ and thus $l_j \cdot \beta'' \equiv 0 \bmod 2^{u-j-2}$. As $k < n_j/2$ this leads to a contradiction and therefore property (1) is fulfilled.

Property (2) follows directly from property (1) and the fact that $|C_{j,k}| = l_j$ for each k.

$$| \cup_{k=0}^{n_j-1} C_{j,k}| = \cup_{k=0}^{n_j-1}|C_{j,k}| = n_j \cdot l_j = |U_j|$$

As all sets are finite and each $C_{j,k}$ is a subset of U_j, the union of all $C_{j,k}$ must form U_j.

If $n_j/2 \leq n$ the $c_{j,k}$ can be computed with the help of parallel prefix in time $O(\log n)$. Otherwise we first compute $\tilde{b} = b^{n_j/(2n)}$ by successsively computing $b^{2^{i+1}} = b^{2^i} \cdot b^{2^i}$. This takes time $O(\log(m/n))$. With parallel prefix we compute $\tilde{b}^i, 0 \leq i < n$, which takes time $O(\log n)$. Last, each processor i computes sequentially $\tilde{b}^i, \tilde{b}^i \cdot b, \ldots, \tilde{b}^{i+1}$. This takes time $O(n_j/n)$. Therefore, the total time to compute all C_j is $O(\sum_j n_j/n) = O(m/n)$ if we assume that $m \geq n(\log n)^2$..

4.3 Scheduling Cycles

We now know that moving cells with function *move* happens in n_j cycles each of length l_j for $j = 0, \ldots, u-1$. For each cycle we know a starting point $c_{j,k}$. We furthermore know that all n_j's and l_j's are powers of two and that $n_j \cdot l_j = 2^{u-j-1}$ for all j.

The basic idea is to perform the movement of the cells in one cycle sequentially and to schedule the cycles among the processors. This is very easy if $n_j \geq n$. Both are powers of two, so n_j/n is an integer and each processor executes n_j/n cycles for each j. The distribution of cycles to processors can be done during the generation of the C_j and needs no extra time.

If however one of the n_j is smaller than n but still $n_j \cdot l_j \geq n$ which means there are not less than n elements in these cycles to be moved, we split each of these n_j cycles in n/n_j pieces to obtain a total of n pieces. As both numbers are powers of two, $n/n_j = 2^q \leq 2^t$. Therefore $q = O(\log n)$. Each processor then has to execute one piece of length $l_j/2^q$. Because of $2^q = n/n_j$ it holds $l_j/2^q = l_j \cdot n_j/n = 2^{u-j-1}/2^t = 2^r$ with $r = u-j-t-1 \leq \log m - \log n$.

Let the cycle to be split be $\{1, b, b^2, \ldots, b^{l_j-1}\}$ (the procedure is similar for the other cycles). The elements to start the pieces can be chosen to be

$$1, b^{2^r}, \left(b^{2^r}\right)^2, \ldots, \left(b^{2^r}\right)^{2^q-1}$$

b^{2^r} can be computed by successively multiplying $b^{2^{i+1}} = b^{2^i} \cdot b^{2^i} \bmod 2^{u-j}$ for $i = 0, \ldots, r-1$. This needs time $O(r) = O(\log m)$. All of the above elements can be computed by a parallel prefix operation in time $O(q) = O(\log n)$ with $2^q \leq n$ processors. The extra time for these j is at most $O(\log m \log n)$, since computing the pieces on a cycle has to be done only once per j and there are $\log m$ different j.

The case $n_j \cdot l_j < n$ happens for $j = u - \log n, \ldots, u-1$. Furthermore it holds $\sum_{j=u-\log n}^{u-1} n_j \cdot l_j = n-1$. We split these cycles completely. This can be done by parallel prefix which for all these j together needs n processors and time $O(\log n)$. Then each processor has to execute one move out of these cycles.

4.4 Runtime and Space Analysis

The total time for redistribution of global memory then is $O(m/n)$ because all n processors have the same amount of work and a total of m moves has to be done.

The total time for computing the schedule consists of:

Operation	Time
Finding ord(b) in all U_j	$O(\log m)$
Computing all C_j	$O(m/n)$
Scheduling cycles	$O(\log m \log n)$

We consider m to be at most polynomial in n and therefore $\log m = O(\log n)$. If furthermore $m \geq n \cdot (\log n)^2$, the total time to compute the schedule is $O(m/n)$.

The storage requirements are as follows:

The global data consist only of the n_j, l_j and intermediate results in parallel prefix computations. Therefore only $O(n + \log m)$ cells in global memory are needed.

The sets C_j can be evenly distributed among the processors (each $c_{j,k}$ is stored locally to the processor that creates it). Therefore each processor has to store $\sum_j n_j/n$ of them.

During redistribution, scheduling information of the form (loop entry, length) is needed. Each processor can maintain this information in its local memory. The amount of storage of scheduling information per processor is

$$s_l = 1 + \sum_{j=0}^{u-\log n-1} \lceil n_j/n \rceil$$

As $n_j = 2^{u-j-1}/l_j$, it holds

$$s_l = 1 + \sum_{j=0}^{u-\log n-1} \lceil 2^{u-j-1}/(l_j n) \rceil$$

Furthermore, either $l_{j-1} = l_j$ or $l_{j-1} = l_j/2$. Therefore s_l can be bounded by

$$1 + \sum_{j=0}^{u-\log n-1} \max\left(\frac{2^{j-1}}{l_0}, 2\right) \cdot \frac{2^{u-j-1}}{n}$$

In order to keep s_l small, one needs a large l_0. This is given with high probability by the following statement:

l_0 equals 2^{u-2-x} with probability 2^{-x-1}.

Therefore, after choosing b randomly, we compute l_0 and s_l in time $O(\log m)$. If s_l is too large, we choose a different value for b. The probability that after choosing y times all obtained orders are less than 2^{u-2-x} is only $\left(2^{-1-x}\right)^y$. Because each try takes only time $O(\log m)$ we can afford to have $O(m/(n \log m))$ tries without affecting the time bound of $O(m/n)$. Therefore, even if we choose a small x, the probability of finding an element that has the required order is extremely high.

Proof of statement: Group $(U_0, \cdot \bmod 2^{u-j})$ is isomorph to $(\{0,1\}, + \bmod 2) \times (\{0, \ldots, 2^{u-2} - 1\}, + \bmod 2^{u-2}) = U' \times U''$ by an isomorphism ψ [18]. The order of an element $b \in U_0$ with $\psi(b) = (b_1, b_2)$ is equal to the order of $b_2 \in U''$. U'' is cyclic and therefore the number of elements with order 2^{u-2-x} equals $\phi(2^{u-2-x})$ (the Euler function) [18]. Therefore for a random chosen element $b_2 \in U''$ the probability of $\text{ord}(b_2) = 2^{u-2-x}$ is $\phi(2^{u-2-x})/|U''| = 2^{-x-1}$. ■

5 Conclusions

Linear hash functions have turned out to have many desirable properties: bijectivity, short evaluation time, low module congestion. In addition we could show that, if necessary, rehashing can be done in optimal time without using secondary storage. Therefore these functions seem to be an ideal choice for architectures with hardware support for emulation of PRAMs. For these reasons they are used in a prototype implementation [1].

It still is an open question whether rehashing of linear functions can be done online, which means that rehashing can be done without stopping an application program but executing $O(1)$ steps of the application program and $O(1)$ steps of the rehashing algorithm alternately for the time span of $O(m/n)$.

Acknowledgements

The author would like to thank Martin Dietzfelbinger for helpful discussions on hash functions, Stefan Ellwert and Volker Müller for providing some help in algebraic notations, and Dany Breslauer for suggestions about the computation of entry elements.

References

[1] Ferri Abolhassan, Reinhard Drefenstedt, Jörg Keller, Wolfgang J. Paul, and Dieter Scheerer. On the physical design of PRAMs. In Johannes Buchmann, Harald Ganzinger, and Wolfgang J. Paul, editors, *Informatik — Festschrift zum 60. Geburtstag von Günter Hotz*, pages 1–19. Teubner Verlag, 1992.

[2] Ferri Abolhassan, Jörg Keller, and Wolfgang J. Paul. On the cost–effectiveness of PRAMs. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 2–9. IEEE, December 1991.

[3] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The tera computer system. In *Proceedings of the Supercomputing'90, Computer Architecture News*, pages 1–6. ACM, 1990.

[4] J.L. Carter and M.N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[5] Martin Dietzfelbinger. Hashing modulo powers of two. Manuscript, Universität–GH Paderborn, October 1990.

[6] Martin Dietzfelbinger. On limitations of the performance of universal hashing with linear functions. Reihe Informatik Bericht Nr. 84, Universität–GH Paderborn, June 1991.

[7] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the 10th ACM Annual Symposium on Theory of Computing*, pages 114–118, 1978.

[8] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[9] R.W. Hockney and C.R. Jesshope. *Parallel Computers 2.* Adam Hilger, Bristol and Philadelphia, 1988.

[10] Anna R. Karlin and Eli Upfal. Parallel hashing: An efficient implementation of shared memory. *Journal of the ACM*, 35(4):876–892, October 1988.

[11] Richard M. Karp and Viaya L. Ramachandran. A survey of parallel algorithms for shared–memory machines. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A*, pages 869–941. Elsevier, 1990.

[12] David May. The next generation transputers and beyond. In A. Bode, editor, *Distributed Memory Computing. 2nd European Conference, EDMCC2, Proceedings*, pages 7–22. Springer, 1991.

[13] Kurt Mehlhorn and Uzi Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.

[14] Alan Norton and Evelyn Melton. A class of boolean linear transformations for conflict–free power–of–two stride access. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 247–254, 1987.

[15] 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 *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771. IEEE, 1985.

[16] Abhiram G. Ranade. How to emulate shared memory. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 185–194, 1987.

[17] Abhiram G. Ranade, Sandeep N. Bhatt, and S. Lennart Johnson. The Fluent Abstract Machine. In *Proceedings of the 5th MIT Conference on Advanced Research in VLSI*, pages 71–93, 1988.

[18] Hans-Jörg Reiffen, Günter Scheja, and Udo Vetter. *Algebra.* B.I.–Wissenschaftsverlag, 2nd edition, 1984.

[19] Y. Shiloach and Uzi Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[20] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[21] Leslie G. Valiant. General purpose parallel architectures. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A*, pages 943–971. Elsevier, 1990.