An O(1) Solution to the Prefix Sum Problem on a Specialized Memory Architecture

Andrej Brodnik[1,2], Johan Karlsson[1], J. Ian Munro[3], and Andreas Nilsson[1]

[1] Luleå University of Technology
Dept. of Computer Science and Electrical Engineering
S-971 87 Luleå
Sweden
{johan.karlsson,andreas.nilsson}@csee.ltu.se
[2] University of Primorska
Faculty of Education
Cankarjeva 5
6000 Koper
Slovenia
andrej.brodnik@pef.upr.si
[3] Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario
Canada, N2L 3G1
imunro@uwaterloo.ca

Abstract. In this paper we study the Prefix Sum problem introduced by Fredman. We show that it is possible to perform both update and retrieval in $O(1)$ time simultaneously under a memory model in which individual bits may be shared by several words. We also show that two variants (generalizations) of the problem can be solved optimally in $\Theta(\lg N)$ time under the comparison based model of computation.

1 Introduction

Models of computation play a fundamental role in theoretical Computer Science, and indeed, in the subject as a whole. Even in modeling a standard computer, the random access machine (RAM) model has been subject to refinements which more realistically model cost or, as in this paper, suggest feasible extensions to the model that permit more efficient computation, at least for some problems. Work taking into account a memory hierarchy, either when memory and page sizes are known (cf. [2]) or not (cf. [11]) is an example of the former. Taking into account parallelism, as in the PRAM model (cf. [17,26]), is an obvious example of the latter. More subtle examples include the recent result that the operations of an arbitrary finite Abelian group can be carried out in constant time (We assume a word of memory is adequate to hold the size of the group.) provided one can reverse the bits of a word in constant time [8]. This argues for a more robust set of operations. Here we deal with the way a single level memory is

organized and demonstrate that the power of a machine can be increased if we permit individual bits to occur in several words simultaneously. This Random Access Machine with Byte Overlap (RAMBO) was first suggested by Fredman and Saks [10] and subsequently used by Brodnik et al. [6] and Brodnik and Iacono [7]. Indeed it is shown in the latter two papers that a priority queue of word sized objects can be maintained in constant time under a particular form of the RAMBO model, whereas Beame and Fich [3] and Brodnik and Iacono [7] have both shown lower bounds on the problem under various forms of the RAM model.

Here we discuss solutions to variants of the *Prefix Sum problem* (i.e. finding the sum of the first j elements in an array and also updating these values) which was introduced by Fredman [9]. Various lower bounds have been proven for the problem. We, however, focus on the problem under a nonstandard, though very feasible, model to achieve a constant time solution.

Fredman and Saks actually suggested the RAMBO model in connection with the Prefix Sum problem. They claim, with no hint of how it may be done, that Prefix Sum mod 2 can be solved in constant time under the model. We show how this can be done not only for Prefix Sum mod 2 but for Prefix Sum modulo an arbitrary universe size $M \leq 2^{\Theta(b/n)}$ where b is the word size, $n = \lceil \lg N \rceil$ and N is the size of the array.

The RAMBO model, besides the usual RAM operations (cf. [27]), also has a part of memory where a bit may occur in several registers or in several positions in one register. The way the bits occur in this part of the memory has to be specified as part of the model. One example of such a memory variant is a square of bits with b rows and b columns. A b-bit word can be fetched either as a row or a column. In such a memory each bit can be accessed either by the row word or the column word.

The form of RAMBO used by Brodnik et al. [6] to solve the priority queue problem in O(1) worst case time makes use of words corresponding to the leaves of a balanced binary tree. Each node of the tree contains a flag bit and each such word contains the flags along the root to leaf path, so, for example, the flag at the root is in all of these words. The specific architecture was called *Yggdrasil* after the giant ash tree linking the worlds in Norse mythology. That variant has been implemented in hardware [18] and the actual rerouting of the bits on a word fetch is not difficult. In this paper we modify the Yggdrasil variant slightly and solve the Prefix Sum problem. This gives further evidence of the value of such an architecture, at least for a special purpose processor.

Now let us formally define the Prefix Sum problem:

Definition 1 *The* Prefix Sum problem *is to maintain an array, $\mathcal{A}$, of size N, and to support the following operations:*
`Update(`j`,` Δ`)` $\mathcal{A}(j) := \mathcal{A}(j) + \Delta$
`Retrieve(`j`)` *return* $\sum_{i=0}^{j} \mathcal{A}(i)$
where $0 \leq j < N$.

Fredman showed that, under the comparison based model of computation, an $O(\lg N)$ solution exists for the Prefix Sum problem [9].

The problem can be generalized in several ways and we start by adding another parameter, k to the `Retrieve` operation. This parameter is used to tell the starting point of the array interval to sum over. Hence, `Retrieve(k,j)` returns $\sum_{i=k}^{j} \mathcal{A}(i)$, where $0 \leq k \leq j < N$. This variant is usually referred to as the *Partial Sum* or *Range Sum problem*. The Partial Sum problem can be solved using a solution to the Prefix Sum problem (`Retrieve(k,j) = Retrieve(j) - Retrieve(k-1)`). In fact, the two problems are often used interchangeably.

Furthermore, there is no obvious reason to only allow addition in the `Update` and `Retrieve` operations. We can allow any binary function, $\oplus$, to be used. In fact we can allow the `Update` operation to use one function, $\oplus_u$, and the `Retrieve` operation to use another function, $\oplus_r$. We will refer to this variant of the problem as the *General Prefix Sum problem*.

Moreover, one can allow array position to be inserted at or deleted from arbitrary places. Hence, we can have sparse arrays, e.g. an array where only $\mathcal{A}(5)$ and $\mathcal{A}(500)$ are present. Positions which have not yet been added or have been deleted have the value 0. We refer to this variant as the *Dynamic Prefix Sum problem*. Brodnik and Nilsson [21, pp 65-80] describe a data structure they call a BinSeT tree which can be modified slightly to support all operation of the Dynamic Prefix Sum problem in $O(\lg N)$ time.

The *Searchable Partial Sum* problem extends the set of operations with a `select(j)` operation which finds the smallest i such that $\sum_{k=0}^{i} \mathcal{A}(k) \geq j$ [23]. Hon et al. consider the Dynamic version of the Searchable Partial Sum problem [16]. Another generalization is to use multidimensional arrays and this variant has been studied by the data base community [4, 12, 13, 15, 24, 25].

Several lower bounds have been presented for the Prefix Sum problem: Fredman showed a $\Omega(\lg N)$ algebraic complexity lower bound and a $\Omega(\lg N / \lg \lg N)$ information-theoretic lower bound [9]. Yao [29] has shown that $\Omega(\lg N / \lg \lg N)$ is an inherent lower bound under the semi-group model of computation and this was improved by Hampapuram and Fredman to $\Omega(\lg N)$ [14]. We side step these lower bounds by considering the RAMBO model of computation [5, 10].

As with all RAM based model we need to restrict the size of a word which can be stored and operated on. We denote the word size with b and assume that b is an integer power of 2 which is true for most computers today. A bounded word size also implies a bounded universe of elements that we store in the array. We use M to denote the universe size. Hence all operations $\oplus$ have to be computed modulo M and we require that each of the operands and the result are stored in one word.

We will use n and m to denote $\lceil \lg N \rceil$ and $\lceil \lg M \rceil$ respectively. Hence, $N \leq 2^n$ and $M \leq 2^m$. Both n and m are less than or equal to b, $(n, m \leq b)$. In one of the solutions we actually require that $nm \leq b$.

In Sect. 2 we show a $O(1)$ solution to the Prefix Sum problem under the RAMBO model using a modified Yggdrasil variant. In Sect. 3 we discuss a

$O(\lg N)$ solution to the General and Dynamic Prefix Sum problems and finally conclude the paper with some open questions in Sect. 4.

2 An $O(1)$ Solution to the Prefix Sum Problem

In our O(1) solution to the Prefix Sum problem we use a complete binary tree on top of the array (Fig. 1). We label the nodes in standard heap order, i.e., the root is node ν_1 and the left and right children of a node ν_i are ν_{2i} and ν_{2i+1} respectively. In each node we store m bits representing the sum of the leaves in the left subtree. Since we build a complete binary tree on top of the array we assume that $N = 2^n$ (if this is not true we still build the complete tree and in worst case waste space proportional to $N/2 - 1$). We do not store the original array $\mathcal{A}$ since its values are stored implicitly in the tree. The only value not stored in the tree (if $N = 2^n$ only) is $\mathcal{A}(N-1)$ and we store this value explicitly (`vn1`). Formally we define:

Definition 2 *A* N-m-tree *is a complete binary tree with N leaves in which the internal nodes (ν_i) store a m-bit value. In addition, a m-bit value is stored separately (*`vn1`*).*

To update $\mathcal{A}(j)$ (Algorithm 1) in this structure we have to update all the nodes on the path from leaf j to the root in which j belongs to the left subtree. To `Retrieve(`j`)` (Algorithm 2) we need to sum the values of all the nodes on the path from leaf $j + 1$ to the root in which $j + 1$ belongs to right subtree. Note that the path corresponding to array position j starts at node $\nu_{N/2+j/2}$.

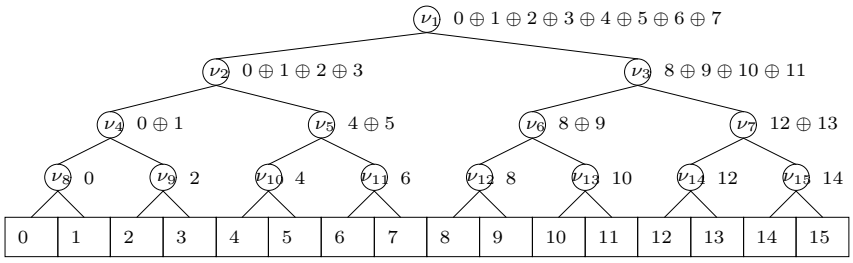

Fig. 1. Complete binary tree on top of $\mathcal{A}$. Nodes are storing the sum of the values in the leaves covered by the left subtree.

The method described above implies a $O(\lg N)$ update and retrieval time in the RAM model. To achieve constant time update and retrieval we use a variant of the RAMBO model similar to the Yggdrasil variant. In the Yggdrasil variant, registers overlap as paths from leaf to root in a complete binary tree with one bit stored in each internal node [6]. We generalize the Yggdrasil variant and let it store m bits in each node and call this variant *m-Yggdrasil*. In any

```
update(j, Δ)
  if (j == N-1)
    vn1 = vn1 + Δ;
  else
    i = N + j;
    while (i > 1)
      next = i div 2;
      if (i mod 2 == 0)
        ν_next = ν_next + Δ mod M);
      i = next;
```

Alg 1: Updating of a N-m-tree in $O(\lg N)$ time.

```
retrieve(j)
  if (j == N-1)
    sum = vn1;
    i = N + j;
  else
    sum = 0;
    i = N + j + 1;
  while (i > 1)
    next = i div 2;
    if (i mod 2 == 1)
      sum = sum + ν_next mod M;
    i = next;
  return sum;
```

Alg 2: Retrieve in a N-m-tree in $O(\lg N)$ time.

m-Yggdrasil, register `reg[i]` corresponds to the path from node $\nu_{N/2+i}$ to the root of the tree. Each register consists of $nm \leq b$ bits. In total the m-Yggdrasil registers need $(N-1) \cdot m$ bits.

Now, we use the registers from m-Yggdrasil to store the nodes of our tree. The path corresponding to array position j is stored in `reg[j/2]` and hence all nodes along the path can be accessed at once.

We let levels of the tree be counted from the internal nodes above the leaves starting at 0 and ending with $n-1$ at the root. If the ith bit of j is 1 then j is in the right subtree of the node on level i of the path and in the left otherwise. Hence j can be used to determine which nodes along the path should be updated (nodes corresponding to bits of j that are 0) and which nodes should be used when retrieving a sum (nodes corresponding to bits of j that are 1).

When updating the m-Yggdrasil registers (Algorithm 3), for all bits of j, if the ith bit of j is 0 we add Δ to the value of the ith node along the path from j to the root. To do this we shift Δ to the corresponding position ($\Delta << (im)$) and add to `reg[j/2]`. Instead of checking whether the ith bit of j is 0 we can

mask the shifted Δ with a value based on NOT j. The value consists of, if the ith bit of NOT j is 1, m 1s shifted to the correct position and m 0s otherwise.

```
update(j, Δ)
  if (j == N-1)
    vn1 = vn1 + Δ;
  else
    for (i=0; 0 < n; i++)
      if (((j >> i) AND 1) == 0)
        reg[j/2] = reg[j/2] + (Δ << (i*m));
```

Alg 3: Updating of a N-m-tree stored in m-Yggdrasil memory ($O(\lg N)$ time).

Actually, as long as the binary operation only affects the m bits that should be updated we can use word-size parallelism (cf. [5]) and perform the update of all nodes in parallel. In Sect. 2.1 we show that addition modulo M can be implemented affecting only m bits.

We use two functions (`dist(i)` and `mask(i)`) to simplify the description of the update and retrieve methods. The function `dist(i)`, $(0 \leq i < 2^m)$ computes nm-bit values. The values are n copies of the m bits in i. For example, given $m = 3, n = 4$ `dist(010)` is 010010010010. The function `mask(i)`, $(0 \leq i < 2^n)$ also computes nm-bit values. These values are computed as follow: bit j $(0 \leq j < n)$ of i is copied to bits $jm..(j+1)m-1$. For example, given $m = 3, n = 4$, `mask(1001)` is 111000000111. Both these functions can be implemented by using word-size parallelism [5].

We can update the tree in constant time using the procedure in Algorithm 4. First we make n copies of Δ and then mask out the copies we need. Then finally we add the value in `reg[j/2]` and the masked distributed Δ and store the result in `reg[j/2]`. For the case when $j = N - 1$ we simply add `vn1` and Δ and store it in `vn1`. This gives us the following lemma:

Lemma 1 *The update operation of the Prefix Sum problem can be supported in $O(1)$ when part of the N-m-tree is stored in a m-Yggdrasil memory.*

```
update(j, Δ)
  if (j == N-1)
    vn1 = vn1 + Δ;
  else
    reg[j/2] = reg[j/2] + (dist(Δ) AND mask(NOT j));
```

Alg 4: Updating of a N-m-tree stored in m-Yggdrasil memory using word size parallelism ($O(1)$ time).

To support the retrieve method in constant time we use a table `SUM[i]`, $(0 \leq i < 2^{nm})$ with m-bit values that are the sum modulo M of the n m-bit values in i.

To retrieve the sum (Algorithm 5) we read the register `reg` corresponding to j and mask out the parts we need. Then we use the table `SUM` to calculate the sum. Finally, we add `vn1` to the sum if $j = N - 1$.

```
retrieve(j)
  if (j == N-1)
    v = reg[j/2] AND mask(j);
  else
    v = reg[(j+1)/2] AND mask(j+1);
  sum = SUM[v];
  if (j == N-1)
    sum = vn1 + sum;
  return sum;
```

Alg 5: Retrieve in a N-m-tree stored in m-Yggdrasil memory using word size parallelism ($O(1)$ time).

The space needed by the table `SUM` is $2^{nm} \cdot m = N^{\lg M} \cdot m = M^{\lg N} \cdot m$, which is rather large. In order to reduce the space requirement we can reduce, by half, the number of bits used as index into the table. This gives us a space requirement of $\sqrt{M^{\lg N}} \cdot m$. We do this by shifting the top $n/2$ m-bit values from `reg` down and computing the sum modulo M of these values and the bottom $n/2$ values. Then this new $(n/2)m$-bit value is used as index into `SUM` instead.

We can actually repeat this process until we get the m-bit we desire, and hence we do not need the table `SUM` (Algorithm 6). However, this does increase the time complexity to $O(\lg n) = O(\lg \lg N)$. This gives us a trade off between space and time. By allowing $O(\iota)$ steps for the retrieve method we need $M^{\lg N/2^\iota} \cdot m$ bits for the table.

Lemma 2 *The retrieve operation of the Prefix Sum problem can be supported in $O(\iota + 1)$ time using $O(M^{\lg N/2^\iota} \cdot m + m)$ bits of memory in addition to the N-m-tree. Part of the N-m-tree is stored in m-Yggdrasil memory.*

By adjusting ι we can achieve the following result:

Corollary 1 *The retrieve operation of the Prefix Sum problem can be supported in:*

- $O(1)$ *time using* $O(M^{(\lceil \lg N \rceil)/2} \cdot m)$ *bits of memory in addition to the N-m-tree, with* $\iota = 1$.
- $O(\lg \lg N)$ *time using* $O(m)$ *bits of memory in addition to the N-m-tree, with* $\iota = \lceil \lg \lg N \rceil$.

```
retrieve(j)
  if (j == N-1)
    v = reg[j/2] AND mask(j);
  else
    v = reg[(j+1)/2] AND mask(j+1);
  ι = ⌈lg n⌉;
  do
    ι = ι-1;
    vnew = (v>>((2^ι)m)) + (v AND ((1<<((2^ι)m))-1));
    v = vnew;
  while (ι > 0)
  sum = v;
  if (j == N-1)
    sum = vn1 + sum;
  return sum;
```

Alg 6: Retrieve in a N-m-tree stored in m-Yggdrasil memory using no additional memory ($O(\lg \lg N)$ time).

2.1 Addition modulo M

Let us consider the two m-bit operands a and b which are split into two pieces each (a_{lo}, a_{hi}, b_{lo} and b_{hi}). The two pieces a_{lo} and a_{hi} contain the $m/2$ least and most significant bits of a respectively (similarly for b_{lo} and b_{hi}). Note that a_{lo} and the other pieces are stored in m-bit but only the $m/2$ least significant bits are used.

We can now add the the two operands

$$c1_{lo} = a_{lo} + b_{lo} \tag{1}$$
$$c1_{hi} = a_{hi} + b_{hi} \ . \tag{2}$$

However, both $c1_{lo}$ and $c1_{hi}$ might need $m/2+1$ bits for its result. The $m/2+1$ bit of $c1_{lo}$ should be added to $c1_{hi}$ and we split $c1_{lo}$ into two pieces ($c1_{lo,lo}$ and $c1_{lo,hi}$) and add the most significant bits to $c1_{hi}$,

$$c_{hi} = c_{hi} + c_{lo,hi} \tag{3}$$
$$c_{lo} = c_{lo,lo} \ . \tag{4}$$

The result of $a+b$ is now stored in c_{lo} and c_{hi} and we have not used more than m bits in any word. However, in total $m+1$ might be needed for the value.

To compute $c \bmod M$ we can check whether or not $c - M >= 0$, if so $c \bmod M = c - M$ and otherwise $c \bmod M = c$. However, we do not want to produce a negative value since that would affect all the bits in the word. Instead we add an additional 2^m to the value and compare to 2^m, i.e. $c + 2^m - M \geq 2^m$. Since $2^m - M \geq 0$ this will never produce a negative value. Note that $c + 2^m - M < M - 1 + M - 1 + 2^m - M = M + 2^m - 2 <= 2^{m+1} - 2$ which

only needs $m+1$ bits to be represented. Hence, if we calculate this value using the strategy above we will not use more than m bits of any word.

Furthermore, a straight forward less than comparison can not be performed using word-size parallelism since all bits of the words are considered. Instead we view the comparison as a check whether the $m+1$st bit is set or not. If it is set the value is larger than or equal to 2^m (cf. [19, 22]). We can actually create a bit mask which consists of m 1s if the $m+1$st bit is set and m 0s otherwise

$$d = (c + 2^m - M \text{ AND } 2^m) - ((c + 2^m - M \text{ AND } 2^m) >> m) \ . \tag{5}$$

This bit mask d can then be used to calculate $res = c \bmod M$. Since res is equal to $c - M$ if the $m+1$st bit of c is set and c otherwise we get

$$res = ((c - M) \text{ AND } d) \text{ OR } (c \text{ AND NOT } d) \ . \tag{6}$$

When computing $c - M$ we must make sure that we do not produce a negative value. This is done by using a similar strategy as for addition above, but we also set any of the bits in $c_{hi,hi}$ to 1 during the computation. If $c - M$ is greater than 0 this will not affect the result and otherwise the result will not be used.

We have a procedure which can be used to compute $(a+b) \bmod M$ without using more than m bits in any word. Hence, word-size parallelism can be used and we get our main result from this section:

Theorem 1 *Using the N-m-tree together with the m-Yggdrasil memory we can support the operations of the Prefix Sum problem in $O(\iota+1)$ time using $(N-1)m$ bits of m-Yggdrasil memory and $O(M^{n/2^\iota} \cdot m + m)$ bits of ordinary memory.*

3 An $O(\lg N)$ Solution to the General and Dynamic Prefix Sum Problem

We can actually partially solve the General Prefix Sum problem using the N-m-tree data structure and the m-Yggdrasil variant of RAMBO. All binary operations such that all elements in the universe have a unique inverse element (i.e. binary operations which form a *Group* with the set of elements in the universe) and only affect the m bits involved in the operation can be supported. This includes for example addition and subtraction but not the maximum function.

To solve the General and Dynamic Prefix Sum problem for semi-group operations we modify the Binary Segment Tree (BinSeT) data structure suggested by Brodnik and Nilsson. It was designed to handle in-advance resource reservation [21, pp 65-80] and if it is slightly modified it can solve both the General and Dynamic Prefix Sum problems efficiently. The original BinSeT stores, in each internal node, μ, the maximum value over the interval, and δ, the change of the value over the interval. Further, it also stores τ, the time of the left most event in the right subtree.

Instead of storing times as interval dividers we store array indices. To solve the Dynamic Prefix Sum problem with addition as operation and we only need

to store δ. When solving the General and Prefix Sum problem one need to store information depending on the two binary operations $\oplus_u$ and $\oplus_r$.

When adding a new array position or deleting an array position the tree is rebalanced (cf. [1, 20]) and hence the height is always $O(\lg N)$. When updating a value in an array position we start at the root and search for the proper leaf using the interval dividers. During the back tracking of the recursion we update the information stored in each affected node.

At retrieval we process the information of the proper nodes when traversing the tree. Since the height of the tree is $O(\lg N)$ all the operations can be performed in $O(\lg N)$ time. This matches the lower bound by Hampapuram and Fredman [14]

BinSeT consists of $O(N)$ nodes when we use it to solve the General Prefix Sum. Each node contains $O(1)$ m-bit values and hence the total space requirement is $O(Nm)$ bits.

4 Conclusion

The Dynamic and General Prefix Sum problems can both be solved optimally in $\Theta(\lg N)$ using $O(Nm)$ space under the comparison based model with semigroup operations.

The Prefix Sum problem can be solved in $O(1)$ time under the RAMBO model when we allow $O(\sqrt{M^{(\lceil \lg N \rceil)}} \cdot m)$ bits of ordinary memory and $O(Nm)$ bits of m-Yggdrasil memory to be used. This is a huge amount of ordinary memory and if we restrict the space requirement to be sub exponential in both N and M ($O(m)$ bits of ordinary memory and $O(Nm)$ bits of m-Yggdrasil memory) we need to used $O(\lg \lg N)$ time. We know of no better lower bound under RAMBO than the trivial $\Omega(1)$ when only allowing $O((N^{O(1)} + M^{O(1)})m)$ space.

Further, it is currently unknown if one can achieve a $O(1)$ solution to the Dynamic and General Prefix Sum problems using the RAMBO model. Another open question is whether or not it is possible achieve a $o(\lg N)$ solution to the multidimensional variant.

Acknowledgment

We thank the anonymous reviewers for helpful comments and additional references.

References

1. G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. In *Soviet Math. Doclady 3*, pages 1259–1263, 1962.

2. Alok Aggarwal and Ashok K. Chandra. Virtual memory algorithms (preliminary version). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 173–185. ACM Press, May 2–4 1988.
3. P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002.
4. Fredrik Bengtsson and Jingsen Chen. Space-efficient range-sum queries in OLAP. In Yahiko Kambayashi, Mukesh Mohania, and Wolfram Wöß, editors, *Data Warehousing and Knowledge Discovery: 6th International Conference DaWaK*, volume 3181 of *Lecture Notes in Computer Science*, pages 87–96. Springer, September 2004.
5. Andrej Brodnik. *Searching in Constant Time and Minimum Space (*Minimæ Res Magni Momenti Sunt*)*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1995. (Also published as technical report CS-95-41.).
6. Andrej Brodnik, Svante Carlsson, Michael L. Fredman, Johan Karlsson, and J. Ian Munro. Worst case constant time priority queue. *Journal of System and Software*, 78(3):249–256, December 2005.
7. Andrej Brodnik and John Iacono. Dynamic predecessor queries. Unpublished manuscript, 2006.
8. Arash Farzan and J. Ian Munro. Succinct representation of finite abelian groups. In *Proceedings of the 2006 International Symposium on Symbolic and Algebraic Computation*, Lecture Notes in Computer Science. Springer, 2006. To appear.
9. Michael L. Fredman. The complexity of maintaining an array and computing its partial sums. *Journal of the ACM*, 29(1):250–260, January 1982.
10. Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 345–354. ACM Press, May 14–17 1989.
11. Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In IEEE, editor, *40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 285–297. IEEE Computer Society, IEEE Computer Society, October 17–19 1999.
12. Steven P. Geffner, Divyakant Agrawal, Amr El Abbadi, and T. Smith. Relatve prefix sums: An efficient approach for querying dynamic OLAP data cubes. In *Proceedings of the 15th International Conference on Data Engineering*, pages 328–335, 1999.
13. Steven P. Geffner, Mirek Riedewald, Divyakant Agrawal, and Amr El Abbadi. Data cubes in dynamic environments. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, pages 31–40, 1999.
14. Haripriyan Hampapuram and Michael L. Fredman. Optimal biweighted binary trees and the complexity of maintaining partial sums. *SIAM Journal on Computing*, 28(1):1–9, 1998.
15. C. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 73–88, 1997.
16. Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Succinct data structure for searchable partial sums. In Toshihide Ibaraki, Naoki Katoh, and Hirotaka Ono, editors, *Algorithms and Computation – ISAAC 2003, 14th International Symposium*, volume 2906 of *Lecture Notes in Computer Science*, pages 505–516. Springer, December 2003.
17. Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memeory machines. In van Leeuwen [28], chapter 17, pages 869–941.

18. Roni Leben, Marijan Miletić, Marjan Špegel, Andrej Trost, Andrej Brodnik, and Johan Karlsson. Design of high performance memory module on PC100. In *Proceedings Electrotechnical and Computer Science Conference*, pages 75–78, Slovenia, 1999.
19. Kjell Lemström, Gonzalo Navarro, and Yoan Pinzon. Practical algorithms for transposition-invariant string-matching. *Journal of Discrete Algorithms*, 3(2–4):267–292, 2005.
20. Anany Levitin. *Introduction to The Design & Analysis of Algorithms*. Pearson Education Inc., Addison-Wesley, 2003.
21. Andreas Nilsson. *Data Structures for Bandwidth Reservation and Quiality of Service on the Internet*. Lic. thesis, Department of Computer Science and Electrical Engineering, Luleå University of Technology, Luleå, Sweden, April 2004.
22. W. Paul and J. Simon. Decision trees and random access machines. In *Proc. Int'l. Symp. on Logic and Algorithmic*, pages 331–340, Zurich, 1980.
23. Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct dynamic data structure. In *Algorithms and Data Structures, 7th International Workshop*, volume 2125 of *Lecture Notes in Computer Science*, pages 426–437. Springer, 8–10 August 2001.
24. Mirek Riedewald, Divyakant Agrawal, and Amr El Abbadi. Flexible data cubes for online aggregation. In *Database Theory - ICDT 2001, 8th International Conference, London , UK, January 4-6, 2001, Proceedings*, volume 1973 of *Lecture Notes in Computer Science*, pages 159–173, 2001.
25. Mirek Riedewald, Divyakant Agrawal, Amr El Abbadi, and Renato Pajarola. Space-efficient data cubes for dynamic environments. In *Proceedings of the International Conference on Data Warehousing and Knowledge Discovery (DaWak)*, pages 24–33, 2000.
26. L. G. Valiant. General purpose parallel architectures. In van Leeuwen [28], chapter 18, pages 943–971.
27. Peter van Emde Boas. Machine models and simulations. In van Leeuwen [28], chapter 1, pages 3–66.
28. Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity. Elsevier/MIT Press, Amsterdam, 1990.
29. Andrew C. Yao. On the complexity of maintaining partial sums. *SIAM Journal on Computing*, 14(2):277–288, May 1985.